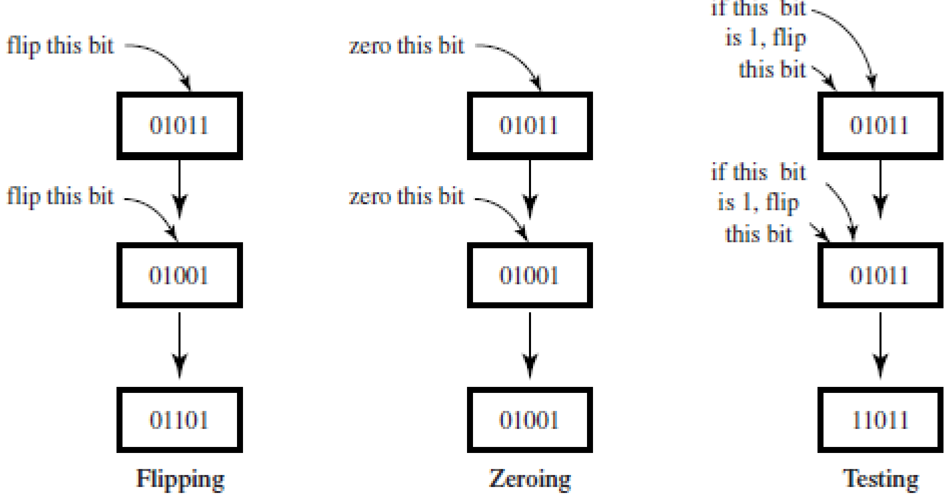
Wstęp i przegląd historyczny, czyli o co w tym wszystkim chodzi?

Komputery to niesamowite maszyny. Wydają się być w stanie zrobić wszystko. Latają samolotami i statkami kosmicznymi, kontrolują elektrownie i niebezpieczne zakłady chemiczne. Firmy nie mogą już działać bez nich, a coraz więcej skomplikowanych procedur medycznych nie może być wykonywanych pod ich nieobecność. Służą prawnikom i sędziom, którzy szukają precedensów sądowych w dziesiątkach udokumentowanych procesów, a także pomagają naukowcom w wykonywaniu niezwykle skomplikowanych i wymagających obliczeń matematycznych. Przekierowują i kontrolują miliony połączeń telefonicznych w sieciach obejmujących kontynenty. Wykonują zadania z niezwykłą precyzją - od czytania map i składu po graficzną obróbkę obrazu i projektowanie układów scalonych. . Mogą odciążyć nas od wielu nudnych obowiązków, takich jak skrupulatne śledzenie wydatków domowych, a jednocześnie zapewnić nam urozmaiconą rozrywkę, np. gry komputerowe lub skomputeryzowana muzyka. Co więcej, dzisiejsze komputery ciężko pracują, pomagając zaprojektować jeszcze potężniejsze komputery jutra. Tym bardziej niezwykłe jest to, że komputer cyfrowy – nawet ten najnowocześniejszy i najbardziej skomplikowany – może być traktowany jako po prostu duża kolekcja przełączników. Te przełączniki lub bity, jak się je nazywa, nie są „odwracane” przez użytkownika, ale są specjalnymi, wewnętrznymi przełącznikami, które są „odwracane” przez sam komputer. Każdy bit może znajdować się w jednej z dwóch pozycji lub, inaczej mówiąc, może przyjmować jedną z dwóch wartości, 0 lub 1. Zazwyczaj wartość bitu jest określona przez pewną charakterystykę elektroniczną, na przykład, czy dany punkt ma ładunek dodatni lub ujemny. Komputer może bezpośrednio wykonać tylko niewielką liczbę niezwykle trywialnych operacji, takich jak odwracanie, zerowanie lub testowanie. Odwracanie zmienia wartość bitu, zerowanie zapewnia, że bit kończy się na pozycji 0, a testowanie robi jedną rzecz, jeśli bit jest już na pozycji 0, a drugą, jeśli tak nie jest.

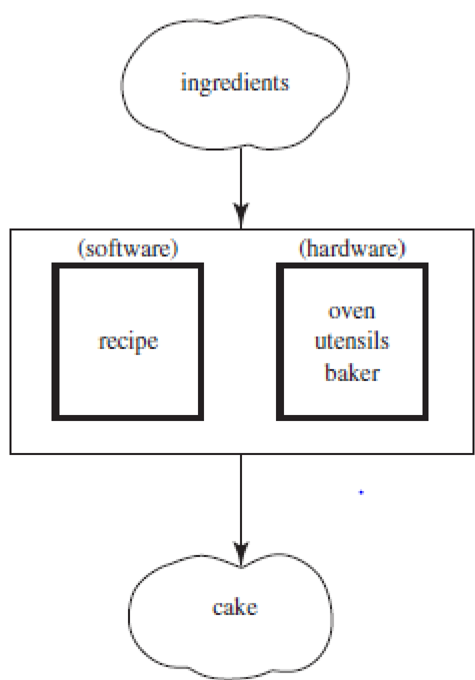


Komputery mogą różnić się wielkością (zgodnie z liczbą dostępnych bitów), rodzajami podstawowych operacji, które mogą wykonywać, szybkością wykonywania tych operacji, fizycznym nośnikiem zawierającym bity i ich wewnętrzną organizacją, oraz znacząco, w ich otoczeniu zewnętrznym. Ta ostatnia pozycja oznacza, że dwa komputery, które poza tym mają podobne funkcje, mogą wydawać się obserwatorowi bardzo różne: jeden może przypominać telewizor z klawiaturą, a drugi może być schowany pod pokrętłami i pokrętłami automatycznej dziewiarki. Jednak wygląd zewnętrzny ma znaczenie peryferyjne w porównaniu z bitami i ich wewnętrznym rozmieszczeniem. To właśnie bity „wyczuwają” zewnętrzne bodźce przychodzące ze świata zewnętrznego za pomocą przycisków, dźwigni, klawiszy na klawiaturze, elektronicznych linii komunikacyjnych, a nawet mikrofonów i kamer.

To właśnie bity „decydują” o tym, jak reagować na te bodźce i odpowiednio reagować, kierując inne bodźce na zewnątrz za pomocą wyświetlaczy, ekranów, drukarek, głośników, brzęczyków, dźwigni i korb. Jak robią to komputery? Co takiego przekształca tak trywialne operacje na bitach w niewiarygodne wyczyny, jakich dokonują komputery? Odpowiedź tkwi w głównych pojęciach : procesie i algorytmie, który go określa i powoduje, że ma miejsce.

Trochę gastronomii

Wyobraź sobie kuchnię, zawierającą zapas składników, szereg przyborów do pieczenia, piekarnik i (człowieka) piekarza. Pieczenie pysznego ciasta rodzynkowego to proces, który ze składników przeprowadza piekarz za pomocą piekarnika, a przede wszystkim zgodnie z przepisem. Składniki są danymi wejściowymi do procesu, ciasto jest jego wyjściem, a przepis jest algorytmem. Innymi słowy, algorytm określa czynności, które składają się na proces. Receptury lub algorytmy odnoszące się do zestawu omawianych procesów są ogólnie nazywane oprogramowaniem, podczas gdy przybory i piekarnik reprezentują to, co ogólnie nazywa się sprzętem. Piekarz w tym przypadku można uznać za część sprzętu.



Podobnie jak w przypadku operacji bitowych, konstelacja piekarz/piekarnik/przybory ma bardzo ograniczone możliwości bezpośrednie. Ten sprzęt do pieczenia ciast może nalewać, mieszać, rozsmarowywać, kapać, rozpalać piekarnik, otwierać drzwi piekarnika, mierzyć czas lub mierzyć ilości, ale nie może bezpośrednio piec ciast. To przepisy - te magiczne recepty, które zamieniają ograniczone możliwości sprzętu kuchennego w ciasta - a nie piekarniki czy piekarnie, są tematem tego tekstu. Przepisy, jak już wspomniano, nazywane są tutaj algorytmami, podczas gdy dziedzina badań nad ludźmi, wiedzy i ekspertyzy, która dotyczy algorytmów, będzie nazywana algorytmiką. Narysowana tu analogia została wykonana możliwie jak najdokładniej: przepis, który jest w pewnym sensie bytem abstrakcyjnym, jest algorytmem; formalna pisemna wersja przepisu, taka jak ta znaleziona w książce kucharskiej, jest analogiczna do programu komputerowego. Oprogramowanie w rzeczywistości odnosi się bardziej do programów — precyzyjnych reprezentacji algorytmów napisanych w specjalnych językach odczytywanych przez komputer — niż do samych algorytmów. Jednak dopóki nie omówimy

języków programowania, to rozróżnienie jest dość nieistotne. Konfrontujemy algorytmy, gdziekolwiek się udamy. Wiele codziennych procesów rządzi się algorytmami: wymiana przebitej opony, budowanie szafki zrób to sam, robienie na drutach swetra, dzielenie liczb, sprawdzanie numeru telefonu, aktualizowanie listy wydatków czy wypełnianie deklaracji podatkowej. Niektóre z nich (na przykład podział) mogą być w naszych umysłach bardziej bezpośrednio powiązane z komputerami, niż inne (na przykład budowa szaf), ale tutaj nas to dotyczy mniej. Chociaż komputery mają fundamentalne znaczenie dla tematu, w ogóle nie będziemy koncentrować się na ich fizycznych aspektach. To właśnie ich duchem zajmujemy się; z przepisami, które sprawiają, że działają zgodnie z ich algorytmami.

Algorytmika a informatyka

Algorytmika to coś więcej niż dziedzina informatyki. Jest to rdzeń informatyki i, uczciwie, można powiedzieć, że ma znaczenie dla większości nauki, biznesu i technologii. Sama natura algorytmiki sprawia, że jest ona szczególnie przydatna w tych dyscyplinach, które czerpią korzyści z wykorzystania komputerów, a te szybko stają się przytłaczającą większością. Wiadomo, że ludzie pytają: „Czym naprawdę jest informatyka? Dlaczego nie mamy nauki o łodziach podwodnych, nauki o zmywarkach ani nauki o telefonach?" Można argumentować, że telefony i zmywarki są tak samo ważne dla współczesnego życia jak komputery; może bardziej. Nieco bardziej skoncentrowanym pytaniem jest, czy informatyka obejmuje takie klasyczne dyscypliny, jak matematyka, fizyka, neurologia, elektrotechnika, językoznawstwo, logika i filozofia. My nie próbujemy odpowiedzieć na te pytania. Mamy jednak nadzieję, że domyślnie przekażemy coś z wyjątkowości i uniwersalności algorytmiki, a co za tym idzie, coś z wagi informatyki jako autonomicznej, choć młodej, dziedziny studiów. Ponieważ komputery mogłyby ograniczać ogólność algorytmiki, niektórzy uważają nieunikniony związek między nimi za niefortunny. W rzeczywistości określenie dziedziny „informatyka", jak powiedział ktoś kiedyś, jest jak odnoszenie się do chirurgii jako „nauki o nożach". Tak czy inaczej, jasne jest, że algorytmika nigdy nie rozwinęłaby się w taki sposób, jak bez tego łącza. Jednak ogólnie przyjmuje się, że termin „informatyka" jest mylący i że coś takiego jak „informatyka", „nauka o procesach" lub „nauka dyskretna" może być lepsze. Ponownie twierdzimy tylko, że nasza tematyka, algorytmika, stanowi podstawę informatyki, a nie ją zastępuje. Niektóre z tematów, które poruszamy w sequelu, takie jak istnienie problemów, których komputery nie mogą rozwiązać, mają implikacje filozoficzne, nie tylko na granicach wspaniałych maszyn, które jesteśmy w stanie zbudować, ale także na naszych własnych granicach jako śmiertelników o skończonej masie i skończonej żywotności. Pomimo głębokiej natury takich implikacji, w tej książce nacisk kładzie się na bardziej pragmatyczny cel, jakim jest zdobycie dogłębnego zrozumienia podstaw procesów wykonywanych maszynowo oraz receptur lub algorytmów, które nimi rządzą.

Trochę historii

Przyjrzyjmy się teraz kilku ważnym kamieniom milowym w rozwoju komputerów i informatyki, głównie po to, aby zilustrować, że jako uporządkowana dyscyplina naukowa jest to niezwykle młoda dziedzina. Gdzieś pomiędzy 400 a 300 rokiem p.n.e. wielki grecki matematyk Euklides wynalazł algorytm znajdowania największego wspólnego dzielnika (nwd) dwóch dodatnich liczb całkowitych. gcd X i Y jest największą liczbą całkowitą, która dokładnie dzieli X i Y . Na przykład gcd 80 i 32 wynosi 16. Szczegóły samego algorytmu nie mają tutaj znaczenia, ale algorytm Euklidesa, jak się go nazywa, jest uważany za pierwszy nietrywialny algorytm, jaki kiedykolwiek opracowano. Słowo algorytm wywodzi się od nazwiska perskiego matematyka Mohammeda al-Chuarizmi, który żył w IX wieku i któremu przypisuje się dostarczanie szczegółowych zasad dodawania, odejmowania, mnożenia i dzielenia zwykłych liczb dziesiętnych. Napisana po łacinie nazwa przekształciła się w Algorismus, od którego algorytm jest tylko małym krokiem. Najwyraźniej Euklides i al-Chuarizmi byli algorytmistami par excellence. Przechodząc od oprogramowania do sprzętu, jedną z pierwszych maszyn, które przeprowadzały proces

kontrolowany przez coś, co można by nazwać algorytmem, było krosno tkackie wynalezione w 1801 roku przez Francuza Josepha Jacquarda. Utkany wzór wyznaczały karty z dziurkami w różnych miejscach. Otwory te, wyczuwane przez specjalny mechanizm, sterowały doborem nici i innymi czynnościami maszyny. Interesujące jest to, że krosno Jacquard nie miało nic wspólnego z wąską konotacją numeryczną terminu „obliczenia". Jedną z najważniejszych i najbardziej barwnych postaci w historii informatyki był Charles Babbage. Ten angielski matematyk, po częściowym zbudowaniu w 1833 r. maszyny zwanej „silnikiem różnicowym" do oceny pewnych wzorów matematycznych, wymyślił i zaplanował niezwykłą maszynę, którą nazwał „silnikiem analitycznym". W przeciwieństwie do silnika różnicowego, który został zaprojektowany do realizacji określonego zadania, silnik analityczny miał być zdolny do wykonywania algorytmów lub programów, zakodowanych przez użytkownika jako dziurki w kartach. Gdyby skonstruowano silnik analityczny, byłby to matematyczny odpowiednik krosna Jacquarda, które w rzeczywistości było jego inspiracją. Nie trzeba dodawać, że maszyna Babbage'a była z natury mechaniczna, oparta na dźwigniach, zębatkach i przekładniach, a nie na elektronice i krzemie. Niemniej idee obecne w jego projekcie silnika analitycznego stanowią podstawę wewnętrznej struktury i działania współczesnych komputerów. Powszechnie uważa się, że Babbage żył na długo przed swoim czasem, a jego pomysły nie zostały docenione znacznie później. Ada Byron, hrabina Lovelace, była programistką Babbage'a. Jest jedną z najciekawszych postaci w historii informatyki, której przypisuje się położenie podwalin pod programowanie ponad sto lat przed pojawieniem się pierwszego działającego komputera. Amerykański inżynier Herman Hollerith wynalazł maszynę, również opartą na kartach dziurkowanych, która została wykorzystana przez Amerykańskie Biuro Spisu Ludności (American Census Bureau) do spisu ludności z 1890 roku. Jednak pierwsze komputery ogólnego przeznaczenia zbudowano dopiero w latach 40. XX wieku, częściowo jako odpowiedź na potrzeby obliczeniowe fizyków i astronomów, a częściowo jako naturalny wynik dostępności odpowiednich urządzeń elektromechanicznych i elektronicznych. Jak na ironię, pomogła też druga wojna światowa, z jej działalnością polegającą na budowaniu bomb i łamaniu kodów. Niektóre z kluczowych postaci w tym przełomowym i ekscytującym okresie to Anglik Alan Turing, Amerykanie Howard Aiken, John Mauchly, J. Presper Eckert i Herman Goldstine oraz słynny niemiecko-amerykański matematyk John von Neumann. Wracając do oprogramowania i algorytmiki, połowa lat 30. była świadkiem jednych z najbardziej fundamentalnych prac nad teorią algorytmów, przynoszących wyniki dotyczące możliwości i ograniczeń algorytmów wykonywanych maszynowo. Godne uwagi jest to, że praca ta, której fragmenty zostaną opisane w dalszej części , poprzedzała rzeczywistą materializację komputera. Niemniej jednak ma uniwersalne i trwałe znaczenie. Niektóre z kluczowych postaci, wszyscy matematycy, to znowu Alan Turing, Niemiec Kurt Gödel, Rosjanin Andrjej A. Markov i Amerykanie Alonzo Church, Emil Post i Stephen Kleene. Lata pięćdziesiąte i sześćdziesiąte były świadkami daleko idących i szybkich postępów technologicznych w projektowaniu i budowie komputerów. Można to przypisać z jednej strony nadejściem ery badań jądrowych i eksploracji kosmosu, a z drugiej boomowi dużych firm i banków oraz zróżnicowanej działalności rządowej. Precyzyjne przewidywanie różnych zjawisk jądrowych wymagało bardzo dużej mocy obliczeniowej, podobnie jak planowanie i symulacja misji kosmicznych. Eksploracja kosmosu wymagała również postępów w komunikacji wspomaganej komputerowo, ułatwiającej niezawodną analizę i filtrowanie, a nawet ulepszanie danych przesyłanych do i z satelitów i statków kosmicznych. Działalność biznesowa, bankowa i rządowa wymagała komputerów, które pomagałyby w przechowywaniu, manipulowaniu i wyszukiwaniu informacji dotyczących bardzo dużej liczby osób, pozycji inwentarzowych, danych fiskalnych i tak dalej. Interesujące dowody na znaczenie rozwoju technologicznego zorientowanego na maszyny w tym okresie można znaleźć w nazwach największej na świecie firmy komputerowej, IBM, i jednej z największych na świecie profesjonalnych organizacji związanych z komputerami, ACM. Pierwsza powstała około 1920 r., druga pod koniec lat 40. XX wieku. W obu przypadkach „M" pochodzi od słowa „machine": International Business Machines i Association for Computing Machinery. (IBM

wyewoluował z firmy utworzonej w 1896 r. przez wspomnianego Hermana Holleritha, aby produkować jego maszyny do tabulacji). Uznanie informatyki jako niezależnej dyscypliny akademickiej nastąpiło w połowie lat 60., kiedy kilka uniwersytetów utworzyło wydziały informatyki. W 1968 roku ACM opublikowała cieszącą się szerokim uznaniem rekomendację dotyczącą programu nauczania przedmiotów z informatyki, która stanowi podstawę większości aktualnych programów studiów z zakresu informatyki na poziomie licencjackim. Ten program jest okresowo aktualizowany. Dziś prawie każda instytucja akademicka posiada wydział informatyki lub grupę informatyczną w ramach wydziału matematyki lub elektrotechniki. Lata 60. pokazały ponowne zainteresowanie pracami z lat 30. nad algorytmiką i od tego czasu ta dziedzina jest przedmiotem szeroko zakrojonych i dalekosiężnych badań. Nie będziemy się więcej rozwodzić nad obecną sytuacją technologiczną: komputery są po prostu wszędzie. Używamy ich do surfowania po Internecie, co oznacza, że używamy ich do odbierania i dostarczania informacji, czytania, słuchania i oglądania oraz oczywiście przeglądania i kupowania. Istnieją komputery stacjonarne, laptopy i komputery wielkości dłoni, więc nigdy nie musimy ich bez nich obejść, a szybko zmniejszająca się przepaść między telefonami komórkowymi a komputerami zwiastuje erę komputerów do noszenia. Prawie każde nowoczesne urządzenie jest sterowane przez komputer, a na przykład jeden nowoczesny samochód zawiera ich dziesiątki. Dzieci proszą o komputery osobiste na urodziny i otrzymują je; studenci informatyki na większości uczelni muszą posiadać własne komputery do odrabiania prac domowych; i nie ma działalności przemysłowej, naukowej czy handlowej, która nie jest w sposób zasadniczy wspomagana przez komputery.

Dziwna dychotomia

Pomimo tego wszystkiego (lub prawdopodobnie w wyniku tego) opinia publiczna jest dziwnie podzielona, jeśli chodzi o umiejętność obsługi komputera. Wciąż są tacy, którzy nie wiedzą absolutnie nic o komputerach, a także członkowie stale rosnącej klasy informatyków. Poczynając od 10-letnich właścicieli komputerów osobistych, ta powiększająca się grupa osób korzystających na co dzień z komputerów obejmuje menedżerów, inżynierów, bankierów, techników i oczywiście profesjonalnych programistów, analityków systemowych i członków samego przemysłu komputerowego. Dlaczego to dziwne? Otóż, oto nauka, o której niektórzy ludzie nic nie wiedzą, ale o której szybko rosnąca liczba ludzi najwyraźniej wie wszystko! Tak się jednak składa, że naprawdę niezwykłym zjawiskiem jest to, że duże i ważne części informatyki nie są wystarczająco znane nie tylko członkom pierwszej grupy, ale także członkom drugiej grupy. Jednym z celów jest próba naświetlenia ważnego aspektu rewolucji komputerowej poprzez przedstawienie podstawowych pojęć, wyników i trendów leżących u podstaw nauki o obliczeniach. Skierowany jest zarówno do nowicjusza, jak i eksperta. Czytelnik bez wiedzy o komputerach (miejmy nadzieję) dowie się tutaj o ich „duchu" i sposobie myślenia, który zmusza ich do pracy, szukając gdzie indziej materiałów dotyczących ich „ciała". Czytelnik znający się na komputerach, który może uznać, że pierwsze kilka rozdziałów jest raczej powolne, (mamy nadzieję) będzie mógł się wiele nauczyć od późniejszych.

Niektóre ograniczenia komputerów

Zanim rozpoczniemy naszą trasę, skontrastujmy pierwszy akapit tej części z niektórymi wyczynami, których obecne komputery nie są jeszcze w stanie wykonać. Powrócimy do tych kontrastów w ostatniej części, który dotyczy relacji między komputerami a ludzką inteligencją. Obecnie komputery są w stanie przeanalizować na miejscu ogromną ilość danych pochodzących z wielu zdjęć rentgenowskich mózgu pacjenta, zrobionych pod stopniowo rosnącymi kątami. Analizowane dane są następnie wykorzystywane przez komputer do wygenerowania przekrojowego obrazu mózgu, dostarczającego informacji o strukturze tkanek mózgu, umożliwiając w ten sposób precyzyjne zlokalizowanie takich nieprawidłowości jak guzy czy nadmiar płynów. W przeciwieństwie do tego, żaden obecnie dostępny komputer nie jest w stanie przeanalizować pojedynczego, zwykłego obrazu twarzy tego samego

pacjenta i określić jego wieku z marginesem błędu, powiedzmy, wynoszącym pięć lat. Jednak większość 12-letnich dzieci może! Jeszcze bardziej uderzająca jest zdolność rocznego dziecka do rozpoznawania twarzy matki na zdjęciu, którego nigdy wcześniej nie widziała, wyczyn, którego komputery nie są w stanie naśladować (i to nie tylko dlatego, że nie mają matek...) . Komputery są w stanie sterować w najbardziej precyzyjny i efektywny sposób, jaki można sobie wyobrazić, niezwykle wyrafinowanymi robotami przemysłowymi używanymi do konstruowania skomplikowanych części maszyn składających się z setek komponentów. W przeciwieństwie do tego, dzisiejsze najbardziej zaawansowane komputery nie są w stanie pokierować robotem, aby skonstruował ptasie gniazdo ze stosu gałązek, czego może dokonać każdy 12-miesięczny ptak! Dzisiejsze komputery potrafią grać w szachy na poziomie międzynarodowego arcymistrza, a tym samym mogą pokonać ogromną większość ludzkich graczy. Jednak przy bardzo nieznacznej zmianie zasad gry (na przykład poprzez umożliwienie skoczkowi dwóch ruchów naraz lub ograniczenie ruchów hetmana do pięciu pól), najlepsze z tych komputerów nie będą w stanie się przystosować bez przeprogramowania. lub zrekonstruowane przez ludzi. W przeciwieństwie do tego, 12-letni szachista amator będzie w stanie rozegrać całkiem dobrą partię z nowymi zasadami w bardzo krótkim czasie, a wraz z doświadczeniem stanie się coraz lepszy. Jak wspomniano, te różnice są związane z różnicą między inteligencją ludzką a skomputeryzowaną. Będziemy w lepszej pozycji do dalszego omówienia tych kwestii w rozdziale 15, po tym, jak dowiemy się więcej o algorytmach i ich właściwościach.

Przepis

Oto przepis na mus czekoladowy zaczerpnięty z francuskiej kuchni Sinclaira i Malinowskiego. Składniki - to znaczy wkłady - obejmują 8 uncji półsłodkich kawałków czekolady, 2 łyżki wody, 14 filiżanek cukru pudru, 6 oddzielnych jajek i tak dalej. Daje od sześciu do ośmiu porcji pysznego musu z czekoladą. Oto przepis lub algorytm. Rozpuść czekoladę i 2 łyżki wody w podwójnym bojlerze. Po roztopieniu dodaj cukier puder; dodawać po trochu masło. Odłożyć na bok. Ubijaj żółtka do gęstej i cytrynowej barwy, około 5 minut. Delikatnie zawinąć w czekoladę. W razie potrzeby lekko podgrzej, aby rozpuścić czekoladę. Dodaj rum i wanilię. Białka ubić do uzyskania piany. Ubij 2 łyżki cukru; ubijaj, aż uformują się sztywne szczyty. Delikatnie złóż białka w mieszankę czekoladowo-żółtkową. Wlać do poszczególnych porcji dań. Schłodź co najmniej 4 godziny. W razie potrzeby podawać z bitą śmietaną. Robi od 6 do 8 porcji.

Jest to „oprogramowanie” związane z przygotowaniem musu; jest to algorytm, który kontroluje proces wytwarzania musu ze składników. Sam proces jest wykonywany przez „sprzęt”, w tym przypadku osobę przygotowującą mus, wraz z różnymi przyborami: podwójnym bojlerem, urządzeniem grzewczym, trzepaczką, łyżkami, minutnikiem i tak dalej.

Poziomy szczegółowości

Przyjrzyjmy się bliżej najbardziej elementarnym instrukcjom zawartym w tym przepisie. Rozważ instrukcję „domieszaj cukier puder”. Dlaczego przepis nie mówi „weź trochę cukru pudru, wlej go do roztopionej czekolady, wmieszaj, weź trochę więcej, wlej, wymieszaj, . . .? A dokładniej, dlaczego nie jest napisane „weź 2365 ziaren cukru pudru, wlej je do roztopionej czekolady, weź łyżkę i mieszaj okrężnymi ruchami, . . .? Lub, by być jeszcze bardziej precyzyjnym, dlaczego nie „przesunąć ręką w kierunku składników pod kątem 14◦, z przybliżoną prędkością 18 cali na sekundę, . . .? Odpowiedź jest oczywiście oczywista. Sprzęt wie, jak wymieszać cukier puder z rozpuszczoną czekoladą i nie wymaga dalszych szczegółów. A co powiesz na odwrócenie sprawy i pytanie, czy to możliwe, że sprzęt wie, jak przygotować mieszankę czekolady z cukrem i masłem? W takim przypadku całą pierwszą część przepisu można by zastąpić prostą instrukcją „przygotuj mieszankę czekoladową”. Doprowadzając to do skrajności, może sprzęt wie, jak przygotować mus czekoladowy. Umożliwiłoby to zastąpienie całego

przepisu „przygotuj mus czekoladowy”. Przy takim poziomie wiedzy o sprzęcie, pojedyncza linijka instrukcji jest idealnym przepisem na uzyskanie musu z czekolady; ten krótki przepis jest jasny, nie zawiera błędów i gwarantuje uzyskanie pożądanych wyników. Takie eksperymenty myślowe jasno pokazują, że poziom szczegółowości jest bardzo ważny, jeśli chodzi o podstawowe instrukcje algorytmu. Musi być dostosowany do konkretnych możliwości sprzętu, a także powinien być dostosowany do poziomu zrozumienia potencjalnego czytelnika lub użytkownika algorytmu. Rozważmy inny przykład poznany na początku naszego życia, który jest nieco bliższy obliczeniom - uporządkowane mnożenie liczb. Załóżmy, że zostaniemy poproszeni o pomnożenie 528 przez 46. Dokładnie wiemy, co robić. Pomnożymy 8 przez 6, otrzymując 48, zapisujemy cyfrę jednostki wyniku 8 i pamiętamy cyfrę dziesiątek 4; następnie mnożymy 2 przez 6 i dodajemy 4, otrzymując 16; zapisujemy cyfrę jednostek 6 po lewej stronie 8 i pamiętamy cyfrę dziesiątek 1; i tak dalej. Tutaj można zadać te same pytania. Dlaczego „mnożyć 8 przez 6?” Dlaczego nie „sprawdzić wpisu znajdującego się w ósmym wierszu i szóstej kolumnie tabliczki mnożenia” lub „dodać 6 do siebie 8 razy”? Podobnie, dlaczego nie możemy rozwiązać całego problemu za jednym pociągnięciem za pomocą prostego i satysfakcjonującego algorytmu „pomnóż dwie liczby?” To ostatnie pytanie jest dość subtelne: dlaczego możemy bezpośrednio pomnożyć 8 przez 6, ale nie 528 przez 46? Ponownie, jasne jest, że poziom szczegółowości jest kluczową cechą naszej akceptacji algorytmu mnożenia. Zakładamy, że odpowiedni sprzęt (w tym przypadku my sami) jest w stanie wykonać 8 razy 6, ale nie 528 razy 46, i że możemy to zrobić w naszych głowach, a przynajmniej znamy jakiś inny sposób, aby nie trzeba było nam mówić, jak sprawdzić wynik w tabeli. Te przykłady pokazują potrzebę uzgodnienia na samym początku podstawowych działań, które algorytm uważa za zdolny do przepisania. Bez tego nie ma sensu szukać algorytmów do czegokolwiek. Co więcej, różne problemy są naturalnie związane z różnymi rodzajami podstawowych działań. Przepisy wymagają mieszania, mieszania, nalewania i podgrzewania; mnożenie liczb pociąga za sobą dodawanie, mnożenie cyfr i, co ważne, zapamiętywanie cyfry; wyszukanie numeru telefonu może wymagać odwrócenia strony, przesunięcia palcem w dół listy i porównania podanego nazwiska z tym, na które wskazujemy. W precyzyjnych rodzajach algorytmów, które będziemy omawiać, te podstawowe instrukcje muszą być sformułowane jasno i precyzyjnie. Nie możemy zaakceptować rzeczy takich jak „ubijanie białek do uzyskania piany”, ponieważ wyobrażenie jednej osoby na pianę może być zupełnie niepodobne do innej! Instrukcje muszą być odpowiednio odróżnialne od nieinstrukcyjnych, takich jak „robi 6 do 8 porcji”. Zwroty rozmyte, takie jak „około 5 minut”, nie mają miejsca w algorytmie przystosowanym do wykonywania przez komputer, jak ma to miejsce w przypadku niejasności, takich jak „podawaj z bitą śmietaną, jeśli chcesz”. (Czy to faktyczna porcja, czy dodanie bitej śmietany, zależy od pragnień danej osoby?) Przepisy na mus, w przeciwieństwie do algorytmów, które będą nas interesować, zbyt wiele rzeczy uznają za oczywiste, z których najważniejsza czyli fakt, że człowiek jest częścią sprzętu. Nie możemy polegać na tego rodzaju luksusie, dlatego musimy być znacznie bardziej wymagający. Ogólna jakość algorytmu zależy przede wszystkim od wyboru dozwolonych działań podstawowych i ich adekwatności do danej sprawy.

Abstrakcja

Wcześniej stwierdzono, że prawdziwe komputery mogą wykonywać tylko niezwykle proste operacje na niezwykle prostych obiektach. Może się to wydawać sprzeczne z obecną dyskusją, w której zaleca się projektowanie różnych algorytmów przy użyciu podstawowych działań o różnym poziomie szczegółowości. Jednak analogia jest nadal aktualna. Uczniowi szefa kuchni może być konieczne podanie przepisu na mus czekoladowy, ale po kilku latach przygotowania musu wystarczy instrukcja „przygotuj mus czekoladowy”. Mówimy, że pojęcia takie jak „mus czekoladowy”, „beza cytrynowa” i „krem bawarski” są na wyższym poziomie abstrakcji niż operacje takie jak „mieszanie”, „mieszanie” i „przelewanie” stosowane w przepisach na ich przygotowanie. W ten sam sposób, dzięki odpowiedniemu programowaniu, komputer może rozpoznawać abstrakcje wyższego poziomu, takie

jak liczby, tekst i obrazy. Podobnie jak w gotowaniu, w komputerze istnieje wiele poziomów abstrakcji, z których każdy jest odpowiedni do opisywania różnych rodzajów algorytmów. Na przykład ten sam komputer jest postrzegany inaczej przez 12-latka grającego w grę komputerową, przez jego siostrę, która surfuje po Internecie, przez ojca, który używa arkusza kalkulacyjnego do obliczania ocen uczniów, a przez matkę. kto pisze program zarządzania testem skuteczności nowej szczepionki. Żadne z nich nie zna ani nawet nie dba o bity, które naprawdę składają się na proces obliczeniowy, którego używają. Ten proces abstrahowania od szczegółów w celu dostrzeżenia wspólnych wzorców w pozostałej części jest sednem prawie każdego ludzkiego przedsięwzięcia. Na przykład czytanie książki ma wpływ na mózg, który składa się z kilku odrębnych regionów, z których każdy składa się z neuronów i innych komórek. Komórki te zbudowane są ze złożonych cząsteczek, które zbudowane są z atomów, które z kolei zbudowane są z bardziej elementarnych cząstek. Wszystkie te różne poziomy abstrakcji odnoszą się do tego, co dzieje się w twoim mózgu, ale nie można ich wszystkich rozpatrywać łącznie. W rzeczywistości należą one do różnych dziedzin nauki: fizyki cząstek elementarnych, chemii, biologii molekularnej, neurobiologii i psychologii. Psycholog przeprowadzający eksperymenty z krótkotrwałą retencją pamięci będzie rozpraszany tylko przez myślenie o związkach między atomami i cząsteczkami w mózgu. To samo dotyczy informatyki. Gdybyśmy byli zmuszeni do myślenia na poziomie bitowym przez cały czas, komputer nie byłby przydatny. Zamiast tego możemy na przykład pomyśleć o grupie bitów (zwykle osiem bitów lub „bajt") jako oznaczającej znak. Możemy teraz rozważyć sekwencje bajtów do oznaczania słów angielskich, sekwencje słów i znaki interpunkcyjne do oznaczania zdań itd. do akapitów, rozdziałów i książek. Dla każdego z tych poziomów istnieją algorytmy. Na przykład sprawdzanie pisowni dotyczy słów, ale nie znaków, justowanie do lewej dotyczy akapitów, a tworzenie spisu treści dotyczy książek. W każdym przypadku możemy opisać algorytm, całkowicie ignorując fragmenty składające się na słowa, akapity lub całe książki. W miarę rozwoju, będziemy omawiać środki techniczne, które pozwalają nam tworzyć takie abstrakcje. Tymczasem opiszemy każdy algorytm na odpowiednim dla niego poziomie abstrakcji.

Krótkie algorytmy dla długich procesów

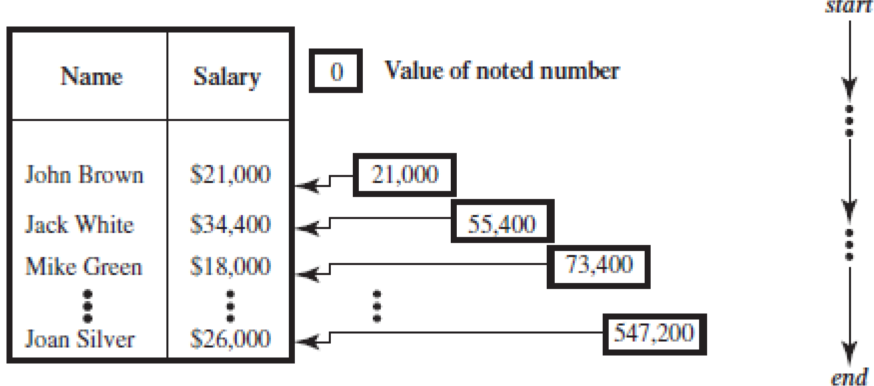
Załóżmy, że otrzymujemy listę akt personalnych, po jednym dla każdego pracownika w określonej firmie, z których każda zawiera imię i nazwisko pracownika, dane osobowe oraz wynagrodzenie. Interesuje nas łączna suma wszystkich wynagrodzeń wszystkich pracowników. Oto algorytm wykonania tego zadania:

(1) zanotuj cyfrę 0;

(2) przejdź przez listę, dodając wynagrodzenie każdego pracownika do zanotowanej liczby;

(3) po dojściu do końca listy, wygeneruj odnotowaną liczbę jako wynik.

Zanim przejdziemy dalej, powinniśmy najpierw przekonać się, że ten prosty algorytm spełnia swoje zadanie. „Zanotowana" liczba, którą można uznać za zapamiętaną lub zapisaną na kartce papieru, zaczyna się od wartości zero. Po dokonaniu doliczenia w ust. 2 dla pierwszego pracownika, liczba ta faktycznie przyjmuje wartość wynagrodzenia tego pracownika. Po drugim pracowniku jego wartość jest sumą wynagrodzeń pierwszych dwóch pracowników. Ostatecznie jego wartość jest wyraźnie sumą wszystkich wynagrodzeń.

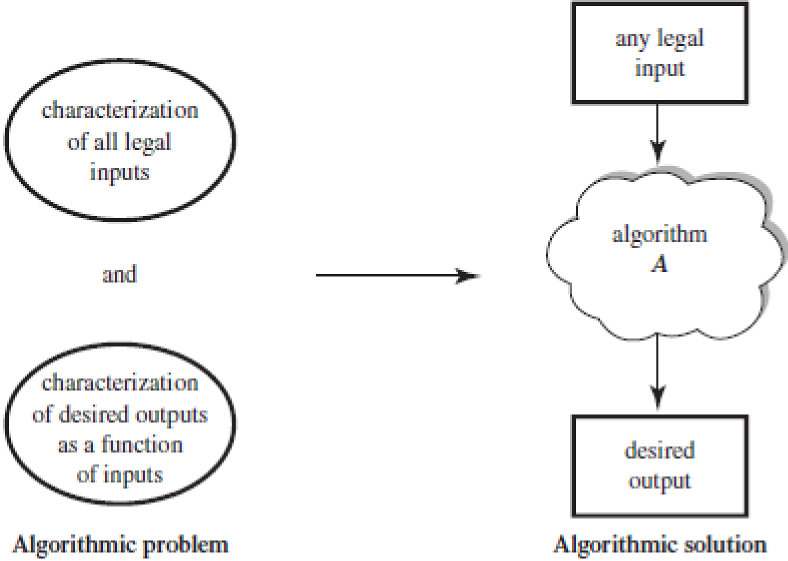


Interesujące jest to, że tekst tego algorytmu jest krótki i ma stałą długość, ale proces, który opisuje i kontroluje, zmienia się wraz z długością listy pracowników i może być bardzo, bardzo długi. Dwie firmy, pierwsza z jednym pracownikiem, a druga z milionem, mogą wprowadzić swoją listę pracowników do tego samego algorytmu, a problem sumowania wynagrodzeń zostanie rozwiązany dla każdej z nich równie dobrze. Oczywiście w przypadku pierwszej firmy proces ten nie potrwa długo, natomiast w przypadku drugiej będzie to dość długi czas. Algorytm jest jednak ustalony. Nie tylko tekst algorytmu jest krótki i ma stałą wielkość, ale zarówno mała jak i duża firma potrzebuje tylko jednej zanotowanej liczby, aby wykonać zadanie, dzięki czemu ilość „przyborów" jest tutaj również mała i stała. Oczywiście potencjalna wartość podanej liczby będzie prawdopodobnie musiała być większa dla większych firm, ale cały czas będzie tylko jedna liczba

Problem algorytmiczny

I tak mamy ustalony algorytm określający wiele procesów o różnej długości, dokładny czas trwania i charakter procesu w zależności od danych wejściowych do algorytmu. Rzeczywiście, nawet prosty przykład sumowania wynagrodzeń pokazuje różne możliwe dane wejściowe: firmy jednoosobowe, firmy liczące milion osób, firmy, w których część pensji wynosi zero lub te, w których wszystkie pensje są równe. Czasami algorytm musi również działać z dziwacznymi danymi wejściowymi, takimi jak firmy bez pracowników lub takie, które zatrudniają ludzi otrzymujących ujemne pensje (czyli pracowników, którzy płacą firmie za przyjemność pracy dla niej). W rzeczywistości algorytm wynagrodzeń powinien działać zadowalająco dla nieskończonej liczby wejść. Istnieje nieskończona liczba całkowicie akceptowalnych list pracowników, a algorytm powinien być w stanie zsumować pensje w dowolnej z nich, gdy zostanie podany jako dane wejściowe. Ta kwestia nieskończenie wielu potencjalnych danych wejściowych nie do końca pasuje do analogii przepisu, ponieważ chociaż przepis powinien działać doskonale bez względu na to, ile razy jest używany, jego składniki są zwykle opisywane jako ustalone w ilości, a zatem w istocie przepis ma tylko jeden potencjalny wkład (przynajmniej w ilościach; oczywiście cząsteczki i atomy będą za każdym razem inne). Jednak przepis na mus czekoladowy mógł być ogólny; to znaczy, jego lista składników mogła brzmieć coś w stylu „X uncji kawałków czekolady, X/4 łyżek wody, X/32 filiżanki cukru pudru itp.", a jego ostatnia linia mogła brzmieć „sprawia, że 3X/4 do X porcji." Byłoby to bardziej zgodne z prawdziwym pojęciem algorytmu. W obecnej formie przepis jest algorytmem o nieco banalnym charakterze, ponieważ jest dostosowany do jednego konkretnego zestawu składników. Może być przeprowadzana (lub, w terminologii algorytmicznej, może być uruchamiana lub wykonywana) kilka razy, ale z zasadniczo tymi samymi danymi wejściowymi, ponieważ jedna filiżanka mąki jest uważana za dokładnie taką samą jak każda inna. Same dane wejściowe muszą być zgodne z przeznaczeniem algorytmu. Oznacza to na przykład, że lista bestsellerów New York Times nie byłaby akceptowana jako dane wejściowe do algorytmu sumowania

wynagrodzeń, podobnie jak masło orzechowe i galaretka nie byłyby akceptowane jako składniki przepisu na mus. Pociąga to za sobą pewien rodzaj specyfikacji dozwolonych wejść. Ktoś musi dokładnie określić, które listy pracowników są legalne, a które nie; gdzie dokładnie na liście znajduje się wynagrodzenie; czy jest podana odręcznie (na przykład 32 000 USD), czy może w jakiejś skróconej formie (na przykład 32 000 USD); gdzie kończy się rekord pracownika, a zaczyna kolejny i tak dalej. Mówiąc najogólniej, przepisy lub algorytmy są rozwiązaniami na pewno rodzaje problemów, zwanych problemami obliczeniowymi lub algorytmicznymi. W przykładzie płacowym problem można sprecyzować w postaci prośby o numer reprezentujący sumę wynagrodzeń z listy pracowników organizacji. Lista ta może mieć różną długość, ale musi być ułożona w określony sposób. Problem taki można postrzegać jako poszukiwanie zawartości „czarnej skrzynki", która jest określona przez precyzyjną definicję legalnych danych wejściowych i precyzyjną definicję wymaganych wyników jako funkcji tych danych wejściowych; czyli sposób, w jaki każde wyjście zależy od wejścia.



Problem algorytmiczny został rozwiązany po znalezieniu odpowiedniego algorytmu. Czarna skrzynka została wtedy faktycznie zaopatrzona w zawartość; to „działa" zgodnie z tym algorytmem. Innymi słowy, czarna skrzynka może wytworzyć odpowiednie dane wyjściowe z dowolnego legalnego wkładu, wykonując proces, który jest określony i zarządzany przez ten algorytm. Słowo „dowolny" w poprzednim zdaniu jest bardzo ważne. Nie interesują nas rozwiązania, które nie działają dla wszystkich podanych wejść. Łatwo jest znaleźć rozwiązanie, które działa dobrze tylko w przypadku niektórych legalnych danych wejściowych. Jako skrajny przykład trywialny algorytm:

(1) wyprodukuj 0 jako wyjście.

bardzo dobrze sprawdza się w przypadku kilku interesujących list pracowników: tych bez pracowników, tych, w których każdy zarabia 0,00 zł (lub ich wielokrotności), a także tych z listą płac, która odzwierciedla idealną równowagę między pensją dodatnią a ujemną. Później zajmiemy się takimi zagadnieniami, jak wydajność i praktyczność algorytmów. Tutaj stawiamy minimalny wymóg, że algorytm faktycznie rozwiązuje problem, nawet jeśli może to zrobić nieefektywnie. Oczywiście sam problem może określać wymagane zachowanie potencjalnego algorytmu na niepożądanych danych wejściowych, ale wtedy te dane wejściowe, chociaż niepożądane, są nadal legalne. Na przykład problem sumowania wynagrodzeń mógłby zawierać wymóg, aby w przypadku pracownika, którego

rekord nie zawierał numeru w obszarze wynagrodzeń, ale powiedzmy znak zapytania lub inne bezsensowne dane, algorytm powinien dodać nazwisko tego pracownika do specjalnej listy, która zostanie przekazana do biura płac do dalszych działań. Taka niekonwencjonalna lista pracowników jest jednak legalna; po prostu nie jest traktowana w standardowy sposób, ale jest traktowana w specjalny sposób, który pasuje do jego nienormalnej natury. W związku z tym trzymanie nielegalnych danych wejściowych oddzielnie jest odpowiedzialnością problemu algorytmicznego, podczas gdy traktowanie specjalnych klas nietypowych lub niepożądanych danych wejściowych jest odpowiedzialnością samego algorytmu.

Granice podstawowych działań

Jest jeszcze jedna ważna sprawa, którą musimy się w tym miejscu poruszyć, dotycząca wykonania podstawowych czynności lub operacji, zaleconych przez algorytm. Oczywistym jest, że każda z tych czynności musi być wykonana w skończonym czasie, inaczej oczywiście algorytm nigdy się nie skończy. Tak więc nieskończenie długie działania są złe. Działania, które mogą zająć nieskończenie małą ilość czasu, są również zakazane, co nie wymaga uzasadnienia. Jest nie do pomyślenia, aby maszyna kiedykolwiek była w stanie wykonywać czynności w coraz krótszym czasie. Na przykład prędkość światła zawsze byłaby ograniczeniem prędkości każdej maszyny. Podobne ograniczenia zasobów (czyli narzędzi) wykorzystywanych do wykonywania podstawowych czynności również muszą być narzucone, ale nie będziemy tu omawiać przyczyn. Jasne jest, że te założenia dotyczące podstawowych działań rzeczywiście obowiązują w przypadku prawdziwych komputerów. Na przykład podstawowe akcje manipulacji bitami są precyzyjne i jednoznaczne oraz zajmują ograniczoną ilość czasu i zasobów. Tak więc, zgodnie z obietnicą, opisana tu teoria algorytmiki będzie miała bezpośrednie zastosowanie do problemów przeznaczonych do rozwiązania komputerowego.

Problem i jego rozwiązanie: podsumowanie

Podsumowując, problem algorytmiczny składa się z:

1. scharakteryzowanego legalnego, możliwie nieskończonego zbioru potencjalnych zbiorów wejściowych, oraz

2. specyfikację pożądanych wyjść jako funkcję wejść.

Zakłada się, że z góry podany jest również opis dozwolonych akcji podstawowych lub konfiguracja sprzętowa wraz z jej wbudowanymi akcjami podstawowymi. Rozwiązaniem problemu algorytmicznego jest algorytm składający się z elementarnych instrukcji zalecających działania z uzgodnionego zbioru. Algorytm ten, gdy jest wykonywany dla dowolnego dozwolonego zestawu danych wejściowych, rozwiązuje problem, wytwarzając wymagane dane wyjściowe.

Ważne jest, aby rozpoznać znaczne trudności związane z zadowalającym rozwiązywaniem problemów algorytmicznych. Zaczynając od przepisu na mus, a następnie podając prosty algorytm sumowania, popełniono pewną niesprawiedliwość, ponieważ mogłoby się wydawać, że sprawy są łatwe. Nic nie jest dalsze od prawdy. W praktyce problemy algorytmiczne mogą być niezwykle złożone, a ich pomyślne rozwiązanie może zająć lata pracy. Co gorsza, jak zobaczymy dalej, wiele problemów nie daje zadowalających rozwiązań, podczas gdy inne w ogóle nie dopuszczają żadnych rozwiązań. W przypadku wielu problemów status, jeśli chodzi o dobre rozwiązania algorytmiczne, jest jeszcze nieznany, mimo intensywnej pracy wielu utalentowanych ludzi. Oczywiście nie będziemy w stanie zilustrować zagadnień poruszanych tu zbyt obszernymi i złożonymi przykładami, ale możemy wyczuć trudność w projektowaniu algorytmów, myśląc o następujących (nieformalnie opisanych) problemach algorytmicznych. W pierwszym zadaniu dane wejściowe to legalna pozycja w szachach (czyli opis

sytuacji osiągniętej w pewnym momencie podczas partii szachów), podczas gdy dane wyjściowe to najlepszy ruch dla białych (czyli opis ruchu, który maksymalizuje szanse białych na wygranie meczu). Drugi problem dotyczy dystrybucji gazet. Załóżmy, że 20 000 gazet ma zostać rozprowadzonych do 1000 lokalizacji w 100 miastach przy użyciu 50 ciężarówek. Dane wejściowe zawierają odległości drogowe między miastami i lokalizacjami w każdym mieście, liczbę dokumentów wymaganych w każdej lokalizacji, obecną lokalizację każdej ciężarówki, zdolność każdej ciężarówki do przewozu gazet, a także pojemność benzyny i przebieg -wydajność galonów i szczegóły dotyczące dostępnych kierowców, w tym ich aktualne miejsce pobytu. Wynikiem ma być lista, dopasowująca kierowców do ciężarówek i zawierająca szczegółowe trasy dla każdej z ciężarówek, aby zminimalizować całkowitą liczbę przejechanych mil. W rzeczywistości problem wymaga algorytmu, który działa dla dowolnej liczby gazet, lokalizacji, miast i ciężarówek, tak aby ich liczba również się różniła i stanowiła część danych wejściowych. Zanim będziemy mogli omówić kwestie poprawności i skuteczności, czy też głębsze pytania dotyczące natury lub samego istnienia rozwiązań pewnych problemów algorytmicznych, musimy dowiedzieć się więcej o budowie algorytmów i strukturze obiektów, którymi manipulują.

Algorytmy i dane

Wiemy już, że algorytmy zawierają starannie dobrane instrukcje elementarne, które określają podstawowe czynności do wykonania. Nie omawialiśmy jeszcze rozmieszczenia tych instrukcji w algorytmie, który umożliwia człowiekowi lub komputerowi ustalenie dokładnej kolejności czynności, które należy wykonać. Nie omawialiśmy też przedmiotów, którymi manipulują te działania. Algorytm może być traktowany jako wykonywany przez małego robota lub procesor (który może być odpowiednio nazwany Runaround). Procesor otrzymuje rozkazy biegania dookoła robiąc to i tamto, gdzie „to i tamto" są podstawowymi działaniami algorytmu. W algorytmie sumowania wynagrodzeń z poprzedniej części, małemu Runaroundowi kazano zanotować 0, a następnie zacząć przeglądać listę pracowników, znajdować pensje i dodawać je do zanotowanej liczby. Powinno być dość oczywiste, że kolejność wykonywania podstawowych czynności jest kluczowa. Niezwykle ważne jest nie tylko to, aby podstawowe instrukcje algorytmu były jasne i jednoznaczne, ale to samo powinno dotyczyć mechanizmu sterującego kolejnością wykonywania tych instrukcji. Algorytm musi zatem zawierać instrukcje sterujące, aby „pchać" procesor w tym lub innym kierunku, mówiąc mu, co ma robić na każdym kroku i kiedy przestać i powiedzieć „jestem jednym".

Struktury kontrolne

Sterowanie sekwencją jest zwykle realizowane za pomocą różnych kombinacji instrukcji zwanych strukturami przepływu sterowania lub po prostu strukturami sterowania. Nawet przepis na mus czekoladowy zawiera kilka typowych, takich jak:

* Sekwencjonowanie bezpośrednie, w formie „zrób A, a potem B" lub „zrób A, a potem B". (Każdy średnik lub kropka w przepisie kryje w sobie frazę „a potem", na przykład „delikatnie złóż czekoladę; [a następnie] lekko podgrzej…")

* Rozgałęzienie warunkowe w postaci „jeśli Q, to zrób A, w przeciwnym razie wykonaj B" lub po prostu „jeśli Q, to zrób A", gdzie Q jest pewnym warunkiem. (Na przykład w przepisie „w razie potrzeby lekko podgrzej, aby rozpuścić czekoladę" lub „w razie potrzeby podawaj z bitą śmietaną").

Tak się składa, że te dwie konstrukcje sterujące, sekwencjonowanie i rozgałęzianie, nie wyjaśniają, w jaki sposób algorytm o stałej - może nawet krótkiej - długości może opisywać procesy, które mogą rosnąć coraz dłużej, w zależności od konkretnego wejścia. Algorytm zawierający tylko sekwencjonowanie i rozgałęzianie może zalecić procesy tylko o pewnej ograniczonej długości, ponieważ żadna część takiego algorytmu nie jest nigdy wykonywana więcej niż raz. Konstrukty sterujące, które są odpowiedzialne za przepisywanie coraz dłuższych procesów, są rzeczywiście ukryte nawet w przepisie na mus, ale są znacznie bardziej wyraźne w algorytmach, które radzą sobie z wieloma danymi wejściowymi o różnych rozmiarach, takimi jak algorytm sumowania wynagrodzeń. Są one ogólnie nazywane iteracjami lub konstrukcjami zapętlonymi i występują w wielu odmianach. Oto dwa:

* Ograniczona iteracja w formie ogólnej „zrób A dokładnie N razy", gdzie N jest liczbą.

* Iteracja warunkowa, czasami nazywana iteracją nieograniczoną, w postaci „powtórz A do Q" lub „gdy Q do A", gdzie Q jest warunkiem. (Na przykład w przepisie „ubijaj białka do uzyskania piany").

Opisując algorytm sumowania wynagrodzeń dość nieprecyzyjnie podchodziliśmy do sposobu realizacji głównej części algorytmu; powiedzieliśmy „przejdź przez listę, dodając pensję każdego pracownika do zanotowanej liczby", a następnie „po osiągnięciu końca listy wygeneruj odnotowany numer jako wynik". Powinniśmy naprawdę użyć konstrukcji iteracyjnej, która nie tylko precyzuje zadanie procesora przechodzącego przez listę, ale także sygnalizuje koniec listy. Załóżmy zatem, że dane wejściowe do

problemu obejmują nie tylko listę pracowników, ale także jej długość; to jest całkowita liczba pracowników, oznaczona literą N. Teraz można użyć ograniczonej konstrukcji iteracyjnej, dając następujący algorytm:

(1) zanotuj 0; aby wskazać pierwszą pensję na liście;

(2) wykonaj następujące N - 1 razy:

(2.1) dodać wskazaną pensję do zaznaczonego numeru;

(2.2) wskazać następną pensję;

(3) dodać wskazaną pensję do zaznaczonego numeru;

(4) wygeneruj odnotowaną liczbę jako dane wyjściowe.

Wyrażenie „następujące” w punkcie (2) odnosi się do segmentu składającego się z podrozdziałów (2.1) i (2.2). Ta konwencja, w połączeniu z wcięciem tekstu, aby podkreślić „zagnieżdżony” charakter (2.1) i (2.2), będzie swobodnie używana w sequelu. Zachęcamy do szukania powodu używania N - 1 i dodawania końcowej pensji oddzielnie, zamiast po prostu używać N, a następnie tworzyć wyniki i zatrzymywać się. Zauważ, że algorytm zawodzi, jeśli lista jest pusta (to znaczy, jeśli N wynosi 0), ponieważ druga część klauzuli (1) nie ma sensu. Jeśli dane wejściowe nie zawierają N, całkowitej liczby pracowników, musimy użyć warunkowej iteracji, która wymaga od nas podania sposobu, w jaki algorytm może wykryć, kiedy dotarł do końca listy. Wynikowy algorytm wyglądałby bardzo podobnie do podanej wersji, ale używałby formy „powtórz następujące do osiągnięcia końca listy” w klauzuli (2). Powinieneś spróbować spisać pełny algorytm dla tego przypadku. Zwróć uwagę, jak konstrukcje iteracyjne umożliwiają krótkiej części tekstu algorytmu przypisanie bardzo długich procesów, których długość jest podyktowana wielkością danych wejściowych — w tym przypadku długością listy pracowników. Iteracja jest zatem kluczem do pozornego paradoksu jednego, ustalonego algorytmu wykonującego zadania o coraz dłuższym czasie trwania.

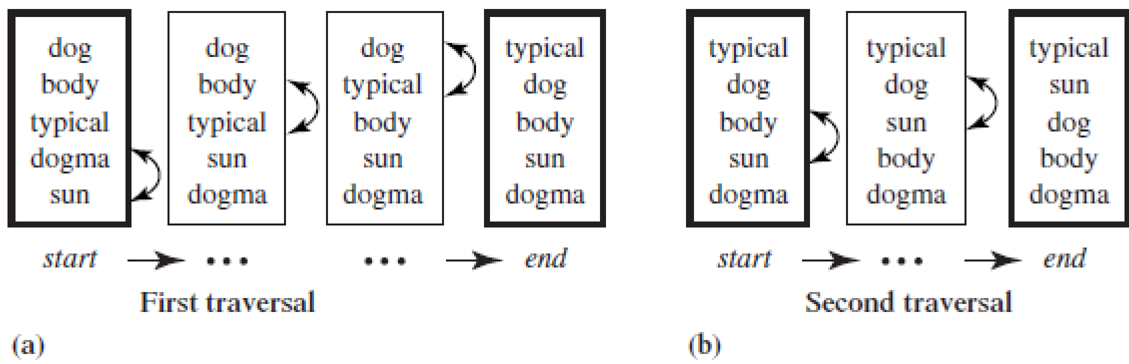
Łączenie struktur kontrolnych

Algorytm może zawierać wiele konstrukcji przepływu sterowania w nietrywialnych kombinacjach. Sekwencjonowanie, rozgałęzianie i iteracje mogą być przeplatane i zagnieżdżane w sobie. Na przykład algorytmy mogą zawierać zagnieżdżone iteracje, częściej nazywane zagnieżdżonymi pętlami. Pętla wewnątrz pętli może przybrać formę „zrób A dokładnie N razy”, gdzie samo A jest, powiedzmy, postacią „powtórz B do C”. Procesor realizujący taki segment musi dość ciężko pracować; za każdym razem, gdy wykonuje A, za każdym razem, gdy przechodzi pętla zewnętrzna, pętla wewnętrzna musi być przemierzana wielokrotnie, aż C stanie się prawdziwe. Tutaj zewnętrzna pętla jest ograniczona, a wewnętrzna warunkowa, ale możliwe są również inne kombinacje. Część A zewnętrznej pętli może zawierać wiele dalszych segmentów, z których każdy może z kolei wykorzystywać dodatkowe konstrukcje sekwencjonowania, rozgałęziania i iteracji, to samo dotyczy pętli wewnętrznej. Tym samym nie ma ograniczeń co do potencjalnej złożoności algorytmów. Rozważmy prosty przykład potęgi iteracji zagnieżdżonych. Załóżmy, że problemem było zsumowanie wynagrodzeń, ale nie wszystkich pracowników, tylko tych, którzy zarabiają więcej niż ich bezpośredni przełożeni. Oczywiście zakłada się, że (poza prawdziwym „szefem”) w kartotece pracownika znajduje się nazwisko przełożonego tego pracownika. Algorytm rozwiązujący ten problem może być skonstruowany tak, aby pętla zewnętrzna przebiegała w dół listy, jak poprzednio, ale dla każdego pracownika „wskazanego” pętla wewnętrzna przeszukuje listę w celu znalezienia rekordu bezpośredniego przełożonego tego pracownika. Po znalezieniu menedżera stosuje się konstrukcję warunkową, aby określić, czy wynagrodzenie pracownika powinno być gromadzone w „zanotowanej liczbie”, co wymaga porównania dwóch

wynagrodzeń. Po wykonaniu tej „wewnętrznej” czynności pętla zewnętrzna wznawia kontrolę i przechodzi do następnego pracownika, którego menedżer jest następnie poszukiwany, aż do osiągnięcia końca listy.

Sortowanie bąbelkowe: Przykład

Aby dokładniej zilustrować struktury kontrolne, przyjrzyjmy się algorytmowi sortowania. Sortowanie to jeden z najciekawszych tematów w algorytmice, z którym wiąże się w ten czy inny sposób wiele ważnych zmian. Dane wejściowe do problemu sortowania to nieuporządkowana lista elementów, powiedzmy liczb. Naszym zadaniem jest stworzenie listy posortowanej w porządku rosnącym. Problem można sformułować bardziej ogólnie, zastępując, powiedzmy, listami słów listy liczbowe, z zamiarem ich posortowania według ich leksykograficznej kolejności (tj. jak w słowniku lub książce telefonicznej). Zakłada się, że lista elementów poprzedzona jest jej długością N, a jedynym sposobem na uzyskanie informacji o wielkości tych elementów polega na wykonaniu porównań binarnych; to znaczy porównywać dwa elementy i działać zgodnie z wynikiem porównania. Jeden z wielu znanych algorytmów sortowania nazywa się sortowaniem bąbelkowym. W rzeczywistości sortowanie bąbelkowe jest uważane za zły algorytm sortowania z powodów wyjaśnionych w Części 6. Jest on tutaj używany tylko do zilustrowania struktur kontrolnych. Algorytm sortowania bąbelkowego opiera się na następującej obserwacji. Jeśli pomieszana lista jest przemierzana po kolei, po jednym elemencie na raz i gdy dwa sąsiednie elementy są w złej kolejności (to znaczy, że pierwszy jest większy niż drugi), są one wymieniane, to po zakończeniu przechodzenia , największy element jest na swoim miejscu; mianowicie na końcu listy. Rysunek (a) ilustruje takie przechodzenie dla prostej pięcioelementowej listy.



(Lista została sporządzona od dołu do góry: pierwszy element jest najniższy na rysunku. Strzałki pokazują tylko wymienione elementy, a nie te, które są porównywane.) Oczywiście, sortowanie może poprawić inne nieprawidłowe kolejności poza umieszczeniem maksymalnego elementu w jego ostateczna pozycja. Jednak rysunek (a) pokazuje, że jedno przejście niekoniecznie sortuje listę. Teraz drugie przejście doprowadzi drugi co do wielkości element do właściwego punktu spoczynku, przedostatniej pozycji na liście, jak widać na rysunku (b). Prowadzi to do algorytmu, który wykonuje N-1 takich przejść (dlaczego nie N?), co daje posortowaną listę. Nazwa „bubblesort” pochodzi od sposobu, w jaki duże elementy „wyskakują” na górę listy w miarę postępu algorytmu, zamieniając miejsca na mniejsze elementy, które są przesuwane niżej. Przed bardziej szczegółowym opisem algorytmu należy zauważyć, że drugie przechodzenie nie musi sięgać do ostatniego elementu, ponieważ w momencie rozpoczęcia drugiego przemierzania ostatnia pozycja na liście zawiera już prawowitego dzierżawcę - największy element na liście. Podobnie, trzecie przejście nie musi iść dalej niż pierwsze N-2 elementy. Oznacza to, że bardziej wydajny algorytm przemierzyłby tylko pierwsze N elementów w pierwszym przejściu, pierwsze N-1 w drugim, N-2 w trzecim itd. Powrócimy do algorytmu

sortowania pęcherzyków i to ulepszenie Część 6, ale na razie wystarczy wersja nieulepszona. Algorytm brzmi następująco:

(1) wykonaj następujące N - 1 razy:

(1.1) wskazać na pierwszy element;

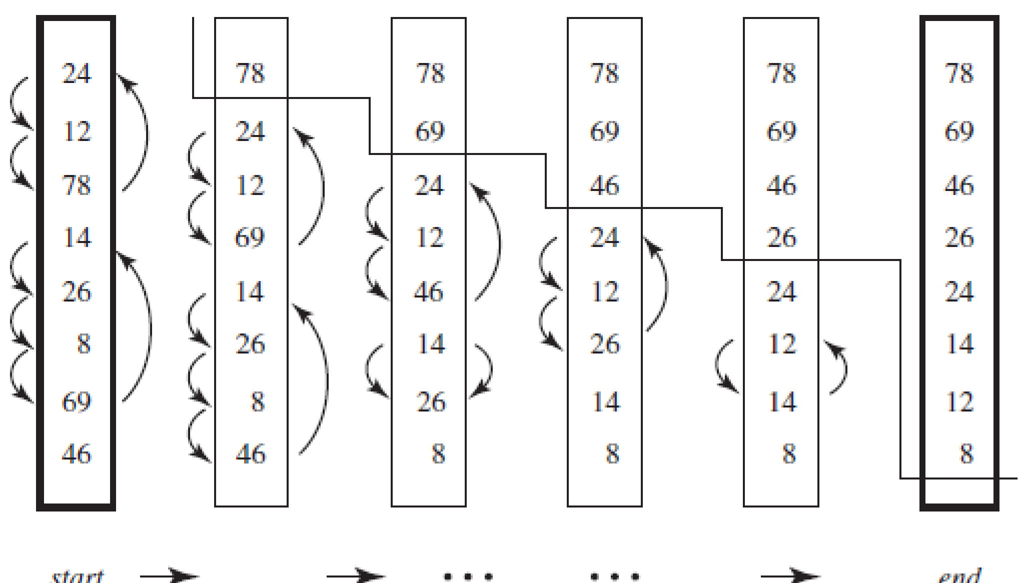
(1.2) wykonaj następujące N - 1 razy:

(1.2.1) porównać wskazany element z następnym elementem;

(1.2.2) jeśli porównywane elementy są w złej kolejności, wymienić je;

(1.2.3) wskazuje na następny element.

Zwróć uwagę, jak jest tutaj używane wcięcie dwupoziomowe. Pierwsza „następna”, w linii (1), obejmuje wszystkie linie zaczynające się od 1, a druga, w linii (1.2), obejmuje te zaczynające się od 1.2. W ten sposób wyraźnie widać zagnieżdżony charakter konstrukcji pętli. Główne kroki podejmowane przez algorytm na ośmiopunktowej liście są zilustrowane na rysunku 2.2, gdzie sytuacja jest przedstawiona tuż przed każdym wykonaniem punktu (1.2).



Elementy pojawiające się nad linią znajdują się w swoich ostatecznych pozycjach. Zauważ, że w tym konkretnym przykładzie dwa ostatnie przejścia (nie pokazane) są zbędne; lista jest sortowana po pięciu, a nie siedmiu przejściach. Należy jednak zauważyć, że jeśli na przykład najmniejszy element jest ostatnim na oryginalnej liście (czyli na górze na naszych ilustracjach), to w rzeczywistości konieczne są wszystkie przejścia N - 1, ponieważ elementy, które mają być „ bulgotać” (elbbubed? …) sprawiają więcej kłopotów niż te, które „bulgotają”.

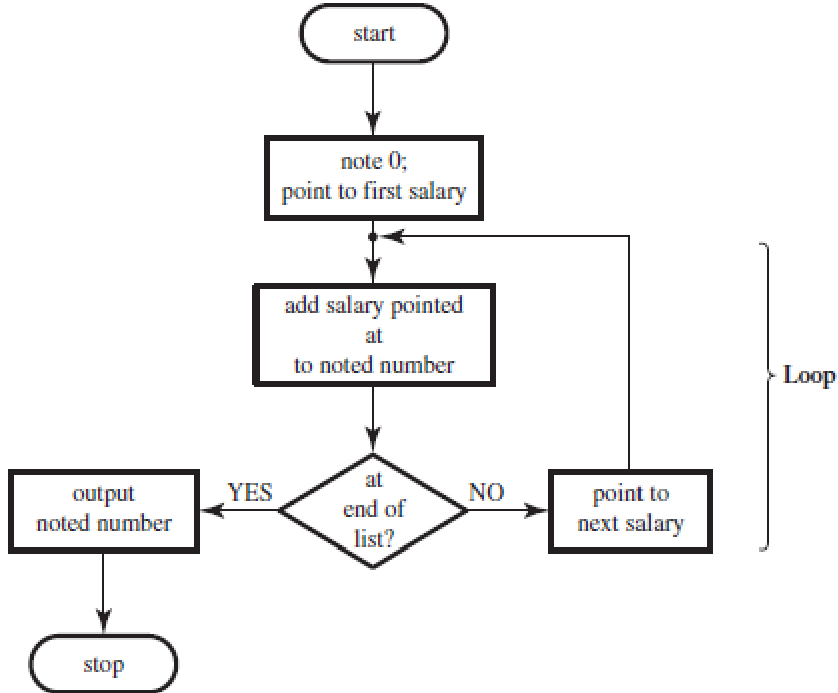
Instrukcja „Goto”

Jest jeszcze inna ważna instrukcja kontrolna, która jest ogólnie nazywana instrukcją goto. Ma ogólną postać „goto G”, gdzie G zaznacza jakiś punkt w tekście algorytmu. W naszych przykładach moglibyśmy napisać, powiedzmy, „goto (1.2)”, instrukcję, która powoduje, że procesor Runaround dosłownie przechodzi do wiersza (1.2) algorytmu i wznawia wykonywanie od tego miejsca. Konstrukcja ta jest

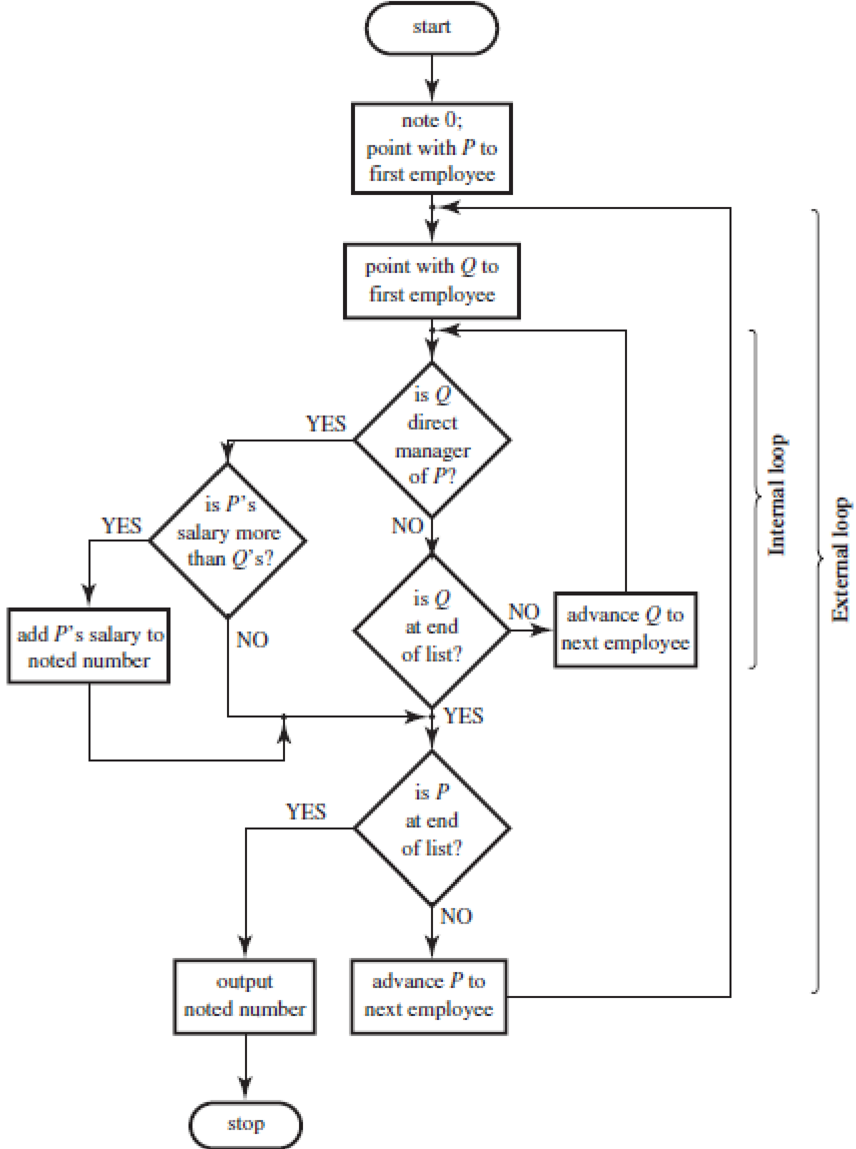
kontrowersyjna z wielu powodów, z których najbardziej oczywisty ma charakter pragmatyczny. Algorytm, który zawiera wiele instrukcji „goto" kierujących sterowanie w tę i z powrotem w splątany sposób, szybko staje się bardzo trudny do zrozumienia. Przejrzystość, jak będziemy argumentować później, jest bardzo ważnym czynnikiem w projektowaniu algorytmicznym. Poza potencjalnym ograniczeniem naszej zdolności do zrozumienia algorytmu, istrukcje „goto" mogą również powodować trudności techniczne. Co się stanie, jeśli instrukcja „goto" skieruje procesor w sam środek pętli? Przykładem tego jest wstawienie instrukcji „goto (1.2.1)" między (1.1) a (1.2) do algorytmu sortowania bąbelkowego. Czy procesor ma wykonać (1.2.1) do (1.2.3), a następnie zatrzymać się, czy też należy wykonać je N-1 razy? Co się stanie, jeśli ta sama instrukcja pojawi się w sekwencji (1.2.1)-(1.2.3)? Problem ten ma swoje źródło w niejednoznaczności wynikającej z próby dopasowania tekstu algorytmu do zaleconego przez niego procesu. Oczywiście istnieje takie dopasowanie, ale ponieważ ustalone algorytmy mogą określać procesy o różnej długości, pojedynczy punkt w tekście algorytmu może być powiązany z wieloma punktami wykonywania odpowiedniego procesu. W związku z tym istrukcje „goto" są w pewnym sensie istotami z natury niejednoznacznymi, a wielu badaczy sprzeciwia się używaniu go swobodnie w algorytmach.

Diagramy dla algorytmów

Wizualne, diagramatyczne techniki są jednym ze sposobów przedstawiania przepływu sterowania algorytmem w jasny i czytelny sposób. Istnieją różne sposoby „rysowania" algorytmów, w przeciwieństwie do ich zapisywania. Jednym z najbardziej znanych jest zapisywanie podstawowych instrukcji w prostokątnych pudełkach, a testów w rombowych pudełkach i używanie strzałek do opisania, w jaki sposób procesor Runaround wykonuje algorytm. Powstałe obiekty nazywane są schematami blokowymi. Rysunek 1 przedstawia schemat blokowy zwykłego algorytmu sumowania wynagrodzeń,



a Rysunek 2 przedstawia schemat bardziej wyrafinowanej wersji, która obejmuje tylko pracowników zarabiających więcej niż ich bezpośredni przełożeni.



Zwróć uwagę na sposób, w jaki iteracja przedstawia się wizualnie jako cykl prostokątów, rombów i strzałek, a zagnieżdżone iteracje są wyświetlane jako cykle w cyklach. To wyjaśnia użycie terminu „pętla". Schematy blokowe na rysunkach 2 i 3 ilustrują również stosowność terminu „rozgałęzienie", który był powiązany z instrukcjami warunkowymi. Schematy blokowe mają również wady. Jednym z nich jest fakt, że trudniej jest zachęcić ludzi do przestrzegania niewielkiej liczby „dobrze uformowanych" struktur kontrolnych. Korzystając ze schematów blokowych, łatwo jest ulec pokusie użycia wielu „goto", ponieważ są one przedstawiane po prostu jako strzałki, takie jak te, które reprezentują pętle „while" lub instrukcje warunkowe. Tak więc to nadużywanie medium schematów blokowych spowodowało, że wielu badaczy zaleciło ich stosowanie z ostrożnością. Innym problemem jest fakt, że wiele rodzajów algorytmów po prostu nie nadaje się w naturalny sposób do graficznego, diagramowego odwzorowania oferowanego przez takie jak schematy blokowe. Powstałe artefakty będą często przypominać spaghetti, zmniejszając, a nie zwiększając, zdolność widza do zrozumienia, co naprawdę się dzieje. Niezależnie od powyższej dyskusji, w Części 14 zobaczymy, że istnieją języki diagramowe (nazwiemy je formalizmami wizualnymi), które są bardzo skuteczne, głównie w kontekście określania zachowania dużych i złożonych systemów. Problemem nie jest opisywanie obliczeń, jak robią to algorytmy, ale określenie reaktywnego i interaktywnego zachowania w czasie.

Podprogramy lub procedury

Załóżmy, że otrzymujemy obszerny tekst i chcemy dowiedzieć się, jak chciwy jest jego autor, licząc zdania, które zawierają słowo „pieniądze”. W takich przypadkach nie interesuje nas liczba wystąpień słowa „pieniądze”, ale liczba zdań, w których występuje. Można zaprojektować algorytm, który będzie przechodził przez tekst w poszukiwaniu „pieniędzy”. Po znalezieniu takiego zdarzenia biegnie naprzód, szukając końca zdania, które dla naszych celów przyjmuje się jako kropkę, po której następuje spacja; to jest ". ” kombinacja. Po znalezieniu końca zdania algorytm dodaje 1 do licznika (czyli „zanotowanej liczby”, jak w algorytmie sumowania wynagrodzeń), który został zainicjowany na 0 na początku. Następnie wznawia poszukiwanie „pieniędzy” od początku następnego zdania; to znaczy od litery następującej po kombinacji. Oczywiście algorytm musi cały czas zwracać uwagę na koniec tekstu, aby po jego osiągnięciu mógł wypisać wartość licznika. Algorytm przyjmuje postać zewnętrznej pętli, której zadaniem jest liczenie odpowiednich zdań. W tej pętli znajdują się dwa wyszukiwania, jedno dla „pieniądze”, a drugie dla „. ”, z których każda sama w sobie stanowi pętlę. Chodzi o to, że dwie wewnętrzne pętle są bardzo podobne; w rzeczywistości obaj robią dokładnie to samo – szukają sekwencji symboli w tekście. Obecność obu pętli w algorytmie wyraźnie działa, ale możemy zrobić to lepiej. Pomysł polega na napisaniu pętli wyszukiwania tylko raz, z parametrem, który z założenia zawiera konkretną kombinację poszukiwanych symboli. Ten segment algorytmiczny nazywa się podprogramem lub procedurą i jest aktywowany (lub wywoływany lub wywoływany) dwukrotnie w głównym algorytmie, raz z parametrem „pieniądze”, a raz z „. ” kombinacja. Tekst podprogramu jest dostarczany osobno i odnosi się on do zmiennego parametru poprzez nazwę, powiedzmy X. Podprogram zakłada, że wskazujemy jakieś miejsce w tekście wejściowym i może wyglądać następująco: podprogram szukaj X :

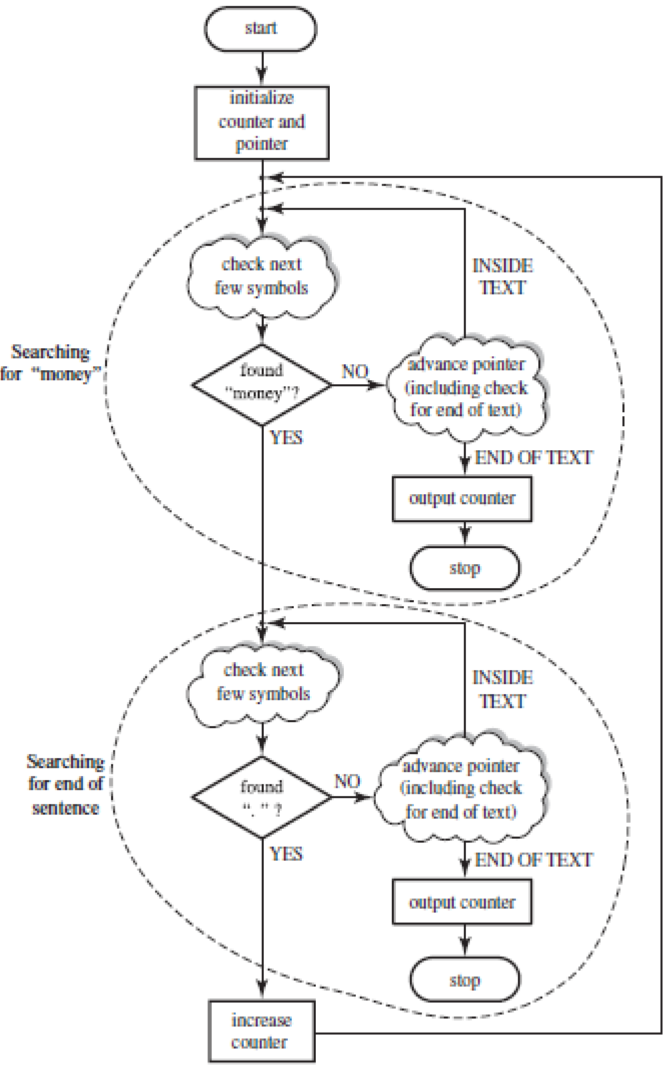
(1) wykonaj następujące czynności, aż wskażesz kombinację X lub dotrzesz do końca tekstu:

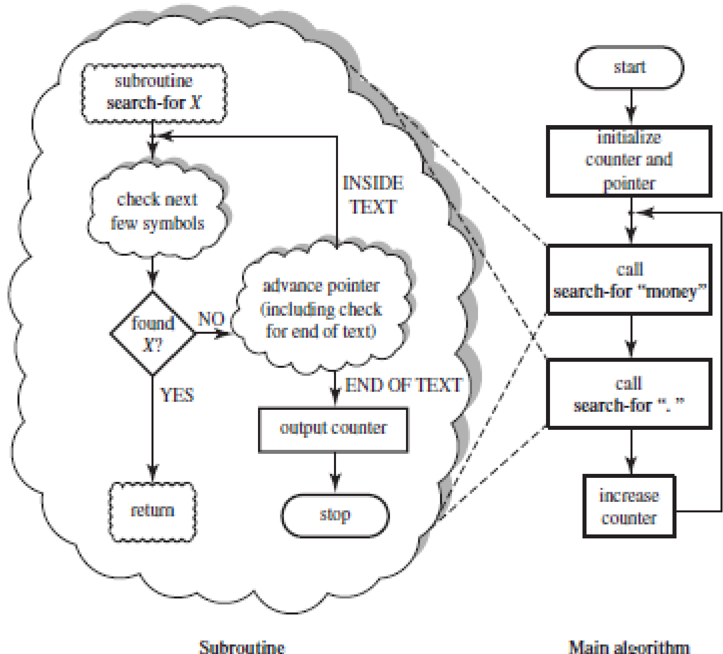
(1.1) przesunąć wskaźnik o jeden symbol w tekście;

(2) po osiągnięciu końca tekstu wyprowadź wartość licznika i zatrzymaj się;

(3) w przeciwnym razie wróć do głównego algorytmu.

Główna część algorytmu użyje podprogramu wyszukiwania dwukrotnie, za pomocą instrukcji w postaci „wywołaj wyszukiwanie pieniędzy” i „wywołaj wyszukiwanie”. ’ ” Porównaj rysunek 1 z rysunkiem 2, na którym schematycznie pokazano wersję z podprogramem.


start
initialize counter and pointer
check next few symbols
found "money"?
NO
YES
advance pointer (including check for end of text)
INSIDE TEXT
END OF TEXT
output counter
stop
Searching for "money"
check next few symbols
found "." ?
NO
YES
advance pointer (including check for end of text)
INSIDE TEXT
END OF TEXT
output counter
stop
Searching for end of sentence
increase counter



Procesor, który działa, będzie teraz musiał być nieco bardziej wyrafinowany. Gdy zostaniesz poproszony o „wywołanie" podprogramu, zatrzyma to, co robił, zapamiętaj, gdzie to było, wybierz parametr(y), że tak powiem, i przejdzie do tekstu podprogramu. Następnie zrobi to, co każe mu podprogram, używając bieżącej wartości parametru, gdziekolwiek podprogram odwołuje się do tego parametru za pomocą swojej wewnętrznej nazwy (w naszym przykładzie X). Jeśli i kiedy podprogram każe mu powrócić, to właśnie to zrobi; mianowicie powróci do punktu następującego po „wezwaniu", które doprowadziło go w pierwszej kolejności do podprogramu, i stamtąd wznowi swoje obowiązki.

Cnoty podprogramów

Oczywiście podprogramy mogą być bardzo ekonomiczne, jeśli chodzi o rozmiar algorytmu. Nawet w tym prostym przykładzie pętla wyszukiwania jest napisana raz, ale jest używana dwa razy, podczas gdy bez niej algorytm musiałby oczywiście uwzględnić dwie szczegółowe wersje tej pętli. Ta ekonomia staje się znacznie ważniejsza dla złożonych algorytmów z wieloma podprogramami, które są wywoływane z różnych miejsc. Ponadto algorytm może zawierać podprogramy, które wywołują inne podprogramy i tak dalej. W ten sposób struktura algorytmiczna otrzymuje nowy wymiar; istnieją nie tylko zagnieżdżone pętle, ale zagnieżdżone podprogramy. Co więcej, pętle, instrukcje warunkowe, konstrukcje sekwencyjne, instrukcje „goto", a teraz podprogramy, mogą być przeplatane, dając algorytmy o rosnącej złożoności strukturalnej. Ekonomia to jednak nie jedyna zaleta podprogramów. Podprogram może być postrzegany jako „kawał" materiału algorytmicznego, pewnego rodzaju cegiełka, która po utworzeniu może być wykorzystana w innym fragmencie algorytmicznym przez pojedynczą instrukcję. To tak, jakby powiedzieć, że rozszerzyliśmy nasz repertuar dozwolonych instrukcji elementarnych. W przykładzie liczenia „pieniędzy", gdy procedura wyszukiwania jest już dostępna (a nawet wcześniej, o ile zdecydowano, że taka procedura zostanie ostatecznie napisana), instrukcja „call search-for 'abc' " jest dla każdego praktyczny cel, nowa podstawowa instrukcja. Tak więc podprogramy są jednym ze sposobów, w jaki możemy tworzyć własne abstrakcje, zgodnie z

konkretnym problemem, który próbujemy rozwiązać. To bardzo potężny pomysł, ponieważ nie tylko skraca algorytmy, ale także czyni je przejrzystymi i dobrze zorganizowanymi. Przejrzystość i struktura, jak wielokrotnie podkreślano, mają ogromne znaczenie w algorytmice i wiele wysiłku poświęca się znalezieniu sposobów narzucenia ich projektantom algorytmów.

W ten sam sposób, w jaki użytkownik programu komputerowego zwykle nie wie nic o algorytmach, których używa, podprogram może być używany jako „czarna skrzynka", nie wiedząc, jak jest zaimplementowany. Wszystko, co użytkownik podprogramu musi wiedzieć, to co robi, ale nie jak to robi. To znacznie upraszcza problem, zmniejszając ilość szczegółów, o których należy pamiętać. Za pomocą podprogramów można stopniowo, krok po kroku, rozwijać złożony algorytm. Typowy problem algorytmiczny wymaga w pełni szczegółowego rozwiązania, które wykorzystuje tylko dozwolone czynności elementarne. Projektant może stopniowo dążyć do tego celu, najpierw opracowując algorytm wysokiego poziomu, który wykorzystuje „podstawowe" instrukcje, których nie ma w książce. W rzeczywistości są to wywołania podprogramów, które projektant ma na myśli, a które zostaną napisane później (lub być może wcześniej). Te podprogramy z kolei mogą korzystać z innych instrukcji, które, nie będąc wystarczająco elementarnymi, są ponownie uważane za wywołania podprogramów, które zostaną ostatecznie napisane. W pewnym momencie wszystkie podstawowe instrukcje są na wystarczająco niskim poziomie, aby znaleźć się wśród tych wyraźnie dozwolonych. Wtedy kończy się stopniowy proces rozwoju. Takie podejście daje początek albo projektowi „z góry na dół", który, jak już opisano, idzie od ogółu do konkretu, albo konstrukcji „od dołu do góry", w której przygotowuje się potrzebne podprogramy, a następnie projektuje bardziej ogólne procedury, które je nazywają, pracując w ten sposób od szczegółowych do ogólnych. Nie ma powszechnej zgody co do tego, które z nich jest lepszym podejściem do projektowania algorytmicznego. Powszechnie uważa się, że należy użyć jakiejś mieszanki. Wracając na chwilę do gastronomii, przygotowanie „mieszanki czekoladowej" może być dobrym kandydatem na podprogram w przepisie na mus czekoladowy z Części 1. To pozwoliłoby nam opisać przepis w następującym sposób, w którym każda z czterech instrukcji jest traktowana jako wywołanie podprogramu (lub raczej podprzepisu), którego tekst byłby następnie napisany osobno:

(1) przygotować mieszankę czekoladową;

(2) mieszanka do wytworzenia mieszanki czekoladowo-żółtkowej;

(3) przygotować piankę z białek jaj;

(4) wymieszaj oba, aby uzyskać mus.

Warto zaznaczyć, że książka, z której zaczerpnięto przepis na mus, dość obszernie wykorzystuje podprogramy. Na przykład opisujemy szereg przepisów, których składniki zawierają pozycje takie jak Przepis na Słodkie Ciasto, Rogaliki czy Ciasto Francuskie, dla których użytkownik jest odsyłany do wcześniej podanych przepisów dedykowanych tym właśnie produktom. Można śmiało powiedzieć, że nie można przecenić mocy i elastyczności zapewnianych przez podprogramy

Rekurencja

Jednym z najbardziej użytecznych aspektów podprogramów, który dla wielu ludzi jest również jednym z najbardziej mylących, jest rekurencja. Rozumiemy przez to po prostu zdolność podprogramu lub procedury do wywołania samej siebie. Może to zabrzmieć absurdalnie, skoro jak, u licha, nasz procesor przybliża się do rozwiązania problemu, gdy w trakcie próby rozwiązania tego problemu mówi się, żeby zostawić wszystko i zacząć rozwiązywać ten sam problem od nowa? Poniższy przykład powinien nam pomóc w rozwiązaniu tego paradoksu i może pomóc, jeśli powiemy na początku, że rozwiązanie opiera

się na tej samej właściwości algorytmów, o których mowa wcześniej: ten sam tekst (w tym przypadku podprogram rekurencyjny) może odpowiadać wielu częściom opisanego przez nią procesu. Konstrukcje iteracyjne są jednym ze sposobów mapowania długich procesów na krótkie teksty; podprogramy rekurencyjne to kolejny. Przykład opiera się na dość starożytnej zagadce znanej jako Wieże Hanoi, wywodzącej się od hinduskich kapłanów z wielkiej świątyni Benares. Załóżmy, że otrzymaliśmy trzy wieże, a żeby być bardziej skromnym, trzy kołki, A, B i C. Na pierwszym kołku, A, są trzy pierścienie ułożone w malejącym porządku wielkości, podczas gdy pozostałe są puste. Jesteśmy zainteresowani w przenoszeniu pierścieni z A do B, być może przy użyciu C w procesie. Zgodnie z zasadami gry, pierścienie należy przesuwać pojedynczo i w żadnym momencie nie można umieszczać większego pierścienia na mniejszym. Tę prostą zagadkę można rozwiązać w następujący sposób:

przenieś A do B;

przenieś A do C;

przenieś B do C;

przenieś A do B;

przenieś C do A;

przenieś C do B;

przenieś A do B.

Zanim przejdziemy dalej, powinniśmy najpierw przekonać się, że te siedem akcji naprawdę działa, a następnie powinniśmy wypróbować tę samą łamigłówkę z czterema, a nie trzema pierścieniami na kołku A (liczba kołków się nie zmienia). Umiarkowana ilość pracy powinna wystarczyć, aby odkryć sekwencję 15 akcji „przesuń X na Y", które rozwiązują wersję czteropierścieniową. Takie łamigłówki są intelektualnie trudne, a ci, którzy lubią łamigłówki, mogą chcieć rozważyć oryginalną hinduską wersję, która zawiera te same trzy kołki, ABC, ale z nie mniej niż 64 pierścieniami na A. Jak zobaczymy, wynalazcy układanki nie byli całkowicie oderwali się od rzeczywistości, gdy stwierdzili, że świat skończy się, gdy wszystkie 64 pierścienie zostaną prawidłowo ułożone na kołku B. Jednak nie mamy tutaj do czynienia z zagadkami, ale z algorytmiką, a w konsekwencji bardziej interesuje nas ogólny problem algorytmiczny związany z Wieżami Hanoi niż z tym czy innym konkretnym przypadkiem. Dane wejściowe to dodatnia liczba całkowita N, a pożądanym wynikiem jest lista działań „przenieś X do Y", które, jeśli zostaną wykonane, rozwiązują zagadkę obejmującą N pierścieni. Oczywiście rozwiązaniem tego problemu musi być algorytm, który działa dla każdego N, tworząc listę działań spełniających ograniczenia; w szczególności podążanie za nimi nigdy nie powoduje umieszczenia większego pierścienia na mniejszym. To jest problem, który naprawdę powinniśmy próbować rozwiązać, ponieważ gdy algorytm jest już dostępny, każdą instancję łamigłówki, niezależnie od tego, czy jest to wersja trzy-, cztero- czy 3078-pierścieniowa, można rozwiązać po prostu uruchamiając algorytm z żądaną liczbę dzwonków jako dane wejściowe. Jak to się robi? Odpowiedź jest prosta: dzięki magii rekurencji.

Rozwiązanie dla wież Hanoi

Przedstawiony tutaj algorytm realizuje zadanie przeniesienia N pierścieni z A do B przez C w następujący sposób. Najpierw sprawdza, czy N wynosi 1, w którym to przypadku po prostu przesuwa jeden pierścień, o który został poproszony, do miejsca docelowego (a dokładniej, wyświetla opis jednego ruchu, który wykona zadanie), i następnie wraca natychmiast. Jeśli N jest większe niż 1, najpierw przesuwa górne N-1 pierścieni z A do „dodatkowego" kołka C przy użyciu tej samej procedury

rekurencyjnej; następnie podnosi jedyny pozostały pierścień w A, który musi być największym pierścieniem (dlaczego?) i przenosi go do miejsca docelowego, B; następnie, ponownie rekursywnie, przesuwa pierścienie N-1, które wcześniej „przechowywał” w C do ich ostatecznego miejsca przeznaczenia, B. Że ten algorytm przestrzega reguł gry jest trochę trudny do zauważenia, ale przy założeniu, że dwa procesy związane z pierścieniami N-1 zawierają tylko legalne ruchy, dość łatwo zauważyć, że tak samo jest z całym procesem przemieszczania pierścieni N. Oto algorytm. Jest napisany jako procedura rekurencyjna, której tytuł i parametry (jest ich cztery) mówią same za siebie:

podprogram przesuń N z X na Y za pomocą Z:

(1) jeśli N wynosi 1, to wypisz „przenieś X do Y”;

(2) w przeciwnym razie (tj. jeśli N jest większe niż 1) wykonaj następujące czynności:

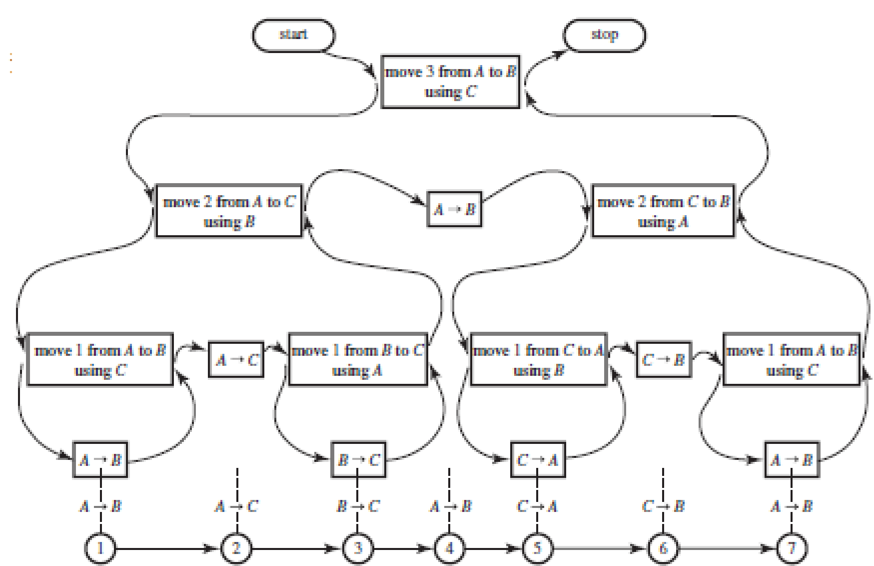
(2.1) wywołaj przeniesienie N − 1 z X do Z używając Y ;

(2.2) wyjście „przesuń X do Y”;

(2.3) wywołaj przeniesienie N − 1 z Z do Y za pomocą X;

(3) powrót.

Aby zilustrować działanie tej procedury, która na pierwszy rzut oka może wyglądać dość śmiesznie, moglibyśmy spróbować uruchomić ją, gdy N wynosi 3; czyli symulowanie pracy procesora, gdy są trzy pierścienie. Należy to zrobić wykonując „główny algorytm” składający się z pojedynczej instrukcji: wywołaj ruch 3 z A do B za pomocą C. Symulację należy przeprowadzić ostrożnie, ponieważ nazwy parametrów X, Y i Z zaczynają się dość niewinnie, jako są A, B i C, ale zmieniają się za każdym razem, gdy procesor ponownie wejdzie w procedurę. Jakby, aby było bardziej zagmatwane, wznawiają swoje stare wartości, gdy „rozmowa” się kończy, a procesor wraca do miejsca, z którego pochodzi. Rysunek



pomaga zilustrować kolejność działań w tym przypadku. Zauważ, że procesor (i my też, jeśli rzeczywiście próbujemy symulować procedurę) musi teraz pamiętać nie tylko skąd pochodzi, aby powrócić do właściwego miejsca, ale także jakie wartości nazw parametrów były przed swoje nowe zadanie, aby prawidłowo wznowić pracę. Rysunek jest zorganizowany tak, aby odzwierciedlić

głębokość rekurencji i odpowiedni sposób zmiany wartości parametrów. Strzałki reprezentują podróże procesora: strzałki w dół odpowiadają wywołaniom, strzałki w górę do powrotów, a strzałki poziome do prostego sekwencjonowania. Jak za dotknięciem czarodziejskiej różdżki okazuje się, że ostateczna sekwencja działań jest identyczna z tą, do której doszło wcześniej metodą prób i błędów. Jak się okazuje, problem Wież Hanoi dopuszcza inne rozwiązanie algorytmiczne, które wykorzystuje prostą iterację i w ogóle nie wywołuje wywołań podprogramów, i może być wykonane przez małe dziecko!

Ekspresyjna moc struktur kontrolnych

Czy możemy zrobić tylko z kilkoma prostymi strukturami kontrolnymi? Odpowiedź brzmi tak. Zidentyfikowano różne minimalne zestawy struktur kontrolnych, co oznacza, że w pewnych technicznych aspektach inne struktury kontrolne można zastąpić odpowiednimi kombinacjami tych ze zbioru minimalnego, tak że w praktyce są one jedyne potrzebne. Dobrze znany zestaw minimalny składa się z sekwencjonowania (a następnie), rozgałęzienia warunkowego (jeśli-to) i pewnego rodzaju konstrukcji pętli nieograniczonej (na przykład while-do). Nietrudno np. pokazać, że instrukcję w postaci „powtórz A, aż Q będzie prawdziwe”, można zastąpić „zrób A, a potem, gdy Q jest fałszywe, wykonaj A”, aby w obecności konstrukcje „podczas działania”, które można wykonać bez konstrukcji „powtarzaj aż do”. W podobnym duchu możliwe jest całkowite wyeliminowanie stwierdzeń „goto” z dowolnego algorytmu, kosztem jednak pewnego rozszerzenia algorytmu i zmiany jego pierwotnej struktury. Podobnie jest w ścisłym sensie, w którym wszystko, co można zrobić za pomocą podprogramów i rekurencji, można zrobić tylko za pomocą prostych pętli. Wykorzystanie tego wyniku do pozbycia się podprogramów danego algorytmu wiąże się jednak z dodaniem do algorytmu znacznej „maszyny” (w postaci nowych instrukcji elementarnych). Można wykazać, że jeśli taka maszyneria nie jest dozwolona, to podprogramy rekurencyjne są potężniejsze niż iteracja, tak że pewne rzeczy można zrobić tylko za pomocą tych pierwszych. Tematy te zostały tu poruszone tylko w sposób najbardziej powierzchowny, aby dać ci wyczucie istotnych kwestii, które cię interesują.

Typy danych i struktury danych

Wiemy więc, jak wygląda algorytm i jak, biorąc pod uwagę dane wejściowe, procesor wykonuje proces, który opisuje algorytm. Jednak byliśmy dość niejasni co do obiektów manipulowanych przez algorytm. Mieliśmy listy, słowa i teksty, a także zabawne rzeczy, takie jak „zanotowane liczby”, które wzrosły i „wskaźniki” które poczyniły postępy. Jeśli chcemy posunąć się do skrajności, możemy również powiedzieć, że mieliśmy mąkę, cukier, ciasta i mus czekoladowy, a także wieże, kołki i krążki. Obiekty te stanowiły nie tylko wejścia i wyjścia algorytmu, ale także akcesoria pośrednie, które zostały skonstruowane i używane w trakcie jego życia, takie jak liczniki („zanotowane liczby”) i wskaźniki. Dla wszystkich z nich używamy ogólnego terminu dane. Elementy danych występują w różnych odmianach lub mogą być różnego rodzaju. Niektóre z najczęstszych typów danych występujących w algorytmach wykonywanych przez komputery to różnego rodzaju liczby (całkowite, dziesiętne, binarne itd.) oraz słowa zapisane w różnych alfabetach. W rzeczywistości liczby mogą być również rozumiane jako słowa; Na przykład liczby całkowite dziesiętne to „słowa” w alfabecie składającym się z cyfr 0, 1, 2, . . . , 9 i liczby binarne używają alfabetu składającego się tylko z 0 i 1. Korzystne jest jednak trzymanie takich typów oddzielnie, nie tylko dla przejrzystości i porządku, ale także dlatego, że każdy typ dopuszcza własny specjalny zestaw dozwolonych operacji lub akcji. Wyliczanie samogłosek w liczbie nie ma większego sensu niż mnożenie dwóch wyrazów! I tak, naszym algorytmom trzeba będzie powiedzieć, jakimi obiektami mogą manipulować i jakie manipulacje są dozwolone. Manipulacja obejmuje nie tylko wykonywanie operacji, ale także zadawanie pytań, np. sprawdzanie, czy liczba jest parzysta lub czy dwa słowa zaczynają się na tę samą literę. Obserwacje te wydają się całkiem naturalne w świetle naszej dyskusji na temat operacji elementarnych w części 1 i emanują faktami dotyczącymi możliwości komputerów do manipulacji bitami. Tutaj przyjmujemy za pewnik podstawowe typy danych oraz

operacje i testy z nimi związane. Interesuje nas to, w jaki sposób algorytmy mogą organizować, zapamiętywać, zmieniać i uzyskiwać dostęp do zbiorów danych. Podczas gdy struktury kontrolne służą do informowania procesora, dokąd powinien się udać, struktury danych i operacje na nich organizują elementy danych w sposób, który umożliwia mu robienie tego, co powinien zrobić, gdy się tam dostanie. Świat struktur danych jest tak samo bogaty w poziomy abstrakcji, jak świat struktur kontrolnych. W rzeczywistości przydatna sztuczka mentalna, która jest podstawą paradygmatu programowania obiektowego, pokazuje, że możemy się między nimi przełączać!

Zmienne, czyli małe pudełka

Pierwszymi obiektami zainteresowania są zmienne. Na przykład w algorytmie sumowania wynagrodzeń użyliśmy „zanotowanej liczby", która została najpierw zainicjowana na 0, a następnie użyta do akumulacji sumy wynagrodzeń pracowników. Właściwie używaliśmy zmiennej. Zmienna nie jest liczbą, słowem ani jakimś innym elementem danych. Może być raczej postrzegana jako małe pudełko lub komórka, w której można przechowywać pojedynczy przedmiot. Możemy nadać zmiennej nazwę, powiedzmy X, a następnie użyć instrukcji typu „wstaw 0 w X" lub „zwiększ zawartość X o 1.". Możemy również zadawać pytania dotyczące zawartości X, na przykład „czy X zawiera liczbę parzystą?" Zmienne są bardzo podobne do pokoi hotelowych; różne osoby mogą zajmować pokój w różnym czasie, a osobę znajdującą się w pokoju można określić frazą „mieszkańca pokoju 326". Termin „326" to nazwa pomieszczenia, podobnie jak X to nazwa zmiennej. To użycie słowa „zmienna" do oznaczenia komórki, która może zawierać różne wartości w różnym czasie, jest niepodobne do znaczenia zmiennej w matematyce, gdzie oznacza pojedynczą (zwykle nieznaną) wartość. W Części 3 omówimy paradygmat programowania funkcyjnego, który nie zajmuje się komórkami, ale bezpośrednio wartościami, jak w matematyce. Algorytmy zazwyczaj wykorzystują wiele zmiennych o różnych nazwach i do bardzo różnych celów. Na przykład w algorytmie sortowania bąbelkowego szczegółowa wersja może używać jednej zmiennej do zliczania, ile razy wykonywana jest pętla zewnętrzna, innej dla pętli wewnętrznej i trzeciej, aby pomóc w wymianie dwóch elementów na liście. Aby dokonać wymiany, jeden element jest na chwilę „wkładany do pudełka", drugi na swoje miejsce, a następnie „pudełkowy" element jest umieszczany na pierwotnym miejscu drugiego elementu. Bez użycia zmiennej wydaje się, że nie ma sposobu na zachowanie pierwszego elementu bez jego utraty. Ilustruje to użycie zmiennych jako pamięci lub przechowywania w algorytmie. Oczywiście fakt, że elementy są wymieniane wielokrotnie w jednym przebiegu sortowania bąbelkowego, nie oznacza, że potrzebujemy wielu zmiennych - za każdym razem można użyć tego samego „pudełka". Wynika to z faktu, że wszystkie wymiany w algorytmie sortowania pęcherzyków są rozłączne; żadna wymiana nie rozpoczyna się przed zakończeniem poprzedniej. W ten sposób zmienne można ponownie wykorzystać. Gdy w praktyce używa się zmiennych, wyrażenie „zawartość" jest zwykle pomijane i piszemy takie rzeczy jak X ← 0 (czytaj „X dostaje 0" lub „ustaw X na 0"), aby ustawić początkową wartość (czyli zawartość z) X na 0 i X ← X + 1 (odczytaj „X otrzymuje X + 1"), aby zwiększyć wartość X o 1. Ta ostatnia instrukcja, na przykład, mówi procesorowi, aby „odczytał" liczbę znalezioną w X , zwiększ go o jeden i zastąp go wynikiem.

Wektory lub listy

Przyjrzyjmy się bliżej naszej liście pracowników. Taka lista może być postrzegana po prostu jako wiele elementów danych, które możemy zdecydować się zachować lub przechowywać w wielu zmiennych, powiedzmy X, Y, Z, . . . To oczywiście nie pozwoliłoby algorytmowi o stałym rozmiarze „przebiegać" przez listę, której długość może być różna, ponieważ każdy element na liście musiałby się odnosić w algorytmie za pomocą unikalnej nazwy. Dłuższe listy wymagałyby większej liczby nazw zmiennych, a tym samym dłuższych algorytmów. Potrzebujemy sposobu odwoływania się do wielu elementów w jednolity sposób. Potrzebujemy list zmiennych, które można „przejrzeć" lub uzyskać do nich dostęp w

inny sposób, ale bez konieczności jawnego nazywania każdego z ich elementów. aby móc „wskazywać” na elementy na tych listach, odnosić się do „następnego” lub „poprzedniego” elementu i tak dalej. Do tych celów wykorzystujemy wektory, zwane również tablicami jednowymiarowymi. Jeśli zmienna jest jak pokój hotelowy, wektor można traktować jako cały korytarz lub piętro w hotelu. Pokoje 301, 302, . . . , 346 mogą być indywidualnymi „zmiennymi”; do każdego można się odwoływać osobno, ale w przeciwieństwie do prostego zbioru zmiennych, cały korytarz lub piętro ma również nazwę (powiedzmy „piętro 3”), a dostęp do znajdujących się w nim pomieszczeń można uzyskać za pomocą ich indeksu. 15. pokój wzdłuż tego korytarza to 315, a X pokój to 3X. Oznacza to, że możemy użyć zmiennej do indeksowania wektora. Zmiana wartości X może być wykorzystana do odniesienia się do zawartości różnych elementów w wektorze. Notacja stosowana w praktyce oddziela nazwę wektora od jego indeksu nawiasami; piszemy V[6] dla szóstego elementu wektora V i analogicznie V[X] dla elementu V, którego indeksem jest bieżąca wartość zmiennej X. Możemy nawet napisać V[X + 1], który odnosi się do element następujący po V[X] na liście. (Zauważ, że V[X + 1] i V[X] + 1 oznaczają dwie zupełnie różne rzeczy!)

W algorytmie bubblesort możemy na przykład użyć zmiennej X do sterowania pętlą wewnętrzną, a jednocześnie może ona podwoić się jako wskaźnik do wektora V sortowanych elementów. Oto jak może wyglądać wynikowa (bardziej zwięzła, a także bardziej precyzyjna) wersja algorytmu, gdzie „<” oznacza „jest mniejsze niż”:

(1) wykonaj następujące N - 1 razy:

(1.1) X ← 1;

(1.2) podczas gdy X < N wykonaj następujące czynności:

(1.2.1) jeśli V[X + 1] < V[X] to je zamień;

(1.2.2) X ← X + 1.

Zachęcamy do zmodyfikowania tego algorytmu, uwzględniając wspomnianą wcześniej obserwację, tak aby przy każdym przejściu pętli zewnętrznej liczbę elementów sprawdzanych w pętli wewnętrznej można było zmniejszyć o 1. Wektory reprezentujące listy elementów mają wiele zastosowań. Książka telefoniczna to lista, podobnie jak słowniki, akta osobowe, opisy inwentarza, wymagania dotyczące kursów i tak dalej. W pewnym sensie wektor jako struktura danych jest ściśle powiązany z pętlą jako strukturą kontrolną. Przechodzenie przez wektor (w celu inspekcji, wyszukiwania, sumowania itp.) jest zwykle wykonywane za pomocą pojedynczej konstrukcji iteracyjnej. Tak jak pętla jest strukturą kontrolną opisującą długie procesy, tak wektor jest strukturą danych do reprezentowania długich list elementów danych. Istnieją również specjalne „indeksowane” wersje iteracyjnych konstrukcji sterujących, dostosowane do przechodzenia przez wektory. Na przykład możemy napisać:

dla X od 1 do 100 wykonaj następujące czynności

który jest podobny do:

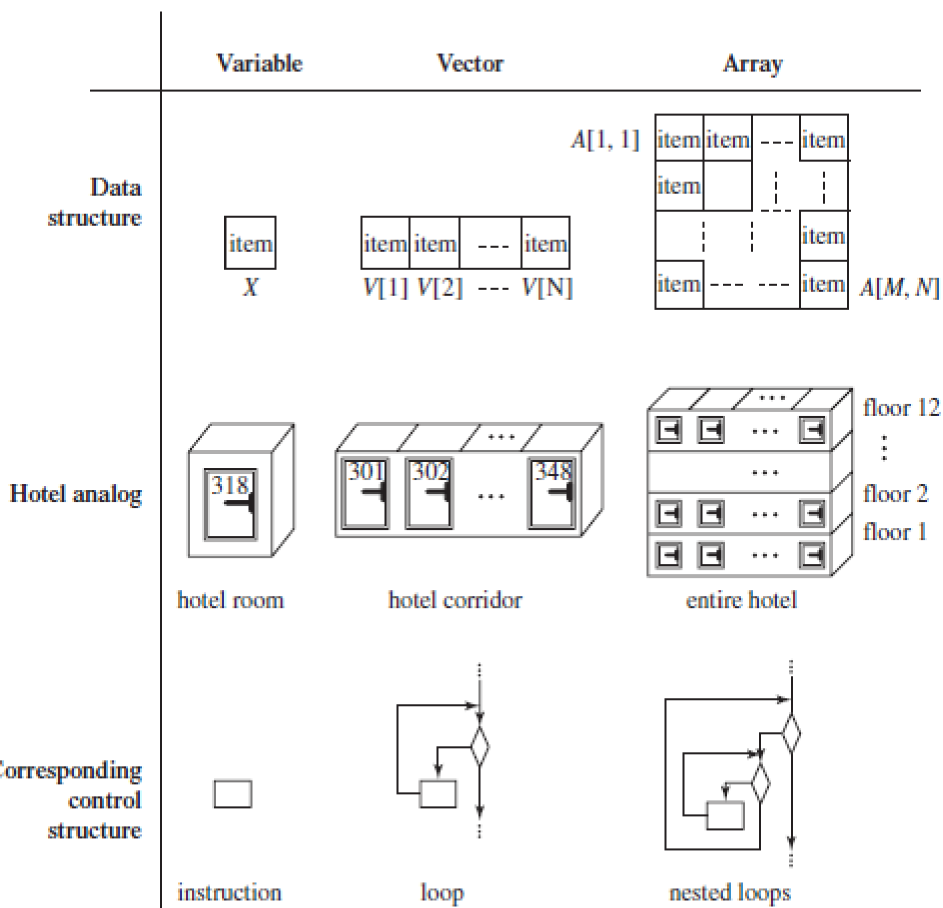
wykonaj następujące 100 razy

z tym wyjątkiem, że w przypadku pierwszego możemy odnieść się bezpośrednio do X-tego elementu w wektorze w powtarzającym się segmencie, podczas gdy w przypadku drugiego nie możemy.

Tablice lub tabele

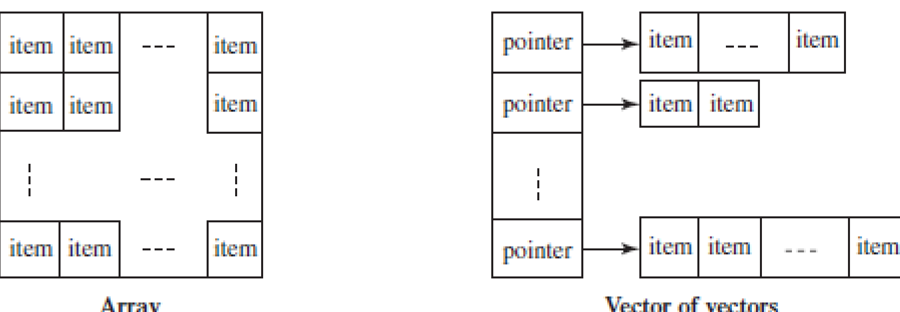
W wielu przypadkach wygodnie jest ułożyć dane nie w postaci prostej, jednowymiarowej listy, ale w formie tabeli. Odpowiednia algorytmiczna struktura danych nazywana jest macierzą, dwuwymiarową

tablicą lub po prostu tablicą. Również tutaj aplikacje są obfite. Standardowa tabliczka mnożenia drugiej klasy to tablica 10 na 10, w której element danych w każdym punkcie jest iloczynem indeksów wiersza i kolumny; listę uczniów wykreśloną na liście kursów można traktować jako tablicę, w której elementy danych są oceną ucznia z kursu; siatka szerokości i długości geograficznej Ziemi może być podstawą tablicy podającej wysokości w każdym punkcie przecięcia i tak dalej. Odwołanie się do elementu tablicy jest zazwyczaj realizowane za pomocą dwóch indeksów, wiersza i kolumny. Piszemy A[5, 3] dla elementu znajdującego się w wierszu 5 i kolumnie 3, tak że np. jeśli A jest tabliczką mnożenia, to wartość A[5, 3] wynosi 15. Tak jak poprzednio, możemy zapisać A[X, Y], a także A[X + 4, 17 − Y] i tym podobne. Jeśli zmienna jest jak pokój hotelowy, a wektor jak hotelowy korytarz/piętro, to macierz lub tablica jest jak cały hotel. Jego „rzędy" to różne piętra, a „kolumny" reprezentują lokalizacje wzdłuż korytarza/piętra. Jeśli wektor jako struktura danych odpowiada pętli jako strukturze kontrolnej, to tablica odpowiada zagnieżdżonym pętlom



Przechodzenie przez całą gamę ocen uczniów można osiągnąć za pomocą zewnętrznej pętli przebiegającej przez wszystkich uczniów i wewnętrznej, przebiegającej przez wszystkie oceny danego ucznia lub odwrotnie. Nie zawsze jest tak, że dane mogą być uporządkowane w ściśle prostokątnym formacie. Weźmy przykład studentów i kursów; różni studenci mogą być powiązani nie tylko z różnymi kursami, ale także z różną ich liczbą. Nadal możemy używać tablicy (wystarczająco szerokiej, aby pomieścić maksymalną liczbę możliwych kursów), ale pozostawiając jej części puste, gdziekolwiek uczeń pytanie nie przeszło danego kursu. Alternatywnie możemy użyć nowej struktury danych,

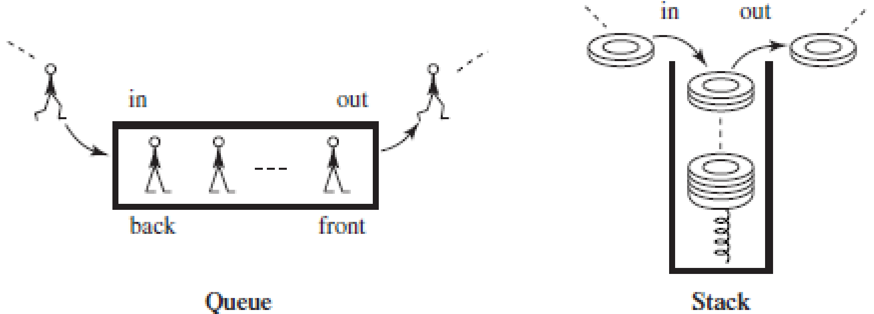
składającej się z wektora, którego elementy same w sobie wskazują na wektory o różnej długości. Różnicę ilustruje rysunek



Algorytmy mogą używać bardziej skomplikowanych tablic, na przykład o większej wymiarowości. Trójwymiarowa tablica jest jak sześcian z trzema indeksami potrzebnymi do wskazania elementu. W razie potrzeby możemy wykorzystać tylko specjalne części tablic, takie jak górna trójkątna część tablicy dwuwymiarowej, uzyskana przez ograniczenie indeksów w A[X, Y] tak, że X jest mniejsze niż Y do elementu. W razie potrzeby możemy wykorzystać tylko specjalne części tablic, takie jak górna trójkątna część tablicy dwuwymiarowej, uzyskana przez ograniczenie indeksów w A[X, Y] tak, że X jest mniejsze niż Y

Kolejki i stosy

Ciekawa wariacja na temat wektora/macierzy wynika z obserwacji, że w wielu zastosowaniach wektorów i tablic nie potrzebujemy pełnej mocy zapewnianej przez indeksy. Czasami lista służy jedynie do modelowania kolejki, w którym to przypadku wszystko, czego algorytm potrzebuje w interakcji z listą, to możliwość dodawania elementów z tyłu" i usuwania ich z „przodu". Innym razem lista ma na celu modelowanie stosu, jak te używane w restauracji do przechowywania talerzy. Tutaj algorytm wymaga dodawania i usuwania zdolności tylko na jednym końcu listy, na jej „szczycie". Kolejka jest czasami określana jako lista FIFO (pierwsze weszło-pierwsze wyszło), a stos jako lista LIFO (ostatnie weszło-pierwsze wyszło). Elementy są wpychane na stos, przy czym najwyższy element jest jedynym odsłoniętym do kontroli, a następnie stos można zdjąć, co oznacza, że najwyższy element jest usuwany.

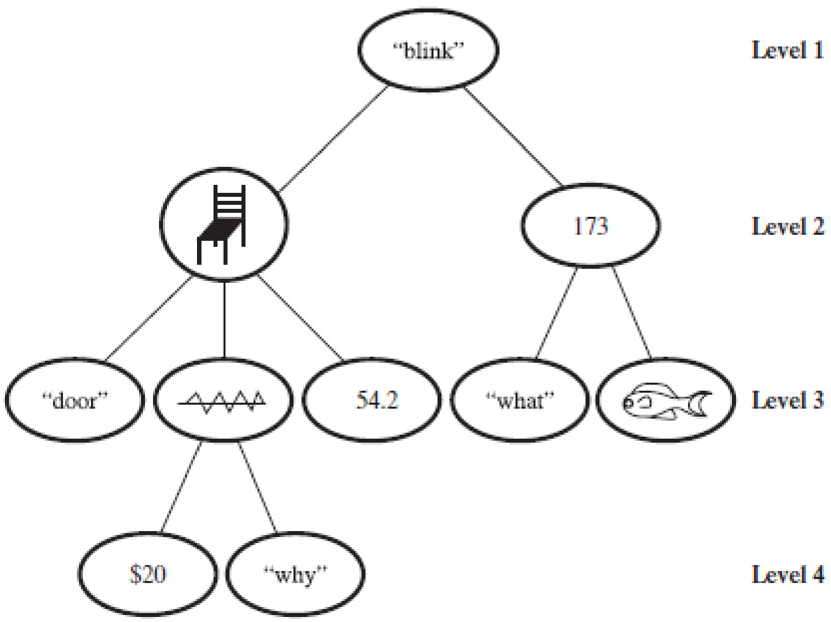


Chodzi o to, że warto, przynajmniej ze względu na przejrzystość algorytmów, myśleć o kolejkach i stosach jako o strukturach danych samych w sobie, a nie o specjalnych rodzajach list. Możemy wtedy użyć specjalnie opracowanych instrukcji elementarnych, takich jak „dodaj X do kolejki A" lub „wsuń X na stos S", zamiast niejasnych sformułowań, które wprost zawierają indeksy.

* Omawiając rekurencję wcześniej, niejawnie natknęliśmy się na potrzebę posiadania wektora, który w rzeczywistości jest używany tylko jako stos. Szczegóły nie zostaną tutaj przedstawione, ale zachęcamy do zastanowienia się nad sposobem, w jaki procesor pamięta, gdzie naprawdę się znajduje, podczas wykonywania algorytmu rekurencyjnego. Zapamiętywanie jego lokalizacji w tekście algorytmu nie stanowi problemu. To, co wymaga prowadzenia księgowości, to zarządzanie listą wywołań rekurencyjnych, które nie zostały jeszcze zakończone, i ustalenie, dokąd wrócić po zakończeniu każdego wywołania. Nietrudno zauważyć – a do zilustrowania tego możemy użyć algorytmu Wież Hanoi – że faktycznie potrzebujemy stosu. Ilekroć jest proszony o ponowne wejście do procedury przez wywołanie rekurencyjne, procesor „wpycha" swój adres zwrotny i bieżące wartości parametrów na stos. (W przykładzie z wieżami do wyboru są dwa takie możliwe adresy, odpowiadające lokalizacjom dwóch instrukcji wywołania w algorytmie.) Po zakończeniu wykonywania procedury rekurencyjnej odczytuje „pchnięte" informacje z góry stosu, przywraca stare wartości do parametrów i powraca do podanego adresu w tekście algorytmu. Jednocześnie „zrzuca" te informacje ze szczytu stosu i odrzuca je. Innym przykładem zastosowania stosów jest przemierzanie labiryntu poprzez wyczerpanie wszystkich możliwości. Takie przemierzanie wymaga utrzymywania listy odwiedzonych już skrzyżowań, dodawania do stosu nowych w miarę ich osiągania oraz usuwania tych, których wszystkie ścieżki zostały przebyte. W ten sposób stos zawsze zawiera ścieżkę od początku do aktualnie odwiedzanego punktu labiryntu.

Drzewa lub hierarchie

Jedną z najważniejszych i najbardziej wyróżniających się istniejących struktur danych jest drzewo. Nie jest to drzewo w botanicznym znaczeniu tego słowa, ale drzewo o bardziej abstrakcyjnej naturze. Wszyscy widzieliśmy takie drzewa używane do opisywania powiązań rodzinnych. Dwa najczęstsze rodzaje drzew genealogicznych to „drzewo przodków", które zaczyna się od osoby i działa wstecz przez rodziców, dziadków itd., oraz „drzewo potomków", które działa naprzód poprzez dzieci, wnuki itd. . Drzewo to zasadniczo hierarchiczny układ danych. Jeden przedmiot znajduje się w specjalnym miejscu zwanym korzeniem, a pozostałe są zorganizowane jako potomkowie korzenia. W informatyce drzewa są zwykle wizualizowane „do góry nogami" – korzeń u góry, a reszta drzewa rozłożona poniżej. Użyta terminologia jest dziwną mieszanką terminów z matematyki, botaniki i życia rodzinnego jednorodzicielskiego. Mówimy o korzeniu, o węzłach drzewa (punktach w drzewie), jego potomstwie (bezpośrednim potomstwie), jego liściach (węzły na „dno" drzewa, bez potomstwa), jego ścieżki lub gałęzie (sekwencje węzłów odpowiadające trawersom w dół w kierunku od korzenia do liścia), a także o rodzicach, przodkach, potomkach i rodzeństwie (dwa węzły są rodzeństwem, jeśli mają tego samego rodzica). Rysunek przedstawia drzewo, celowo skonstruowane bez konkretnego wyjaśnienia jego zawartości lub układu.

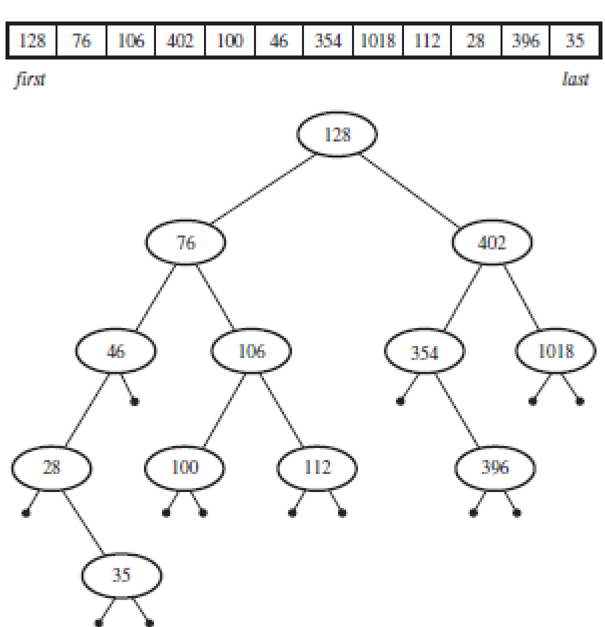


Niemniej jednak słowo „mignięcie" jest jego korzeniem lub, używając innej terminologii, jest to węzeł na poziomie 1 drzewa; Krzesło i liczba 173 to potomstwo korzenia lub, równoważnie, są to węzły na poziomie 2 drzewa i tak dalej. Przykładem drzew znanych użytkownikom komputerów jest hierarchiczna organizacja plików w katalogach, które same mogą znajdować się w innych katalogach. Drzewa można znaleźć gdzie indziej w życiu codziennym. Schematy organizacyjne większości firm to drzewa, podobnie jak schematyczne opisy rozpadu złożonych maszyn. Nawet same algorytmy często można opisać w strukturze drzewa; z grubsza mówiąc, poziomy drzewa odpowiadają poziomom reprezentowanym przez sekwencje liczb, których używamy w tej książce do oznaczania instrukcji. Inny ważny przykład dotyczy drzew łownych. Na przykład korzeń drzewa szachowego zawiera opis konfiguracji szachownicy otwierającej. Potomstwo korzenia reprezentuje 20 możliwych konfiguracji szachownicy wynikających z wykonania przez białe pierwszego ruchu, a potomstwo każdego z nich reprezentuje wyniki wszystkich możliwych odpowiedzi ze strony czarnych i tak dalej. Większość dwuosobowych drzewek do gry, podobnie jak szachy, ma szereg interesujących właściwości. Poziomy o numerach nieparzystych odpowiadają turze pierwszego gracza, podczas gdy te o numerach parzystych odpowiadają turze drugiego gracza. Ścieżki odpowiadają rzeczywistym grom, a każda możliwa gra pojawia się jako ścieżka w drzewie. Wreszcie listki reprezentują zakończenia gry (na przykład w szachach przez mata lub trzykrotne powtórzenie). W dalszej części książki będziemy mieli okazję wrócić do drzew łownych. Drzewa są używane w wielu różnych zastosowaniach i bardziej niż jakakolwiek inna strukturalna metoda łączenia części w całość można powiedzieć, że reprezentują szczególny rodzaj strukturyzacji, który obfituje w algorytmy.

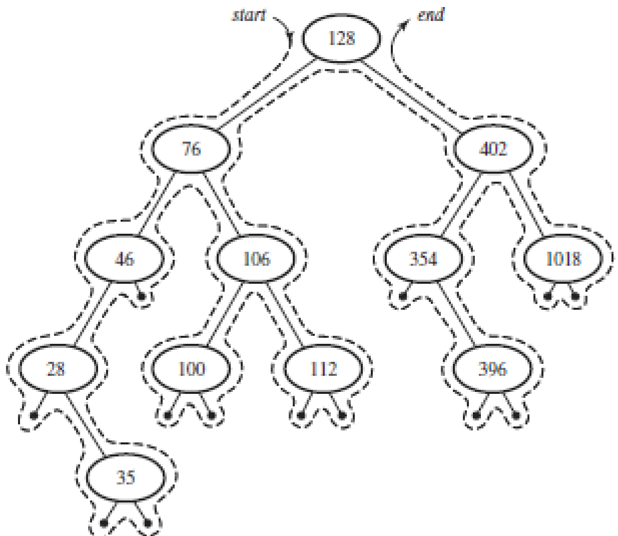
Sortowanie drzew: przykład

Aby dać pewne pojęcie o użyteczności drzew, rozważmy teraz inną procedurę sortowania, opartą na drzewach binarnych. Drzewo binarne to takie, w którym każdy węzeł ma najwyżej dwoje potomków. (Na przykład drzewo genealogiczne przodków jest binarne.) Drzewo binarne można również zdefiniować jako drzewo, którego stopień wyjściowy jest ograniczony przez 2. Zaletą tego ostatniego terminu jest jego ogólność; możemy mówić o drzewach o stopniu wyjściowym 17 lub 938, a nawet o tych z nieskończonym stopniem wyjściowym. Wracając do drzew binarnych, ponieważ stopień wyjściowy w każdym węźle wynosi co najwyżej 2, wygodnie jest rozróżnić te dwa potomstwa, nazywając je odpowiednio lewą i prawą. Treesort, jak będziemy to nazywać, składa się z dwóch

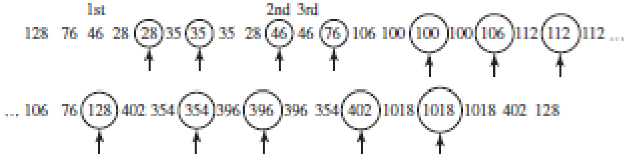
głównych kroków: (1) przekształcenia listy wejściowej w binarne drzewo poszukiwań T ; (2) przejść przez T najpierw w lewo i wyprowadzić każdy element podczas drugiej wizyty. Wyjaśnienie jest tutaj w porządku. Aby posortować listę elementów (powiedzmy liczb), algorytm najpierw organizuje elementy w specjalny rodzaj drzewa binarnego, zwanego drzewem wyszukiwania binarnego. Każdy węzeł tego drzewa ma następującą własność: wszyscy jego „lewi potomkowie" (czyli wszystkie elementy w całym jego lewym poddrzewie, nie tylko jego bezpośrednie potomstwo z lewej strony) mają mniejszą wartość niż wartość węzła samego i wszyscy jego prawi potomkowie są od niego większi. Rysunek 1 pokazuje przykład takiego drzewa wyszukiwania binarnego.



Drzewo to można skonstruować w następujący sposób: pierwszy element na liście jest traktowany jako korzeń, a następnie każdy element jest rozpatrywany po kolei i dołączany do rosnącego drzewa jako nowy liść, prawdopodobnie „odpuszczając" wcześniej wstawione liście procesu. Znalezienie właściwego miejsca nowego elementu jako liścia w drzewie odbywa się poprzez wielokrotne porównywanie go z otrzymanymi węzłami, obracając się w lewo lub w prawo w zależności od wyniku porównania. Jeśli nowy element jest mniejszy niż element w węźle, na który patrzymy, idziemy w lewo (ponieważ wtedy należy do lewej strony tego węzła); w przeciwnym razie idziemy w prawo. Powinieneś skorzystać z rysunku, aby zapoznać się z tą procedurą, a następnie spróbować zapisać podprogram reprezentujący krok (1). Teraz nadchodzi ciekawa część. Po zbudowaniu binarnego drzewa wyszukiwania, drugi etap sortowania drzew wymaga przechodzenia przez nie w następujący sposób. Procesor zaczyna się od nasady i porusza się w dół, zawsze trzymając się lewej strony. Za każdym razem, gdy próbuje poruszyć się w lewo, ale nie może (na przykład, gdy nie ma potomstwa leworęcznego), porusza się niechętnie w prawo. Wyczerpawszy zarówno lewą, jak i prawą stronę, albo dlatego, że nie znajduje potomstwa, albo dlatego, że już je odwiedził, wycofuje się; to znaczy, że przesuwa się w górę do rodzica bieżącego węzła. Jeśli ten ruch w górę zakończy podróż z węzła macierzystego do jego lewego potomstwa, procesor skręca w prawo i ponownie opada w dół; ale jeśli właśnie wrócił z podróży w prawo, w rzeczywistości wyczerpał obie możliwości węzła rodzica, a więc przechodzi do swojego rodzica. Proces ten trwa do wyczerpania obu kierunków w dół korzenia drzewa, przemierzając w ten sposób całe drzewo. Rysunek 2 pokazuje pierwsze przejście w lewo drzewa z rysunku 1.



Dlaczego dokonujemy tego dziwnego przejścia? Cóż, najpierw zauważ, że jeśli uznamy, że nieobecność potomstwa powoduje pojawienie się krótkiego „kikuta" lub „ślepego zaułka", który również musi zostać przebyty, to każdy węzeł na drzewie jest „odwiedzany" dokładnie trzy razy. Teraz, jeśli podczas przemierzania drzewa w ten sposób konsekwentnie wypisujemy element danych znaleziony w węźle podczas odwiedzania go dokładnie po raz drugi, lista ostatecznie zostanie wysłana do wyjścia w całości, posortowana w porządku rosnącym! Może to brzmieć jak magia, ale można to łatwo zilustrować, podążając za skręconymi strzałkami przecinającymi się na Rysunku 2, zapisując liczby podczas ich odwiedzania, a następnie zaznaczając drugie pojawienie się każdego z nich na liście wynikowej. Znaki te tworzą posortowaną listę



Oba etapy algorytmu sortowania drzew można dość łatwo opisać jako podprogramy rekurencyjne. Oto procedura dla drugiego etapu, w której używamy „left(T)" do oznaczenia lewego poddrzewa T , i podobnie do „right(T)". Zgodnie z naszą konwencją dotyczącą obecności pniaków, gdy potomstwo nie istnieje, „left(T)," dla drzewa, które nie ma lewego poddrzewa, będzie specjalnym pustym drzewem; to znaczy drzewo nie zawierające niczego, nawet korzenia podprogramu drugiej-wizyty-przechodzenia-z T :

(1) jeśli T jest puste, to zwróć;

(2) w przeciwnym razie (tj. jeśli T nie jest puste) wykonaj następujące czynności:

(2.1) wywołaj drugą wizytę-traversal-of left(T);

(2.2) wypisz element danych znaleziony w korzeniu T ;

(2.3) zadzwoń do drugiej wizyty-przejścia w prawo(T);

(3) powrót.

Konstruując tę procedurę, wykorzystaliśmy fakt, że drugie odwiedziny węzła zawsze będą miały miejsce tuż po zakończeniu przechodzenia przez całe jego lewe poddrzewo i tuż przed rozpoczęciem przechodzenia jego prawego poddrzewa. Tak więc najpierw wywołujemy podprogram rekurencyjnie w celu ukończenia całego przejścia dla lewego poddrzewa, a kiedy to wywołanie się zakończy (to znaczy, że przejście tego poddrzewa zostało zakończone), wyprowadzamy element w korzeniu, ponieważ jest teraz odwiedzany po raz drugi (za pierwszym razem, gdy przeszliśmy w dół do jego lewego poddrzewa). Następnie przechodzimy rekursywnie w dół w prawo, aby wykonać przejście w prawo. Charakter „lewo-najpierw" tego specjalnego przejścia staje się dość oczywisty ze struktury sformułowania rekurencyjnego, ponieważ dla każdego poddrzewa (to znaczy dla każdego wywołania rekurencyjnego) można łatwo zauważyć, że procedura przechodzi najpierw w lewo, a następnie prawo. Nawiasem mówiąc, interesujące jest zobaczenie, co dzieje się z listą wyników, kiedy to odwracamy, idąc konsekwentnie najpierw w prawo, a potem w lewo. (Co się dzieje i jak odpowiednio zmieniamy podprogram?) Struktura tej procedury przypomina procedurę ruchu dla Wież Hanoi. W rzeczywistości, jeśli zastosujemy tę procedurę do przykładowego drzewa, a następnie narysujemy obraz trudów procesora, obraz będzie dokładnie odzwierciedlał kształt przykładowego drzewa. Możemy zatem stwierdzić, że jeśli wektory i tablice jako struktury danych odpowiadają pętlom, a zagnieżdżone pętle jako strukturom kontrolnym, to drzewa odpowiadają rekurencyjnym podprogramom. Drzewo jest z natury rekurencyjnym obiektem, składającym się z niczego (tj. z pustego drzewa, jak wyjaśniono wcześniej) lub z korzenia dołączonego (rekursywnie) do pewnej liczby drzew. To wyjaśnia, dlaczego przechodzenie przez drzewa, takie jak to właśnie omówione, jest stosunkowo łatwe do opisania rekurencyjnie. Istnieje kilka interesujących sposobów na wprowadzenie elastyczności do istniejących struktur danych. Dobrym przykładem jest koncepcja samoregulacji. Rozumiemy przez to, że za każdym razem, gdy element jest wstawiany lub usuwany ze struktury danych, struktura wykonuje procedurę dostosowania, która wprowadza proste zmiany, mające na celu zachowanie pewnych „ładnych" właściwości. Na przykład w wielu zastosowaniach drzew możliwe jest, że ze względu na pewną sekwencję wstawień/delecji drzewo stanie się cienkie i długie, podczas gdy ze względu na wydajność może być pożądane, aby było szerokie i krótkie. Możliwe jest - choć nie będziemy wchodzić w szczegóły - dokonywanie niewielkich lokalnych zmian w drzewie za każdym razem, gdy jest na nim wykonywana operacja, co zagwarantuje właściwość szerokości i zwięzłości.

Bazy danych i bazy wiedzy

Dla wielu aplikacji komputerowych struktury danych nie wystarczają. Nie zawsze chodzi tylko o rozważenie problemu algorytmicznego i znalezienie lub zdefiniowanie ładnych i użytecznych struktur danych do jego rozwiązania. Czasami istnieje potrzeba bardzo dużej „puli" danych, którą wiele algorytmów traktuje jako część swoich danych wejściowych, a zatem musi mieć stałą strukturę i być łatwo dostępne do wyszukiwania i manipulacji. Przykłady obejmują dane finansowe i osobowe firmy, rezerwacje i informacje o lotach linii lotniczej oraz dane indeksowe biblioteki. Takie masy danych nazywamy bazami danych. Zazwyczaj bazy danych są bardzo duże i zawierają wiele różnych rodzajów danych, od nazw i adresów po niejasne kody i symbole, a w niektórych przypadkach nawet dowolny tekst. Są one zwykle poddawane licznym rodzajom procedur wstawiania, usuwania i pobierania, wykorzystywanych do różnych celów i przez różnych ludzi. O ile dodanie nowego studenta do uniwersyteckiej bazy danych lub usunięcie starej to stosunkowo łatwe zadanie, to w przypadku przeszukiwania bazy danych w celu uzyskania z niej informacji zawiłość wydaje się nie mieć granic. Tylko dla perspektywy, oto kilka zapytań, które można skierować do bazy danych systemu rezerwacji lotów:

- wymień wszystkich pasażerów potwierdzonych lotem 123;
- wymienić wszystkie miejsca w locie 123 zajęte przez pasażerów bez bagażu rejestrowanego;

- znaleźć liczbę pasażerów, którzy zarezerwowali miejsca na dwa identycznie ponumerowane loty zaplanowane na kolejne dni marca tego roku;
- podać nazwiska i numery miejsc wszystkich pasażerów pierwszej klasy na jutrzejszych lotach do Paryża, którzy zamówili wegetariańskie posiłki i mają międzykontynentalne loty kontynuacyjne, na których usługi w klasie ekonomicznej nie zapewniają specjalnych posiłków i których międzylądowanie znajduje się w Zurychu lub Rzym.

Te przykłady ilustrują znaczenie „dobrej" organizacji bazy danych. Jeśli loty kontynuacyjne znajdują się w bazie danych w jakimś niejasnym miejscu i jeśli nie ma łatwego sposobu na zebranie informacji o trasie danego pasażera, to napisanie algorytmu, który rozwiąże ostatni wymieniony problem z odzyskaniem, stanie się naprawdę trudnym zadaniem. Podobnie jak w przypadku struktur danych, dobry projekt bazy danych jest ważny nie tylko ze względu na przejrzystość i łatwość pisania; może to mieć ogromne znaczenie, jeśli chodzi o kwestie wydajności i wykonalności zbudowania systemu bazy danych, który będzie w stanie odpowiedzieć na takie zapytania w rozsądnym czasie. Zaproponowano różne ogólne modele organizacji baz danych, które są wykorzystywane w rzeczywistych bazach danych. Modele te są przeznaczone do przechowywania dużych ilości danych przy jednoczesnym wiernym i wydajnym uchwyceniu relacji między elementami danych. Jeden z najpopularniejszych modeli baz danych, model relacyjny, umożliwia rozmieszczenie danych w dużych tabelach, przypominających strukturę danych tablicowych. Inni domagają się pewnych rodzajów układów podobnych do drzewa lub sieci, takich jak model hierarchiczny, który organizuje dane w wielowarstwowej formie przypominającej drzewo. Wydaje się, że nie ma szerokiego konsensusu co do preferowanych modeli dla poszczególnych zastosowań oraz wielu metod i języków zostały opracowane do manipulowania i odpytywania baz danych skonstruowanych w każdym z tych modeli. Istnieje jednak znaczący ruch w kierunku modelu relacyjnego, a nowsze bazy danych obiektowych czekają poza sceną na swoją kolej. Bazy danych są często wykorzystywane do obsługi systemów operacyjnych, które śledzą zmiany w niezliczonych szczegółach niezbędnych do funkcjonowania dużej organizacji. W związku z tym muszą wspierać skuteczne modyfikacje danych, a także ich zapytania, ale mniej dbają o przechowywanie informacji, które mają jedynie wartość historyczną. Jednak duże ilości danych przechowywane w bazach danych mogą być również wykorzystywane do celów analitycznych, takich jak odkrywanie trendów czy usprawnianie procesów. Na przykład bank może chcieć odkryć korelacje między niespłacaniem kredytów a innymi właściwościami posiadacza rachunku w celu opracowania lepszych narzędzi prognostycznych. Można to zrobić, stosując metody analizy statystycznej do dużej ilości starych danych dostępnych we własnych bazach danych banku. Nauka o odkrywaniu takich użytecznych bryłek informacji z ogromnych źródeł danych nazywana jest eksploracją danych i zajmuje się głównie niezmiennymi danymi historycznymi. Zazwyczaj ilość danych historycznych jest znacznie większa niż potrzebna do celów operacyjnych; często setki razy większe. W ostatnich latach biologia oferuje szczególnie kuszące rodzaje baz danych, wynikające z projektu genomu i jego produktów ubocznych, które zaczynają wymagać opracowania nowych, potężnych technik eksploracji danych. Nowy rodzaj bazy danych, zwany hurtownią danych, został opracowany w celu obsługi ogromnych ilości danych, które zmieniają się powoli lub w których w ogóle zmieniają się tylko bardzo małe porcje. Niektóre hurtownie danych wydają się użytkownikom podobne do innych baz danych; jednak organizacja wewnętrzna może używać zupełnie innego zestawu algorytmów, aby skutecznie osiągać swoje cele. Pewne rodzaje danych są lepiej postrzegane jako fragmenty wiedzy, a nie tylko liczby, nazwy czy kody. Oprócz dużej bazy danych opisującej inwentarz firmy produkcyjnej, możemy chcieć mieć dużą bazę wiedzy zawierającą informacje istotne dla prowadzenia tej firmy. Jego elementy wiedzy mogą w jakiś sposób zakodować informacje, takie jak „Zmiany wynagrodzeń to kwestie personalne", „Pan Smith jest lepszym menedżerem niż pani Brown, jeśli chodzi o problemy kadrowe, ale nie w kwestiach technicznych" lub „Jeśli cena ropy idzie w górę, w następnym miesiącu będziemy musieli

obniżyć wszystkie pensje". W przeciwieństwie do bazy danych, która przechowuje dane do późniejszego pobrania, baza wiedzy wykorzystuje swoją wiedzę w bardziej wyrafinowany sposób. Na przykład, z powyższych reguł i faktu, że ceny ropy wzrosły, moglibyśmy chcieć wnioskować, że pan Smith powinien zająć się zmianami płac. Jest oczywiste, że takie możliwości wnioskowania wymagają bardziej złożonej organizacji niż elementy danych o mniej lub bardziej stałym formacie, zwłaszcza jeśli zależy nam na wydajnym wyszukiwaniu. Bazy wiedzy stają się zatem naturalnym kolejnym krokiem po bazach danych i stanowią bogate źródło interesujących pytań dotyczących reprezentacji, organizacji i wyszukiwania algorytmicznego.

Ćwiczenia

2.1. Przedstawiony w tekście algorytm sumowania wynagrodzeń N pracowników wykonuje pętlę polegającą na dodaniu jednej pensji do sumy i przesunięciu wskaźnika na liście pracowników N -1 razy. Ostatnie wynagrodzenie doliczane jest osobno. Jaki jest tego powód? Dlaczego nie wykonamy pętli N razy?

2.2. Rozważmy algorytm bubblesort przedstawiony w tekście.

(a) Wyjaśnij, dlaczego pętla zewnętrzna jest wykonywana tylko N - 1 razy.

(b) Popraw algorytm tak, aby przy każdym powtórzonym wykonaniu pętli zewnętrznej pętla wewnętrzna sprawdzała o jeden element mniej.

2.3. Przygotuj schematy blokowe dla algorytmu sortowania bąbelkowego przedstawionego w tekście oraz dla ulepszonej wersji, którą poproszono o zaprojektowanie w ćwiczeniu 2.2.

2.4. Napisz algorytmy, które przy danej liczbie całkowitej N i liście L składającej się z N liczb całkowitych dają w S i P sumę liczb parzystych występujących odpowiednio w L i iloczyn liczb nieparzystych.

(a) Korzystanie z iteracji ograniczonej.

(b) Używanie stwierdzeń „goto".

2.5. Pokaż, jak wykonać następujące symulacje niektórych konstrukcji sterowania przez innych. Konstrukcja sekwencjonowania „a następnie" jest domyślnie dostępna dla wszystkich symulacji. W razie potrzeby możesz wprowadzić i używać nowych zmiennych i etykiet.

(a) Symuluj pętlę „for-do" za pomocą pętli „if-then-else" za pomocą pętli „while d0".

(c) Symuluj pętlę „while-do" za pomocą instrukcji „if-then" i „goto".

(d) Symuluj pętlę „while do" za pomocą pętli „repeat until" i instrukcji „if then".

2.6. Zapisz sekwencję ruchów rozwiązujących problem Wież Hanoi dla pięciu pierścieni. Silnia nieujemnej liczby całkowitej N jest iloczynem wszystkich dodatnich liczb całkowitych mniejszych lub równych N. Bardziej formalnie, wyrażenie N silnia, oznaczane przez N!, jest rekurencyjnie definiowane przez $0! = 1$ i $(N + 1)! = N! \times (N + 1)$. Na przykład $1! = 1$ i $4! = 3! \times 4 = \ldots = 1 \times 2 \times 3 \times 4 = 24$.

2.7. Napisz algorytmy, które obliczają N!, mając nieujemną liczbę całkowitą N.

(a) Korzystanie z instrukcji iteracyjnych.

(b) Korzystanie z rekurencji.

2.8. Pokaż, jak symulować pętlę „podczas wykonywania” za pomocą instrukcji warunkowych i procedury rekurencyjnej. Dla dodatniej liczby całkowitej N oznaczmy przez AN zbiór liczb całkowitych od 1 do N. Permutacja zbioru AN jest uporządkowaną sekwencją (a_1, a_2,...,a_N), w której każda liczba całkowita ze zbioru A_N pojawia się dokładnie raz. Na przykład (2, 3, 1) i (1, 2, 3) to dwie różne permutacje zbioru A_3.

2.9. Udowodnij, że liczba permutacji A_N wynosi N!.

2.10. Permutację (a_1,...,a_N) można przedstawić za pomocą wektora P o długości N z P[I] = aI. Zaprojektuj algorytm, który przy danej liczbie całkowitej N i wektorze liczb całkowitych P o długości N sprawdza, czy P reprezentuje jakąkolwiek permutację A_N .

2.11. Zaprojektuj algorytm, który przy dodatniej liczbie całkowitej N wytwarza wszystkie permutacje A_N . Mówimy, że permutację σ = (a_1, ... , a_N) można uzyskać ze stosu, jeśli można rozpocząć od ciągu wejściowego (1, 2, ... , N) i pustego stosu S, i wytworzyć wynik σ stosując tylko następujące typy operacji:

read(X): Wczytaj liczbę całkowitą z wejścia do zmiennej X.

print(X): Wypisuje na wyjściu liczbę całkowitą aktualnie zapisaną w zmiennej X.

push(X, S): Włóż liczbę całkowitą aktualnie zapisaną w zmiennej X na stos S.

pop(X, S): Przenieś liczbę całkowitą ze szczytu stosu S do zmiennej X. (Ta operacja jest niedozwolona, jeśli S jest puste).

Na przykład permutację (2, 1) można uzyskać ze stosu, ponieważ następujące serie operacji

read(X), push(X, S), read(X), print(X), pop(X, S), print(X)

zastosowane do sekwencji wejściowej (1, 2) tworzy sekwencję wyjściową (2, 1). Permutację można uzyskać przez kolejkę, jeśli można ją w podobny sposób uzyskać z danych wejściowych (1, 2, ... , N), używając początkowo pustej kolejki Q i operacji read(X), print(X), i

add(X, Q): Dodaj liczbę całkowitą aktualnie przechowywaną w X na tył Q.

remove(X, Q): Usuń liczbę całkowitą z początku Q do X. (Ta operacja jest nieprawidłowa, jeśli Q jest puste.)

Podobnie możemy mówić o permutacji uzyskanej przez dwa stosy, jeśli pozwolimy na push i pop operacje na dwóch stosach S i S’ .

2.12.

(a) Pokaż, że stos można uzyskać następujące permutacje:

i. (3, 2, 1).

ii. (3, 4, 2, 1).

iii. (3, 5, 7, 6, 8, 4, 9, 2, 10, 1).

(b) Udowodnij, że stos nie może uzyskać następujących permutacji:

i. (3, 1, 2).

ii. (4, 5, 3, 7, 2, 1, 6).

(c) Ile permutacji A_4 nie może uzyskać stos?

2.13. Zaprojektuj algorytm sprawdzający, czy daną permutację można uzyskać za pomocą stosu. W przypadku odpowiedzi twierdzącej algorytm powinien również wypisać odpowiednią serię operacji. W swoim algorytmie, oprócz czytania, drukowania, wypychania i wyskakiwania, możesz użyć testu isempty(S) do testowania pustego stosu S.

2.14.

(a) Podaj serię operacji, które pokazują, że każda z permutacji podanych w ćwiczeniu 2.12(b) może być uzyskana przez kolejkę, a także przez dwa stosy.

(b) Udowodnij, że każdą permutację można uzyskać w kolejce.

(c) Udowodnij, że każdą permutację można uzyskać przez dwa stosy.

2.15. Rozszerz algorytm, który miałeś zaprojektować w ćwiczeniu 2.13, tak aby jeśli danej permutacji nie można uzyskać na stosie, algorytm wypisze serię operacji na dwóch stosach, które ją wygenerują.

2.16. Rozważmy algorytm sortowania drzewa opisany w tekście.

(a) Skonstruuj algorytm, który przekształca daną listę liczb całkowitych w drzewo wyszukiwania binarnego.

(b) Jak wyglądałby wynik funkcji sortowania drzewa, gdybyśmy odwrócili kolejność, w której podprocedura drugiej wizyty-przemierzania wywołuje się rekurencyjnie? Innymi słowy, konsekwentnie odwiedzamy prawe potomstwo węzła, zanim odwiedzimy lewe.

Języki programowania i paradygmaty

Myślenie o problemach algorytmicznych i ich rozwiązywaniu jest w porządku, a właściwie, jak powinno być już oczywiste, jest korzystne nie tylko w rozwiązywaniu problemów związanych z komputerem, ale także w sferze pieczenia ciast, wymiany opon, produkcji szafek i telefonu wyszukiwanie książek. Jednak najwyraźniej naszą główną troską są algorytmy, które są przeznaczone do wykonywania komputerowego, a zatem zanim przejdziemy do opisu naukowych aspektów algorytmiki, powinniśmy poświęcić trochę czasu na powiązanie algorytmów z prawdziwymi komputerami. Jak wspomniano w Części 1, nawet najbardziej wyrafinowany komputer to tak naprawdę tylko duża, dobrze zorganizowana objętość bitów, a ponadto może normalnie wykonywać na nich tylko niewielką liczbę niezwykle prostych operacji, takich jak zerowanie, odwracanie i testowanie . Jak przedstawić algorytm prawdziwemu komputerowi i zmusić go do przeprowadzenia odpowiedniego procesu zgodnie z zamierzeniami? Ujmijmy to bardziej dosadnie. Jak przekonać tę durną maszynę, zdolną tak niewiele, by imponująco wykonywała nasze subtelne i starannie opracowane algorytmy? Oczywiście pytanie to staje się tym bardziej naglące, gdy rozejrzymy się i zobaczymy komputery wykonujące nie tylko nasze zabawkowe przykłady sumowania wynagrodzeń i wyszukiwania słów, ale niezwykle złożone wyczyny, takie jak automatyczne sterowanie lotem, graficzna symulacja reakcji chemicznych czy uporządkowanie. utrzymanie sieci komunikacyjnych z milionami abonentów. Pierwsza obserwacja jest taka, że nasze algorytmy muszą być napisane w sposób jednoznaczny i formalny. Zanim spróbujemy zrozumieć, w jaki sposób można bawić się bitami, aby wykonać nawet proste zadanie, takie jak „przejść przez listę, dodając każdą pensję do zanotowanej liczby", samo zadanie musi być dokładnie określone: Gdzie komputer znajduje listę? Jak przez to przechodzi? Gdzie znajduje wynagrodzenie i „zanotowaną liczbę"? i tak dalej. „Ubijaj białka do uzyskania piany" nie jest dużo gorsze pod względem precyzji i jednoznaczności niż „przechodzenie przez listę". Aby opisać algorytm dla korzyści komputera, a nie tylko dla ludzkiego zrozumienia, używamy języka programowania, w którym piszemy programy. Program jest oficjalną i formalną wersją algorytmu, odpowiednią do wykonania komputerowego. Język programowania składa się z notacji i reguł, według których pisze się programy, a osobę piszącą program nazywamy programistą. (Ta osoba nie musi być tą, która opracowała algorytm, na którym oparty jest program.) Oczywiście „surowy" komputer nie rozumie bezpośrednio żadnego z tych języków programowania, a jedynie programy napisane w języku maszynowym, który składa się z instrukcje manipulacji bitami, zakodowane jako ciągi bitów. Jak wspomniano w Części 1, to właśnie użycie abstrakcji pozwala nam zachowywać się tak, jakby komputer faktycznie rozumiał języki programowania wysokiego poziomu, które tutaj omawiamy. Później przyjrzymy się niektórym mechanizmom tej abstrakcji.

Programy wymagają precyzyjnej składni

Język programowania jest zwykle kojarzony ze sztywną składnią, pozwalającą na użycie tylko specjalnych kombinacji wybranych symboli i słów kluczowych. Każda próba rozciągnięcia tej składni może okazać się katastrofalna; na przykład, jeśli dane wejściowe X są napisane w języku, którego polecenia wejściowe mają postać read X, istnieje prawdopodobieństwo, że wynik będzie tępym „SYNTAX ERROR E4514 ON LINE 108". To oczywiście wyklucza nawet takie uprzejme, ale nieprecyzyjne prośby, takie jak „proszę, przeczytaj wartość X z danych wejściowych" lub „co powiesz na uzyskanie mi wartości X". Może to skutkować długim ciągiem niejasnych komunikatów o błędach. Prawdą jest, że miłe, gadatliwe instrukcje są przyjemniejsze i być może mniej dwuznaczne niż ich lapidarne i bezosobowe odpowiedniki dla ludzkiego czytelnika. Prawdą jest również, że chcielibyśmy, aby komputery były jak najbardziej „przyjazne dla użytkownika". Należy jednak skonfrontować te fakty z obecną niezdolnością komputerów do rozumienia swobodnie mówionych (lub swobodnie pisanych) języków naturalnych, takich jak angielski. Dlatego niezbędny jest formalny, zwięzły i sztywny zestaw

reguł składniowych. Formalna składnia typowego języka programowania obejmuje uporządkowane wersje kilku struktur sterujących, uporządkowane sposoby definiowania różnych struktur danych oraz uporządkowane formaty podstawowych instrukcji obsługiwanych przez język. Różnica polega na tym, że teraz nasz wyimaginowany mały robot Runaround jest mniej wyimaginowany, ponieważ komputer wykonuje pracę i nie wykona żadnej instrukcji, nawet jasnej i jednoznacznej, jeśli ta instrukcja nie znajduje się wśród tych dozwolonych przez język programowania. Algorytm sumowania liczb od 1 do N można zapisać w typowym (hipotetycznym) języku programowania PL w następujący sposób:

input N;

X:=0;

for Y from 1 to N do

X:=X + Y

end;

output X.

Tutaj X:=0 jest instrukcją przypisania, która ustawia zmienną X na wartość początkową 0 (w naszych algorytmach zapisujemy w tym celu X ← 0). Zwróć uwagę na słowa kluczowe pogrubione; są one częścią składni języka PL, co oznacza, że formalna definicja tej składni najprawdopodobniej zawiera klauzulę, zgodnie z którą jednym z prawnych rodzajów zdań w języku jest stwierdzenie for, którego ogólny format składa się ze słowa kluczowego dla , po którym następuje prawny nagłówek, po którym następuje do, po którym następuje oświadczenie prawne, po którym następuje koniec1. następuje, gdzie „|” oznacza „lub”:

<instrukcja> to: <for-statement> | <przypisanie-instrukcji> | …

<for-statement> to: for<for-header> wykonaj <instrukcja> end

Opis klauzuli nagłówka może brzmieć:

<for-header> to: <zmienna> od <wartość> do <wartość>

<wartość> może być wtedy określona jako zmienna, jawna liczba lub jakiś inny obiekt.

W przykładzie pokazano, w jaki sposób język wymusza specjalny format dla określonego rodzaju konstrukcji kontroli pętli ograniczonej. Podobnie wymuszane są formaty służące do definiowania struktur danych. Na przykład dwuwymiarowa tablica AR o rozmiarze 50 na 100, której wartości mogą być liczbami całkowitymi, może być zdefiniowana w hipotetycznym języku PL przez stwierdzenie:

define AR array [1. . 50, 8 . . 107] of integer

a język może umożliwiać odwoływanie się do elementów AR za pomocą wyrażeń postaci:

AR(wartość; wartość)

Zwróć uwagę, że określiliśmy nie tylko wymiary AR, ale także jego wartości indeksów dolnych; jego rzędy są ponumerowane 1, 2, ..., 50, ale jego kolumny są (z jakiegoś powodu) ponumerowane 8, 9, &jellip; 107. W rzeczywistości składnia określa znacznie więcej niż dostępne struktury kontrolne i danych oraz dozwolone operacje elementarne. Zwykle język ma sztywne reguły nawet dla tak drobnych kwestii, jak prawne ciągi symboli, które mogą być używane do nazywania zmiennych lub struktur danych, oraz maksymalna liczba cyfr dozwolona w wartościach numerycznych. Niektóre języki ograniczają długość nazw zmiennych do, powiedzmy, ośmiu symboli, z których pierwszy jest

alfabetyczny. Tutaj również nie można przeoczyć interpunkcji; nie jest nietypowe, aby język programowania wymagał, aby średnik następował po niektórych rodzajach stwierdzeń, że spacja ma następować po innych, że komentarze mają być ujęte w specjalnych nawiasach i tak dalej, a wszystko to z karami za naruszenie. To jest powód, dla którego używamy terminu „zespół średnika” we wstępie do tej książki. Aby czytelnicy byli wystarczająco kompetentni, aby naginać komputery do ich woli, niektóre książki komputerowe zaczynają się od drobiazgowego opisu składni określonego języka programowania, reguł średników i wszystkiego, przy czym te żmudne szczegóły składni są dominującymi elementami studenta. jest zobowiązany do zapamiętania.

Programy wymagają precyzyjnej semantyki

Zapewnienie językowi precyzyjnej składni to tylko część pracy projektanta języka programowania. Bez formalnej i jednoznacznej semantyki - to znaczy bez znaczenia dla każdej składniowo dozwolonej frazy - składnia jest prawie bezwartościowa. Gdyby nie podano znaczenia instrukcji w języku, segment programu:

for Y from 1 to N do

w hipotetycznym języku PL może, o ile wiemy, oznaczać „odjąć Y od 1 i zapisać wynik w N”. Co gorsza, kto mówi, że słowa kluczowe od, do, do itd. mają w ogóle coś wspólnego z ich znaczeniem w konwencjonalnym języku? Może ten sam segment programu oznacza „wykasuj całą pamięć komputera, zmień wartości wszystkich zmiennych na 0, wypisz „DO PIEKŁA Z JĘZYKAMI PROGRAMOWANIA” i przestań! Kto mówi, że „:=” oznacza „przypisz do” i „ +” oznacza „plus?” Kto mówi, że polecenia są wykonywane w ich pisemnym porządku? Być może ";" oznacza „po wykonaniu następujących czynności” zamiast „a następnie wykonaj następujące czynności”. Oczywiście możemy się domyślić, o co chodzi, skoro projektant PL prawdopodobnie wybrał słowa kluczowe i specjalne symbole, aby ich znaczenie było jak najbardziej zbliżone do przyjętej normy. Ale nie można zmusić komputera do działania na podstawie takich założeń. Nawet gdyby mógł w jakiś sposób zrozumieć konwencjonalne znaczenie angielskich słów i symboli takich jak „+”, jak bez wyraźnego wyjaśnienia wiedziałby, co zrobić z nagłówkiem pętli:

for Y from 1 to N do

kiedy wartość N wynosi, powiedzmy, -314,1592? Czy ciało pętli ma być wykonane 316 razy, przy czym Y przechodzi przez wartości 1, 0,−1,−2, . . . ,-313 i -314? Być może przeprowadza się ją 317 razy, w tym sprawę −315; lub 633 razy, przechodząc przez 1, 12 , 0,−12 ,−1, . . . ,−315? Może w ogóle nie jest to przeprowadzane, ponieważ wartość N jest mniejsza niż 1, a pętle for muszą zwiększać swoje wbudowane liczniki, a nie je zmniejszać. Być może dojście do instrukcji for z tak dziwną wartością w N powinno być traktowane jako „LOOP INDEX ERROR Z3088 ON LINE 365”. Wydaje się więc jasne, że język programowania wymaga nie tylko sztywnych reguł określania formy legalnego programu, ale także, równie sztywnych, reguł określania jego znaczenia. Dla niektórych może być zaskoczeniem, że w przypadku wielu nowoczesnych języków programowania satysfakcjonująca semantyka nie została w ogóle wypracowana. I nie mamy na myśli wdrożenia. Niektórzy uważają, że skoro język został zaimplementowany i jego programy faktycznie działają (na przykład ma kompilator, termin, który wyjaśnimy w dalszej części), to jest wystarczająco dobra semantyka. Faktem jest, że potrzebujemy rygorystycznej, niezależnej od maszyny definicji znaczenia każdego programu w języku; taki, który może być wykorzystany do udzielenia jednoznacznych odpowiedzi na każde możliwe pytanie dotyczące tego, co program zrobi w dowolnych okolicznościach, czy zrobi to, co zamierzamy zrobić, i tak dalej. Wszędzie tam, gdzie istnieją adekwatne definicje semantyczne, były one zwykle przygotowywane nie przez projektanta języka czy producenta komputera, ale przez niezależnych badaczy zainteresowanych problemami semantycznymi poszczególnych języków i ich potężnymi cechami. Projektanci i producenci

dostarczają szczegółową dokumentację, która pasuje do języka - tomy. Te podręczniki językowe, jak się je czasami nazywa, zawierają bogactwo informacji dotyczących zarówno składni, jak i semantyki, ale informacje semantyczne zwykle nie wystarczają, aby użytkownik mógł dokładnie zrozumieć, co się stanie w każdym programie zgodnym z prawem składniowym. Problem jest prawdziwy, a dowody na to można znaleźć w głębokiej i zawiłej pracy semantyków języka programowania. Bardzo kuszące jest dodanie nowej i potężnej funkcji do języka oraz określenie, jak sobie z nią radzić w naturalnym kontekście. Jednak to nieprzewidywalność interakcji takiej funkcji ze wszystkimi innymi wspieranymi przez język może spowodować, że sprawy wymkną się spod kontroli.

Podprogramy jako parametry: przykład

Załóżmy, że język obsługujący podprogramy rekurencyjne ma dopuszczać nie tylko zmienne, których wartościami są liczby lub słowa (= ciągi symboli), ale także zmienne specjalne, których wartościami są same nazwy podprogramów. Gdy takie funkcje są dozwolone, powinno być również możliwe używanie zmiennych podprogramów jako parametrów innych procedur. W ten sposób, jeśli podprogram P jest zdefiniowana jako:

podprogram P-z-parametrem-V

gdzie V jest zmienną podprogramu, a gdzieś wewnątrz P znajduje się instrukcja postaci:

wywołaj V

wtedy, jeśli P zostanie wywołane z wartością V będącą procedurą Q, ta instrukcja wpłynie na wywołanie Q wewnątrz P, podczas gdy jeśli P zostanie wywołane, gdy V jest jakąś inną procedurą R, wywoła wywołanie R wewnątrz P. To jest dość potężną funkcją, umożliwiającą zewnętrzną kontrolę nad procedurami, które wywołuje P, po prostu przez zmianę wartości V. Pojawiają się jednak poważne problemy semantyczne. Co jeśli V jest również procedurą, której parametrem jest zmienna podprogramu?

* Aby zaostrzyć pytanie, oto przykład takiego P:

podprogram P-z-parametrem-V

(1) wywołaj V-z-parametrem-V, umieszczając zwróconą wartość w X:

(2) jeśli X = 1 to zwróć z 0; w przeciwnym razie zwróć z 1.

Ale co zrobi nasz zdezorientowany procesor, gdy zostanie poproszony o wykonanie następującego początkowego wywołania P:

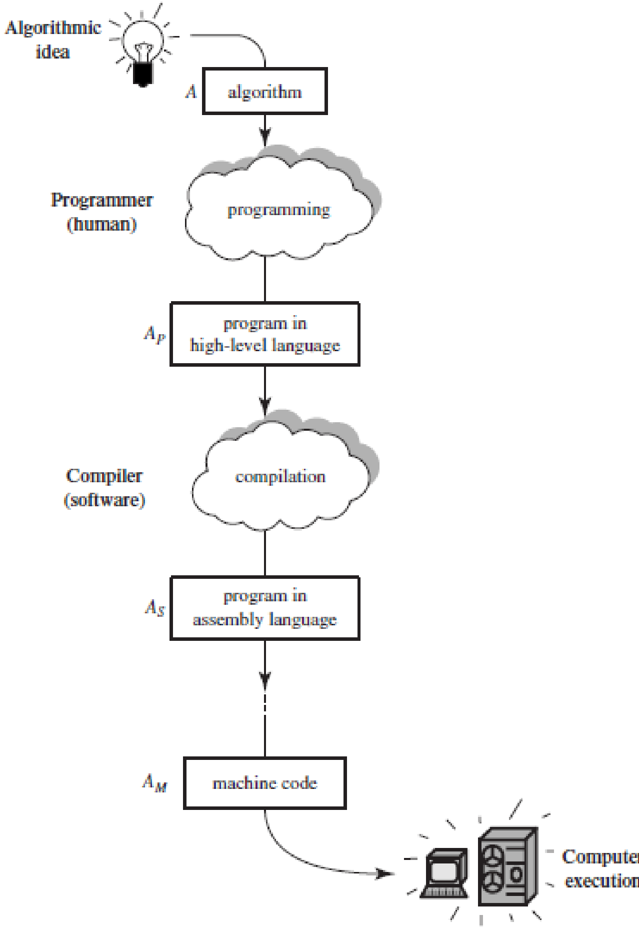
wywołaj P-z-parametrem-P

Syntaktycznie wszystko jest w porządku; wszystkie procedury są wywoływane z właściwą składnią i właściwymi rodzajami parametrów. Niemniej jednak w samym pytaniu zawarty jest paradoks. Niemożliwe jest, aby wywołanie zwróciło 0, ponieważ z definicji ciała P, zwracana wartość wynosi 0, jeśli wywołanie V-z-parametrem-V (co jest teraz tak naprawdę wywołaniem P-z-parametrem-P) zwraca 1. Tak więc wywołanie może zwrócić 0 tylko wtedy, gdy zwróci 1, co jest śmieszne! Podobny argument pokazuje, że wywołanie P-z-parametrem-P również nie może zwrócić 1, ale przez tekst P nie może zwrócić niczego innego niż 0 lub 1. Więc co to robi? Czy to dziwne wezwanie pociąga za sobą niekończący się proces huśtawki? Czy jest to po prostu zabronione przez język? Żadna z tych sugestii nie brzmi całkiem dobrze. W przypadku pierwszego możemy chcieć zobaczyć nieskończoną naturę egzekucji odzwierciedloną wyraźniej w tekście. Druga jest jeszcze gorsza, ponieważ nie jest jasne, jak

scharakteryzować zakazane programy składniowo. Oczywiście formalna semantyka takiego języka musi implikować precyzyjne i jednoznaczne odpowiedzi na takie pytania.

Od języków wysokiego poziomu do manipulacji bitami

A więc oto jesteśmy, z językiem programowania wysokiego poziomu PL, składnią, semantyką i wszystkim, i właśnie zakończyliśmy opracowywanie algorytmu A dla rozwiązania pewnego problemu algorytmicznego. Przechodzimy teraz do programowania algorytmu w języku PL, akcja czasami nazywana kodowaniem A do PL i chcielibyśmy zaprezentować wynikowy program o nazwie AP , naszemu komputerowi. Co się potem dzieje? Odpowiedź brzmi, że w zasadzie istnieją dwie możliwości, w zależności od używanego języka i komputera. Oto pierwszy. Program AP jest wprowadzany do pamięci komputera poprzez wpisanie go z klawiatury, odczytanie go z urządzenia nagrywającego, takiego jak dysk magnetyczny, odebranie go elektronicznym kanałem komunikacyjnym z innego komputera lub w inny sposób. Chociaż te fizyczne nośniki i ich wykorzystanie nie mają tutaj znaczenia, to, co dzieje się dalej, ma dla nas znaczenie. Program AP przechodzi szereg przekształceń, które stopniowo sprowadzają go do poziomu, z którym poradzi sobie komputer. Produktem końcowym tych przekształceń jest program AM na poziomie maszyny (mówi się również, że jest napisany w języku maszynowym), co oznacza, że jego podstawowymi operacjami są te, które komputer „rozumie”, takie jak instrukcje manipulacji bitami. Liczba takich przekształceń w dół różni się w zależności od języka i od jednej maszyny do drugiej, ale zwykle wynosi od dwóch do czterech. Obiekty pośrednie to zazwyczaj legalne programy w coraz bardziej prymitywnych językach programowania. Ich repertuar struktur kontroli i danych - a więc i instrukcji podstawowych - stopniowo staje się coraz bardziej skromny, aż w końcu pozostają tylko najbardziej elementarne możliwości bitowe. To po tym końcowym etapie transformacji komputer może naprawdę uruchomić lub wykonać oryginalny program, a przynajmniej może sprawić wrażenie, że robi właśnie to, prosząc o zestaw legalnych danych wejściowych i uruchamiając przekształconą wersję AM.



Przekształcenia w dół przypominają nieco zastępowanie podprogramów ich ciałami. Instrukcja w języku wysokiego poziomu może być traktowana jako wezwanie do procedury lub jako podstawowa instrukcja, która nie jest wystarczająco podstawowa dla komputera. W konsekwencji musi zostać dopracowany i sprowadzony do poziomu kompetencji komputera.

Kompilacja i języki asemblera

Porozmawiajmy o pierwszej transformacji, zwanej kompilacją, w której wysokopoziomowy program AP jest tłumaczony na program AS w języku niższego poziomu, zwanym asemblerem. Języki asemblera różnią się w zależności od maszyny, ale z reguły używają raczej prostych struktur kontrolnych, które przypominają instrukcje goto i konstrukcje if-then, i zajmują się nie tylko bitami, ale także liczbami całkowitymi i łańcuchami symboli. Mogą odnosić się bezpośrednio do lokalizacji w pamięci komputera - tego dużego obszaru zawierającego akry na akry bitów - mogą odczytywać z tych lokalizacji dowolne liczby i ciągi, które zakodują, i mogą w nich przechowywać kody liczb i ciągów. Typowa konstrukcja pętli wysokiego poziomu, taka jak:

for Y from 1 to N do

(ciało pętli)

end

może być przetłumaczony na odpowiednik języka asemblerowego wyglądający tak (wyjaśnienia w nawiasach nie są częścią programu):

MVC 0, Y (przenieś stałą 0 do lokalizacji Y)

LOOP: CMP N, Y (porównaj wartości w lokalizacjach N i Y)

JEQ REST (jeśli jest równy, przeskocz do instrukcji oznaczonej „REST")

ADC 1, Y (dodaj stałą 1 do wartości w lokalizacji Y)

(przetłumaczone-ciało-pętli)

JMP LOOP (wróć do instrukcji oznaczonej „LOOP")

REST: (reszta programu)

Przed (lub po) samym programie asemblera pojawiałyby się dodatkowe instrukcje kojarzenia symboli Y i N z pewnymi stałymi lokalizacjami pamięci, ale nie będziemy tu wchodzić w te sprawy. Teraz nadchodzi interesujący punkt. Proces kompilacji, który tłumaczy programy komputerowe wysokiego poziomu na język asemblerowy, jest sam wykonywany przez program komputerowy. Ten długi i złożony fragment oprogramowania, zwany kompilatorem, jest zwykle dostarczany przez producenta komputera razem ze sprzętem. Rysunek poiżej ilustruje proces transformacji w dół, którego kompilacja jest pierwszą i najbardziej skomplikowaną częścią. Pozostałe kroki tłumaczą język asemblera na język maszynowy. Są one nieco prostsze niż kompilacje i dlatego nie będziemy się nimi tutaj zajmować. Na marginesie warto zauważyć, że kompilatory dla różnych języków wysokiego poziomu są tylko jednym z wielu programów dostarczanych przez producentów, czasami nazywanych ogólnie oprogramowaniem systemowym. Ich ogólną rolą jest ułatwienie różnych wysokopoziomowych trybów działania komputera, jednocześnie subtelnie izolując użytkownika od wielu szczegółów niskiego poziomu. Jednym z tych trybów jest uruchamianie programów napisanych przez użytkownika, innym jest komunikowanie się z innymi komputerami i łączenie się ze specjalnymi urządzeniami zewnętrznymi.

Interpretery i natychmiastowa egzekucja

Jest jeszcze inny sposób, w jaki komputery mogą wykonywać prezentowane im programy, co nie wymaga tłumaczenia całego programu na język niższego poziomu. Raczej każda z wysokopoziomowych instrukcji programu jest tłumaczona na instrukcje na poziomie maszyny natychmiast po napotkaniu, a te z kolei są natychmiast wykonywane. W pewnym sensie komputer odgrywa rolę robota lub procesora bezpośrednio, biegając i faktycznie wykonując instrukcje wysokiego poziomu jedna po drugiej, dokładnie tak, jak zostały podane. Mechanizm odpowiedzialny za to lokalne tłumaczenie i natychmiastowe wykonanie jest również częścią oprogramowania systemowego, zwykle nazywanego interpreterem. Podejście tłumacza ma pewne oczywiste zalety, wśród których są:

* zwykle łatwiej jest napisać „szybki i brudny", ale dość użyteczny interpreter, niż napisać rozsądny kompilator;

* Wykonywanie sterowane przez interpreter daje bardziej czytelny opis tego, co się dzieje, zwłaszcza podczas interaktywnej pracy z komputerem za pośrednictwem terminala z ekranem.

Istnieje jednak kilka wad interpretacji nad kompilacją, które zostaną pokrótce poruszone w dalszych rozdziałach. To, czy dany komputer skompiluje lub zinterpretuje dany program, zależy od komputera, języka programowania i konkretnego pakietu oprogramowania systemowego w użyciu. Niemniej jednak, podczas gdy niektóre języki programowania poddają się interpretacji łatwiej niż inne, wszystkie języki można w zasadzie skompilować.

Dlaczego nie algorytmiczne esperanto?

Od czasu, gdy w latach pięćdziesiątych zaczęły pojawiać się języki programowania wysokiego poziomu, zaprojektowano ich setki i napisano dla nich kompilatory i/lub interpretery. Wiele z nich wymarło, a prawdopodobnie znacznie więcej powinno wyginąć, ale wciąż istnieją dziesiątki języków w powszechnym użyciu na co dzień. Języki te są w większości dość zróżnicowane pod względem wyglądu i treści. Co więcej, nowe wyrastają jak grzyby po deszczu. Dlaczego? Czy nie byłoby lepiej mieć jeden uniwersalny język, pewnego rodzaju algorytmiczne esperanto, w którym napisane są wszystkie algorytmy i którego każdy może łatwo nauczyć się używać? Dlaczego mnogość języków? Co ich różni od siebie? Kto używa jakich języków i do jakich celów? Aby odpowiedzieć na to pytanie, musimy wrócić do podstawowego celu języków programowania wysokiego poziomu, jakim jest dostarczanie programistom nowych abstrakcji. Istnieją zasadniczo dwie siły, które napędzają potrzebę nowych rodzajów abstrakcji: nowe osiągnięcia technologiczne w sprzęcie oraz nowe i zróżnicowane obszary zastosowań. Dobrym przykładem pierwszego jest rozwój komputerów równoległych. Są to maszyny, które wykorzystują wiele odrębnych, ale połączonych ze sobą „jednotorowych" komputerów. Jedną z konsekwencji tego rozwoju technologicznego jest znaczny wysiłek poświęcony rozwojowi tak zwanych języków programowania współbieżnego, które, w skrócie, obsługują wiele procesorów wykonujących różne czynności jednocześnie. Ponieważ moc komputerów jest wykorzystywana w coraz większej liczbie obszarów zastosowań, programiści napotykają coraz więcej rodzajów abstrakcji: od słów po obrazy, wideo i środowiska wirtualnej rzeczywistości; od ciągów znaków po DNA, fałdowanie białek, a nawet całe procesy biochemiczne; oraz od składni języków naturalnych po rozumienie historii, zdroworozsądkowe rozumowanie i sztuczną inteligencję. Każdy obszar zastosowań ma swój własny zestaw pojęć, które należy włączyć do programów komputerowych. Można to zrobić na wiele sposobów, z których jednym jest tworzenie różnych języków programowania specjalnego przeznaczenia, które zawierają koncepcje określonych obszarów.

Paradygmaty programowania

Wiele istniejących języków programowania można podzielić na kilka rodzin, w oparciu o ich zasady lub paradygmaty organizacyjne. Paradygmat programowania to sposób myślenia o komputerze, wokół którego budowane są inne abstrakcje. Pierwszy i najbardziej rozpowszechniony paradygmat nazywa się programowaniem imperatywnym, a jego pogląd na komputer („model von Neumanna") jest najbliższy samej maszynie. W tym paradygmacie, którego używaliśmy we wszystkich dotychczasowych przykładach, myślimy o komputerze jako o zbiorze komórek pamięci zorganizowanych w wiele rodzajów struktur danych, takich jak tablice, listy i stosy. Programy w tym podejściu zajmują się budowaniem, przechodzeniem i modyfikowaniem tych struktur danych poprzez odczytywanie i modyfikowanie wartości przechowywanych w pamięci. Chociaż ten paradygmat jest bliski prawdziwej architekturze komputera, jest dość daleki od swoich matematycznych początków. Jak wspomniano w Części 2, zmienna matematyczna oznacza pojedynczą wartość. Jest to tylko zmienna w tym sensie, że wyobrażamy sobie różne przypadki tego samego problemu (lub różne wywołania tej samej funkcji), w których może on przyjmować różne wartości. Ale chociaż wartość zmiennej matematycznej nie zmienia się w połowie zadania, dokładnie tak dzieje się w imperatywnym programie komputerowym! Załóżmy, że gdzieś w programie widzisz przypisanie X← 3 . Błędem byłoby zakładać, że gdy napotkasz X ponownie kilka linijek później, będzie on miał wartość 3, chociaż nasz trening matematyczny zachęca nas do takiego założenia. Nawet jeśli nie ma innego przypisania do X w wierszach pośrednich, mogą one zawierać wywołanie podprogramu, które zmienia wartość X. Ogólnie rzecz biorąc, ostrożne i żmudne rozumowanie jest wymagane w celu ustalenia wartości każdej zmiennej w każdym punkcie w programie i jest to jedna z przyczyn złożoności programowania i wynikającej z niej mnóstwa problemów lub „błędów". Inny paradygmat, zwany programowaniem funkcjonalnym, postrzega komputer na wyższym poziomie abstrakcji, jako model matematyczny. Z tego punktu widzenia programy definiują funkcje matematyczne i nie mają pojęcia o pamięci modyfikowalnej. Zamiast

komórek programy te zajmują się tylko wartościami niezmiennymi. Szczegóły, gdzie te wartości są faktycznie przechowywane w komputerze, są ukryte przez abstrakcję funkcjonalną. Programiści przyzwyczajeni do myślenia imperatywnego często czują się skrępowani tą abstrakcją i brakiem kontroli nad pamięcią. Jednak przy użyciu wielu uważa, że jest to wyzwalające, ponieważ mają mniej powodów do zmartwień. Ponadto fakt, że programowanie funkcjonalne zajmuje się matematyką w taki sposób, w jaki jesteśmy do tego przyzwyczajeni, umożliwia zastosowanie szeregu standardowych narzędzi matematycznych do syntezy i analizy programów funkcjonalnych oraz do wnioskowania na ich temat. Z teoretycznego punktu widzenia warto zauważyć, że pojęcie funkcji matematycznej wystarcza do wygenerowania wszystkich możliwych rodzajów obliczeń (w pewnym sensie). Biorąc swój model z logiki, a nie matematyki, paradygmat programowania logicznego paradygmat postrzega komputer jako logiczne urządzenie wnioskowania. Program logiczny jest określony jako zbiór reguł lub prostych instrukcji logicznych postaci, jeśli A, B, ...następnie X. Gdy zapytanie jest wprowadzane do komputera, przeszukuje ten zestaw reguł w poszukiwaniu możliwego dowodu (który może również dostarczyć informacji brakujących w pierwotnym zapytaniu). Taki dowód daje odpowiedź na zapytanie; jeśli nie zostanie znaleziony żaden dowód, uznaje się, że zapytanie nie ma możliwej odpowiedzi. Podobnie jak programy funkcjonalne, programy logiczne nie odwołują się wprost do pamięci komputera. Warto zauważyć, że paradygmaty funkcjonalne i logiczne były bardzo faworyzowane przez badaczy zajmujących się sztuczną inteligencją. Ważnym odgałęzieniem programowania imperatywnego jest dobrze znany paradygmat programowania obiektowego. Głównymi składnikami programu imperatywnego są funkcje (lub podprogramy), które budują i modyfikują struktury danych; funkcje są aktywne, a struktury danych są pasywne. Natomiast paradygmat obiektowy odwraca obraz na bok. Postrzega pamięć komputera jako złożoną z wielu obiektów, odpowiadających strukturom danych widoku imperatywnego. Każdy obiekt ma skojarzony zestaw operacji, które może wykonać, a wykonanie programu składa się z obiektów wysyłających komunikaty żądające od siebie operacji, otrzymywania odpowiedzi i dalszego przetwarzania wyników w celu zaspokojenia własnych wywołujących. W tym ujęciu obiekty są aktywne, a funkcje z widoku imperatywnego zostały zredukowane do komunikatów pasywnych. Ta mentalna sztuczka jest potężnym mechanizmem strukturyzującym. Na przykład, podczas gdy program imperatywny może być zorganizowany jako zbiór podprogramów operujących na różnych strukturach danych, program zorientowany obiektowo jest konstruowany wokół typów danych, zwykle nazywanych klasami. Jedna klasa może reprezentować kolejkę i dostarczać operacji dodawania do niej nowego elementu, usuwania pierwszego elementu, sprawdzania, czy jest pusta i tak dalej. Inna klasa może reprezentować konto bankowe z operacjami takimi jak wypłata, wpłata, uzyskanie bieżącego salda itp. Klasa konta bankowego może używać klasy kolejki do śledzenia transakcji przychodzących. Użyteczność takiego sposobu organizowania programów polega na tym, że w wielu (choć bynajmniej nie wszystkich) przypadkach jest to naturalny sposób myślenia o świecie, który próbujemy modelować wewnątrz komputera. Na przykład możemy mieć dwa obiekty, które reprezentują konta bankowe wymieniające wiadomości w celu modelowania operacji transferu środków. Podobnie przedmiot reprezentujący montowany samochód może wymieniać wiadomości z przedmiotami odpowiadającymi różnym etapom przetwarzania, przez które musi przejść, oraz informacje w oryginalnej kolejności określającej potrzebne akcesoria. Nie powinno dziwić, że paradygmat obiektowy rozwinął się z języków do symulacji procesów w świecie rzeczywistym. Pozostała część krótko opisuje kilka języków programowania, dotykając każdego z tych paradygmatów. Należy zauważyć, że nie podejmuje się żadnych prób nauczania żadnego z tych języków, ani też, jak się twierdzi, leczenie nie zbliża się do ankiety na ich temat. Celem jest raczej przedstawienie małej próbki podstawowych funkcji i przedstawienie pewnego wyobrażenia o tym, jak wyglądają programy i na czym polega ich moc. Wiele innych ważnych języków również zasługuje na wzmiankę, ale różnorodność w formie i zastosowaniach jest bardziej preferowana niż szeroki zakres.

Programowanie imperatywne: Pionierzy

Ponieważ pogląd na paradygmat imperatywny jest najbliższy „nagiej" maszynie, nic dziwnego, że pierwsze języki wysokiego poziomu były imperatywne. Trzy z najwcześniejszych (których rodowody można prześledzić już w późnych latach 50.) to FORTRAN, COBOL i ALGOL. Pierwsze dwa są nadal w użyciu, z powodzeniem ewoluowały, aby sprostać nowym osiągnięciom w sprzęcie i oprogramowaniu, podczas gdy trzeci, choć nieistniejący, był prawdopodobnie najbardziej wpływowy ze wszystkich. FORTRAN (FORMULA TRANSLATOR) był wynikiem pilnej potrzeby obliczeń numerycznych w zastosowaniach naukowych i inżynierskich, takich jak symulacja skutków reakcji jądrowej. Został zaprojektowany z myślą o wydajnej kompilacji, a nie przejrzystości czy czytelności. W rezultacie podstawowa wersja języka nie obsługuje wielu funkcji poprawiających dobrą strukturę programu, które są uważane za ważne w języku wysokiego poziomu. Rozszerzenie języka z 1977 r., FORTRAN 77, w pewnym stopniu zaradza tej sytuacji. FORTRAN obsługuje wektory i tablice wielowymiarowe, ale praktycznie nie obsługuje innych struktur danych. Jeśli chodzi o jego możliwości liczbowe, są one dość potężne i obszerne. Popularność języka w środowisku naukowym i inżynieryjnym zaowocowała wieloma złożonymi funkcjami matematycznymi, które zostały wstępnie zaprogramowane jako stałe podprogramy i które są skatalogowane w różnych bibliotekach funkcji. Takie funkcje, chociaż nie są oryginalną częścią języka FORTRAN, są udostępniane po prostu przez ten sam rodzaj wywołania podprogramu, którego normalnie używa się w tym języku. Może to znacznie rozszerzyć repertuar podstawowych operacji programisty, a dostępność tych bibliotek funkcji odegrała dużą rolę w ciągłej żywotności języka. COBOL (Common Business Oriented Language) jest przeciwieństwem FORTRAN-u w prawie każdym aspekcie. Został zaprojektowany w odpowiedzi na pilną potrzebę języka odpowiedniego do obszernych wymagań przetwarzania danych w bankach, agencjach rządowych i dużych korporacjach. Tak więc, o ile pisanie programów do zarządzania personelem w języku COBOL jest całkiem naturalne, o tyle typowa aplikacja FORTRAN-u, jak numeryczna symulacja reakcji jądrowej, jest beznadziejnie trudna. COBOL został zaprojektowany z myślą o czytelności i pewnych rodzajach przejrzystości, w związku z czym programy nie przypominają lapidarnej i zagadkowej matematyki, a bardziej przypominają wymiany między ludźmi. Ma to oczywiście swoje wady; programy są znacznie dłuższe i „rozwodnione" w języku COBOL, a zatem dość żmudne w pisaniu, a czasami zrozumienie podstawowej struktury programu może być dość trudne. W przeciwieństwie do swoich komercyjnie odnoszących sukcesy kuzynów, które zostały opracowane dla konkretnych zastosowań, z wydajnością najważniejszym kryterium projektowym, ALGOL (ALGOrithmic Language) został zaprojektowany w oparciu o czysto algorytmiczne zasady i zawierał wiele pomysłów wyprzedzających swoje czasy. Duża rodzina języków zwanych zbiorczo potomkami ALGOL obejmuje tak znane języki jak PASCAL i C.

PL/I: język ogólnego przeznaczenia

W 1964 roku IBM ogłosił swój własny język o nazwie PL/I (ambitny akronim od Programming Language ONE). W duchu szkoły „większe jest lepsze" PL/I zebrało wszystkie najlepsze cechy FORTRAN, COBOL i ALGOL w jeden wielki język. W przeciwieństwie do FORTRAN i COBOL, projektanci PL/I położyli nacisk na łatwość programowania, kosztem złożoności implementacji samego języka. W rezultacie PL/I był dostępny tylko na największych komputerach tamtych czasów, komputerach mainframe IBM. (Dzisiaj dostępne są implementacje PL/I dla komputerów PC, które są znacznie silniejsze niż komputery mainframe z lat 60. i 70.) Oto algorytm sortowania pęcherzyków, zakodowany w PL/I:

```
bubblesort: procedure(a);

declare a(∗) binary fixed;

declare i , j , temp binary fixed;
```

```
do i = lbound(a) + 1 to hbound(a);

do j = lbound(a) to i − 1;

if a( j + 1) < a( j ) then

begin

temp = a( j + 1)

a( j + 1) = a( j );

a( j ) = temp;

end;

end;

end;

end;
```

Ta notacja jest dość podobna do tej używanej wcześniej w naszym hipotetycznym języku PL. Zwróć uwagę na niefortunne użycie znaku równości do przypisania, zamiast bardziej odpowiedniej strzałki w lewo (typograficznie reprezentowanej jako „:=” w wielu programach , ze względu na brak strzałki na większości klawiatur). Prowadzi to do potencjalnie mylących stwierdzeń, takich jak x = x + 1, co oznacza „zwiększenie x o 1”, ale wygląda jak równanie matematyczne, które nie ma rozwiązań. Ponadto w tablicach PL/I używa się nawiasów, a nie nawiasów kwadratowych. Konstrukt kontrolny, który napisaliśmy wcześniej dla Y od 1 do N do byłby zapisany w PL/I tak jak Y = 1 do N; Oto program PL/I, który najpierw odczytuje n, liczbę elementów na liście, następnie przydziela wektor o odpowiedniej długości i wczytuje do niego listę, następnie sortuje ją za pomocą podprogramu bubblesort, a na końcu wyświetla posortowany wynik :

```
sort: procedure options(main);

declare n binary fixed;

get list(n);

begin

declare a(n) binary fixed;

declare i binary fixed;

get list ((a(i ), do i = 1 to n));

call bubblesort(a);

put list ((a(i ), do i = 1 to n));

end;

end;
```

Część przetwarzania danych PL/I przyjmuje po COBOL. Główną innowacją w projekcie COBOL jest możliwość definiowania struktury plików, dzięki której możliwe są obiekty przypominające tablice skrzyżowane z drzewami. Oto definicja PL/I prostego pliku uniwersyteckiego zawierającego informacje

istotne dla wyników uczniów i kursów uniwersyteckich. Części obrazu pojawiające się w definicji tekstowej określają format elementów danych: „A” oznacza literę, a „9” cyfrę. Na przykład „(5)A” oznacza pięć liter, a „AAAA999” oznacza cztery litery, po których następują trzy cyfry.

declare 1 UNIVERSITY FILE,

2 STUDENT(100),

3STUDENT NAME picture ‘(15)A’,
3 COURSE(30),

4 COURSE_CODE picture ‘AAAA999’ ,

4 SCORE picture ‘99’ ,

3 STUDENT_ID picture ‘99999’,

2 DEPARTMENT(20),

3 DEP_ NAME picture ‘(10)A’,

3 COURSE(80),

4 COURS_ CODE picture ‘AAAA999’ ,

4 TEACHER picture ‘(10)A’

Odchudzanie

PASCAL, który pojawił się po raz pierwszy w 1970 roku, był po części reakcją na sam rozmiar PL/I. PASCAL został zaprojektowany jako język do nauki programowania i charakteryzuje się raczej elegancją i prostotą niż rozbudowanymi funkcjami. W tym zakresie PASCAL odniósł ogromny sukces i przez wiele lat był używany jako główny język w programie nauczania informatyki w wielu szkołach wyższych i uniwersytetach. Sukces ten nieuchronnie doprowadził do szerokiego wykorzystania PASCAL w zastosowaniach komercyjnych, gdzie jego wady, takie jak skromne możliwości zarządzania pamięcią, stały się oczywiste. To z kolei doprowadziło do rozwoju różnych dialektów PASCAL, które rozszerzają podstawowy język o funkcje niezbędne do zastosowań na dużą skalę. Pod koniec lat 70. język C wyszedł z AT&T Labs i szturmem podbił świat programowania. Oto algorytm sortowania bąbelkowego, tym razem w C:

```
void bubblesort(int *a, int n)

{

int i , j , temp;

for (i = 1; i < n; i++)

for ( j = 0; j < i ; j++)

if (a[ j + 1] < a[ j ])

{

temp = a[ j + 1];

a[ j + 1] = a[ j ];
```

```
a[ j ] = temp;
}
}
```

Pouczające jest porównanie tego programu z odpowiednią implementacją tego samego algorytmu w PL/I pokazanym wcześniej. Oczywiście różnice składniowe są widoczne, ale zignorujemy je i skupimy się na głębszych kwestiach. Podczas gdy procedura PL/I wymagała pojedynczego parametru, tablicy do posortowania, funkcja C musi również otrzymać rozmiar tablicy (n). Tablice w PL/I to obiekty, które oprócz komórek przechowujących zawartość tablicy („pokoje hotelowe") zawierają informacje o dozwolonym zakresie indeksów („numery pokoi"); są one dostępne za pomocą funkcji lbound i hbound. Jednak w C tablica jest tylko wskaźnikiem do pierwszej komórki i nie zawiera żadnych innych informacji. Pierwsza komórka tablicy w C zawsze ma liczbę zero, ale nie ma sposobu, aby stwierdzić, ile komórek zawiera tablica. Ponieważ ta informacja jest niedostępna dla kompilatora, kod maszynowy, który generuje dla wyrażenia a[j], po prostu pobiera adres a (czyli pierwszej komórki tablicy), dodaje do niej j i pobiera wartość przechowywaną w ten adres. (W rzeczywistości to samo wyrażenie można również zapisać w C jako ∗(a + j), co dokładniej odzwierciedla jego implementację.) W przeciwieństwie do tego, gdy kompilator PL/I napotka odpowiednie wyrażenie, wygeneruje również kod do sprawdzenia że indeks rzeczywiście mieści się w granicach prawnych. Jeśli wywołujący funkcję C bubblesort poda błędną wartość dla n, większą niż wartość poprawna, funkcja beztrosko uzyska dostęp i zmodyfikuje wartość zapisaną pod wskazanym adresem, chociaż nie jest ona częścią tablicy. Z dużym prawdopodobieństwem jest to część innej struktury danych, która zostanie błędnie zmodyfikowana. Może to spowodować późniejszą awarię komputera w tajemniczy sposób. To samo stanie się, jeśli sama funkcja bubblesort spróbuje uzyskać dostęp do komórki spoza tablicy (na przykład, jeśli test i < n zostanie omyłkowo zastąpiony przez i <= n). W PL/I pierwszy rodzaj błędu jest niemożliwy, ponieważ wywołujący w ogóle nie podaje argumentu rozmiaru. Drugi rodzaj błędu jest oczywiście możliwy; jednak zostanie on wyłapany znacznie wcześniej, a komunikat o błędzie wskaże programiście procedurę sortowania bąbelków, która zawiera błąd, a nie jakąś inną niewinną procedurę. Powodem takiego zachowania języka C jest to, że pierwotnie miał być używany do programowania systemów; czyli pisanie najbardziej podstawowych programów, bez których komputer jest tylko manipulatorem bitów. Należą do nich system operacyjny, za pośrednictwem którego użytkownicy wchodzą w interakcję z komputerem; sterowniki urządzeń obsługujące urządzenia peryferyjne, takie jak klawiatury, wyświetlacze i drukarki; i kompilatory. W szczególności C został użyty do napisania niezwykle udanego systemu operacyjnego Unix. Programy systemowe często wymagają wyraźnej kontroli zasobów komputera, które zwykle są ukryte przed językami wysokiego poziomu (i nie bez powodu!). C zapewnia taką kontrolę na niskim poziomie, omijając wiele kontroli poprawności wbudowanych w języki wysokiego poziomu. Pozwala to na generowanie wydajnego kodu, który jest często niezbędny w przypadku bardzo często uruchamianych programów systemowych. Pozwala jednak, a nawet zachęca do tworzenia subtelnych błędów, jakie widzieliśmy wcześniej. Co gorsza, takie błędy „przepełnienia bufora" w programach w języku C były wykorzystywane przez hakerów do podważania systemów operacyjnych komputerów i wykorzystywania ich do rozprzestrzeniania wirusów i inne złośliwe oprogramowanie w Internecie. Chociaż z pewnością można pisać wysokiej jakości programy w C, wymaga to więcej wysiłku niż w przypadku niektórych innych języków. Podobnie jak PASCAL, C stał się bardzo popularny i jest używany do prawie każdego rodzaju aplikacji, wykraczając poza pierwotnie zamierzone przeznaczenie. Niestety przyczyniło się to do mnogości oprogramowania niskiej jakości, z którym często się spotykamy. Obecnie imperatywne języki programowania w większości połączyły się z nowszym paradygmatem obiektowym, który omówimy później.

Programowanie funkcjonalne

Jednym z pierwszych języków wysokiego poziomu, którego historia sięga 1958 roku, jest LISP (LISt Processing). W przeciwieństwie do swoich współczesnych, którzy zajmowali się głównie obliczeniami numerycznymi, LISP miał być używany do obliczeń symbolicznych. Bardzo wcześnie w historii informatyki ludzie interesowali się możliwościami komputera jako narzędzia wnioskowania, a nie tylko jako narzędzia liczenia. jeden. Rozumowanie jest naturalnie przeprowadzane za pomocą symboli, a nie liczb, a LISP miał zapewnić łatwe sposoby operowania na symbolach. Jego podstawową strukturą danych jest lista, zapisana jako sekwencja elementów w nawiasach; na przykład (Komputery mają więcej niż 0 i 1). Jednym z wczesnych i najsłynniejszych programów LISP była ELIZA (zwana również Doktorem), która naśladowała psychiatrę prowadzącego dialog z pacjentem. Oto przykład naszego pierwszego algorytmu, algorytmu obliczania pensji. Ten przykład jest napisany w SCHEME, jednym z wielu dialektów LISP-a:

```
(define (sum-salaries employees)

(if (null? employees)

0

(+ (salary (first employees))

(sum-salaries (rest employees)))))
```

Ten kod definiuje funkcję o nazwie sum-salaries z parametrem pracownicy. Jedna z osobliwości LISP-a jest natychmiast widoczna nawet w tym małym przykładzie i jest to użycie nawiasów jako głównej cechy składniowej. Zamiast pisać wyrażenia takie jak a + b, programiści LISP-a piszą(+ a b). Chociaż wymaga to trochę przyzwyczajenia, sprawia, że składnia jest szczególnie prosta, ze względu na jej prostotę i jednolitość: wszystko jest napisane w ten sam sposób i nie ma problemów, takich jak to, które argumenty pasują do której funkcji, w jakiej kolejności. zostać ocenione i tak dalej. Zakładamy, że akta pracowników są podane w formie listy; każdy element listy zawiera imię i nazwisko pracownika (samo jako lista) i wynagrodzenie oraz ewentualnie inne dane osobowe. Oto krótki przykład takiej listy:

```
(((John A. Doe) 85000 (Senior Accountant) Accounting)

((Jane B. Smith) 97000 Manager (Web Services))

((Michael Brown) 70000 Programmer (Systems Support)))
```

Funkcja najpierw zwraca pierwszy element podanej listy, podczas gdy reszta zwraca koniec listy, czyli listę wszystkich elementów z wyjątkiem pierwszego. Funkcja null? sprawdza koniec listy. Teraz łatwo zauważyć, że funkcja sum-wynagrodzeń przechodzi przez listę, element po elemencie, wyciągając pensję bieżącego pracownika i dodając ją do sumy, którą gromadzi. Innym sposobem spojrzenia na powyższy program jest definicja. Faktycznie określa wartość funkcji (suma-pensje pracowników) dla danej listy pracowników, aby była równa 0, jeśli lista jest pusta, a w przeciwnym razie ma być wynagrodzeniem pierwszego pracownika na liście dodanym do sumy wynagrodzeń wszystkich pozostałych pracowników z listy. W przeciwieństwie do sformułowania tego algorytmu w rozdziale 1, ta definicja jest rekurencyjna. W rzeczywistości rekurencja jest centralną i najbardziej naturalną strukturą kontrolną w LISP-ie. Co ciekawe, skoro pensja jest drugim elementem ewidencji pracownika, możemy ją zdefiniować jako pierwszy element reszty listy (co przypomina aforyzm mówiący, że dzisiaj jest pierwszy dzień reszty życia…). .):

```
(define (salary employee)

(first (rest employee)))
```

Teraz jest całkiem prawdopodobne, że będziemy chcieli podsumować wiele rodzajów rzeczy, nie tylko akta pracownicze. W języku funkcjonalnym naturalne jest uogólnienie funkcji sum-wynagrodzeń na rekordy sum, których można użyć do obliczenia sumy dowolnego pola na liście rekordów:

```
(define (sum-records records selector)

(if (null? records)

0

(+ (selector (first records))

(sum-records (rest records)))))
```

Struktura sum-records jest dokładnie taka sama jak sum-salaries, z wyjątkiem tego, że przyjmuje inny parametr, selektor, który sam jest funkcją. Ta funkcja wybiera pole rekordu, który chcemy dodać do sumy. Biorąc pod uwagę tę definicję, sumy wynagrodzeń można teraz zdefiniować prościej jako:

```
(define (sum-salaries employees)

(sum-records employees salary))
```

W rzeczywistości możemy również uogólnić operację akumulacji; oprócz obliczania sum elementów, możemy chcieć obliczyć iloczyny, maksimum elementów i tak dalej. Operacja akumulacji jest określona przez funkcję, która akumuluje nową wartość do sumy bieżącej: dodawanie dla sum, mnożenie dla produktów, i tak dalej. Musimy znać „element jednostki" operacji, czyli wartość, która zostanie użyta, gdy dojdziemy do końca listy; byłoby to 0 dla kwot i 1 dla produktów. To uogólnienie i nowa definicja sum-wynagrodzeń są zapisane w SCHEMIE w następujący sposób:

```
(define (accumulate-records records selector accum unit)

(if (null? records)

unit

(accum (selector (first records))

(accumulate-records records selector accum

unit))))

(define (sum-salaries employees)

(accumulate-records employees salary + 0))
```

Podobnie, jeśli mamy listę ocen uczniów o nazwie cs101, moglibyśmy obliczyć maksymalną ocenę (zakładając, że są one nieujemne) za pomocą następującego wyrażenia (gdzie funkcja max oblicza maksimum z dwóch jej argumentów):

```
(accumulate-records cs101 final-grade max 0)
```

Jak pokazują te przykłady, języki funkcyjne traktują funkcje jak każdy inny typ danych, a w szczególności umożliwiają przekazywanie funkcji jako argumentów do innych funkcji i zwracanie ich jako wyników obliczeń. Ta cecha, albo niedostępna w typowym języku imperatywnym, albo bardzo niewygodna, daje językom funkcjonalnym dużą siłę wyrazu i pozwala na bardzo zwięzłe opisanie wielu algorytmów. Warto zauważyć, że takie traktowanie funkcji i mocy obliczeniowej, jaką niesie, sięga wstecz do rachunku lambda, omówionego dalej w rozdziale 9, który jest formalizmem matematycznym

wynalezionym w latach 30. XX wieku, zanim istniały komputery elektroniczne ogólnego przeznaczenia. Z przykładów jasno wynika, że same programy LISP wyglądają bardzo podobnie do podstawowej struktury danych LISP - listy. W rzeczywistości programy LISP są dokładnie listami, co oznacza, że same mogą być łatwo traktowane jako dane. Dlatego wygodne i naturalne jest pisanie programów LISP, które przetwarzają i/lub generują inne programy. W szczególności, dość łatwo jest napisać interpretery dla LISP-a w LISP-ie. To, w połączeniu z faktem, że wiele osób używających LISP-a ma również duże zainteresowanie projektowaniem języka programowania, spowodowało rozpowszechnienie się wielu różnych dialektów LISP-a. We wczesnych latach 80-tych społeczność LISP-ów podjęła decyzję, że potrzebny jest jeden zunifikowany język, czego rezultatem był nowy język, COMMON LISP. Podobnie jak PL/I, COMMON LISP pochodzi ze szkoły „większe jest lepsze" i zawiera większość dobrych pomysłów z różnych dialektów LISP, które ją poprzedzały. SCHEME, język użyty w powyższych przykładach, można postrzegać jako reakcję „małej i eleganckiej" szkoły na COMMON LISP. (SCHEME jest również nazywany „UnCommon Lisp"). Podobnie jak PASCAL, pierwotnie miał służyć jako narzędzie edukacyjne i dlatego został zaprojektowany tak, aby zawierał kilka silnych i eleganckich funkcji, a nie wszystko, co potencjalnie mogłoby być przydatne w praktyce. Chociaż LISP i SCHEME są oparte na paradygmacie funkcjonalnym i mogą być używane jako języki funkcyjne, zawierają również konstrukcje programowania imperatywnego. W przeciwieństwie do tego istnieją również języki czysto funkcjonalne, takie jak HASKELL i MIRANDA. Te języki nie zawierają żadnych konstrukcji do modyfikowania pamięci; program tworzy nowe elementy danych zamiast modyfikować już istniejące. Oczywiście pamięć komputera jest ograniczona i ostatecznie zostanie wykorzystana przez wszystkie te nowe elementy danych. Jedną z części środowiska uruchomieniowego zapewnianego przez języki czysto funkcjonalne dla ich programów jest garbage collector, który jest odpowiedzialny za automatyczne identyfikowanie elementów danych, które nie są już używane i odzyskiwanie zajmowanej przez nie pamięci. W rezultacie nowy element danych może zajmować tę samą pamięć fizyczną, która była wcześniej używana do czegoś innego. Jednak to ponowne użycie pamięci następuje poniżej poziomu abstrakcji zapewnianego przez język; programista może myśleć w kategoriach elementów danych, które nigdy się nie zmieniają. W wyniku tej abstrakcji semantyka czysto funkcjonalnych języków programowania jest stosunkowo łatwa do formalnego określenia. W praktyce oznacza to, że programiści mogą używać znanych narzędzi matematycznych do wnioskowania o swoich programach. Na przykład najbardziej podstawowe techniki matematyczne zastępowania równych, które nie mają zastosowania w językach imperatywnych, działają zgodnie z oczekiwaniami w językach czysto funkcjonalnych i znacznie upraszczają zadanie pisania poprawnych programów.

Programowanie logiczne

Wszystkie logiczne języki programowania są wariantami pierwszego i najbardziej znanego języka PROLOG (PROgrammation en LOGique), który powstał na początku lat siedemdziesiątych. Program logiczny składa się ze zbioru aksjomatów logicznych, zwanych regułami, które definiują różne właściwości związane z rozwiązywanym problemem. Na przykład program do gry w szachy może zdefiniować prawidłowy ruch predykatu (P, X, Y), co jest prawdziwe, gdy konfiguracja planszy Y jest wynikiem prawidłowego ruchu gracza P w konfiguracji X. Może również użyć wartości predykatu (P, X, N), to prawda, gdy wartość konfiguracji szachownicy X dla gracza P wynosi N. (Dostarczenie dobrej definicji wartości pozycji w grze jest jednym z sekretów dobrego programu do gry w szachy) W prostszym przykładzie rozważmy element predykatu (X, S), który jest prawdziwy, gdy X jest elementem listy S. Wszystkie poniższe twierdzenia powinny być prawdziwe (listy PROLOG są oznaczone nawiasami kwadratowymi):

member(1, [1, 2, 3])

member(b, [a, b, c])

member(apples, [oranges, apples])

Jednak element(2, [a, b, c]) jest fałszywy. Oto definicja tego predykatu jako programu PROLOG:

member(X, [X|Xs]).

member(X, [Y |Ys]) ← member(X, Ys).

Ten program składa się z dwóch zasad. Pierwsza stwierdza, że X jest członkiem każdej listy, której pierwszym elementem jest X. Druga stwierdza, że X jest członkiem listy, której pierwszym elementem jest Y i której ogonem jest lista Ys, jeśli X należy do ogona Ys. (Kierunek strzałki jest bardzo ważny: ta zasada nie oznacza, że aby X był członkiem listy, musi być członkiem ogona; tylko, że jest to jeden z możliwych sposobów, w jaki X może należeć do list.) Mając ten program, interpreter PROLOGa może łatwo udowodnić, że pierwszy zestaw powyższych przykładów jest prawdziwy, podczas gdy member(2, [a, b, c]) jest fałszywy. Jednak może zrobić więcej; na przykład, biorąc pod uwagę element zapytania (X, [a, b, c]), interpreter powie, że to twierdzenie może być prawdziwe (tylko), jeśli X jest jednym z a, b lub c. (PROLOG traktuje nazwy zaczynające się od wielkiej litery jako zmienne, a inne nazwy jako stałe.) Pierwsza reguła programu „członek” jest bezwarunkowa; stwierdza, że jego element docelowy (X, [X|Xs]) jest zawsze prawdziwy. Druga zasada jest warunkowa; mówi, że jednym ze sposobów udowodnienia jego celu, member(X, [Y |Ys]), jest próba udowodnienia member(X, Ys). W obliczu elementu zapytania (X, [a, b, c]), moglibyśmy wypróbować dowolną regułę, aby to udowodnić. Pierwsze dałoby jedno rozwiązanie: X = a. Druga sprowadzałaby problem do udowodnienia elementu (X, [b, c]). Ten nowy problem można rozwiązać za pomocą pierwszej reguły, aby uzyskać X = b, lub sprowadzić do problemu dowodzenia elementu (X, [c]) za pomocą drugiej reguły. To z kolei da trzecie rozwiązanie, X = c, lub zredukuje problem do member(X, []). Jednak na tym się kończymy, ponieważ tego ostatniego problemu nie można rozwiązać za pomocą żadnej z reguł, z których obie wymagają niepustej listy. Nie ma więc dalszych rozwiązań. Tłumacz PROLOG działa w sposób naszkicowany powyżej. Utrzymuje zestaw celów, które stara się udowodnić, i stara się udowodnić każdy z nich po kolei, używając reguł składających się na program. Bezwarunkowa reguła może bezpośrednio udowodnić cel, w którym to przypadku jest on usuwany z listy celów do udowodnienia. Reguła warunkowa może zostać wykorzystana do zredukowania problemu do innych, zastępując cel jednym lub większą liczbą celów podrzędnych, które są następnie dodawane do listy do udowodnienia. Rozwiązanie pierwotnej kwerendy jest znalezione, gdy wszystkie zaległe cele zostały udowodnione (tzn. lista jest pusta). Jeśli jakiś cel nie może być udowodniony przez żadną z reguł, nie ma rozwiązania pierwotnego problemu, czyli powiedzenia, że programu nie można uruchomić do końca. Oto bardziej złożony program PROLOG, rozwiązujący problem Wież Hanoi z Części 2. Predykat hanoi (N, A, B, C, Moves) jest prawdziwy, gdy Moves jest listą ruchów, która rozwiązuje problem przenoszenia N pierścieni z A do B za pomocą C.

hanoi(0, A, B,C, [ ]).

hanoi(N, A, B,C, Moves) ←

N > 0, N1 is N − 1,

hanoi(N1, A,C, B, M1),

hanoi(N1,C, B, A, M2),

append(M1, [move(A, B)|M2], Moves)

Pierwsza zasada jest bezwarunkowa i mówi, że pusta lista ruchów rozwiązuje problem przesuwania pierścieni zerowych. Druga zasada mówi, że ruchy to zestaw ruchów, które zgodnie z prawem

przeniosą dodatnią liczbę N pierścieni z A do B za pomocą C, jeśli można ją rozłożyć na trzy części: początkowa część, M1, to zestaw ruchów, które będą legalnie przenieść N − 1 pierścieni z A do C za pomocą B; następnie pojedynczy ruch, który zdejmuje górny pierścień z A i przenosi go do B; a następnie M2, zbiór ruchów, które legalnie przeniosą N − 1 pierścieni z C do B za pomocą A. Jest to bardzo podobne do naszego oryginalnego sformułowania tego algorytmu w Części 2. Biorąc pod uwagę zapytanie hanoi(3, a, b, c , M), PROLOG odpowie:

M = [move(a, b),move(a, c),move(b, c),move(a, b),

move(c, a),move(c, b),move(a, b)]

Program PROLOG to zbiór logicznych aksjomatów i reguł, które same w sobie nie mają znaczenia obliczeniowego. Istnieje wiele sposobów wykorzystania reguł do udowodnienia danego celu, a interpreter PROLOGa wybiera jeden z nich (jego strategią jest wypróbowanie reguł w podanej kolejności, a także sprawdzenie warunków w regułach w oryginalnej kolejności). Załóżmy, że mamy bazę faktów o pracownikach pewnej firmy. Baza danych zawiera m.in. fakty postaci nadzoruje(X,Y), co oznacza, że X jest bezpośrednim przełożonym Y .Możemy wykorzystać te fakty do zdefiniowania relacji przewyższa(X,Y), co oznacza, że X nadzoruje Y , bezpośrednio lub poprzez sieć menedżerów średniego szczebla. Można to zapisać w PROLOGU w następujący sposób:

outranks(X, Y) ← supervises(X, Y).

outranks(X, Y) ← outranks(X, Z), supervises(Z, Y).

Dla uproszczenia załóżmy, że firma ma tylko trzech pracowników: Huey, dyrektor generalny, nadzoruje Deweya, który z kolei zarządza Louie. Tłumacz PROLOGa z łatwością udowodni, że Huey przewyższa Louiego. Jeśli jednak zmienimy kolejność reguł, nadal będą miały to samo logiczne znaczenie, ale tłumacz będzie miał kłopoty, gdy zostanie poproszony o udowodnienie faktu przewyższającego (huey, louie). Najpierw spróbuje znaleźć część Z, która jest przewyższana przez Huey; można to zrobić za pomocą drugiej reguły (która jest teraz pierwsza), jeśli możemy najpierw znaleźć część Z', która jest przewyższana przez Z. Można to zrobić, znajdując Z'', która jest przewyższana przez Z', a wyszukiwanie jest kontynuowane w nieskończoność w ten sposób, nigdy nie dając użytecznych wyników. W rzeczywistości nawet oryginalny program zawiedzie w ten sam sposób, gdy otrzyma cel przewyższy (louie, huey), zamiast zatrzymać się i powiedzieć, że to jest fałszywe. (Czy widzisz dlaczego?) Rozwiązaniem jest przepisanie drugiej reguły, aby najpierw przetestować dla przełożonych, a następnie dla pracowników przewyższających ranking:

outranks(X, Y) ← supervises(X, Z), outranks(Z, Y)

Ta zależność obliczeń od zachowania interpretera jest niefortunna, ponieważ oznacza, że nie wystarczy, że same reguły są poprawne, a programiści PROLOG-a muszą się również martwić o sposób, w jaki interpreter wypróbowuje reguły. Mogą to robić na różne sposoby, w tym starannie ustalając kolejność zasad i warunków. Jednak potrzeba rozwiązania tych problemów jest słabością języka, który próbuje określić, co należy zrobić, a nie jak to zrobić. PROLOG otrzymał duży impuls dzięki ogłoszeniu Japanese Fifth-Generation Project w 1981 r. Uznano, że ten ambitny projekt może przyczynić się do znacznego pobudzenia najnowocześniejszych badań w dziedzinie informatyki, wykorzystując komputery równoległe jako platformę sprzętową i PROLOG jako główny język programowania. Niestety, choć zaczął się z hukiem, projekt zakończył się czymś w rodzaju kwilenia, a PROLOG jest obecnie używany tylko w niektórych specjalistycznych aplikacjach.

Programowanie obiektowe

Języki zorientowane obiektowo mają swoje początki w języku SIMULA, który został opracowany na początku lat sześćdziesiątych jako język specjalnego przeznaczenia do symulacji (chociaż w rzeczywistości SIMULA stała się językiem programowania ogólnego przeznaczenia). Pierwotnym celem projektantów SIMULI było stworzenie języka, który umożliwiłby łatwe programowanie symulacji zdarzeń dyskretnych, czyli skomputeryzowanych modeli zdarzeń w świecie rzeczywistym, które nie są ciągłe, jak zjawiska fizyczne, ale zachodzą w różnych punktach w czasie. Na przykład symulacja kolejek kasowych w supermarkecie może dostarczyć przydatnych informacji, które można wykorzystać do zarządzania liczbą aktywnych kas kasowych o różnych porach dnia, minimalizując bezczynne liczniki i niedopuszczalnie długie kolejki. Naturalne jest określenie takiego programu poprzez opisanie zachowania każdego obiektu uczestniczącego w symulacji (czasami nazywa się je agentami) oddzielnie. W tym przykładzie występują trzy rodzaje agentów: klienci, kolejki do kasy i kasjerzy. Klienci będą generowani przez program symulacyjny z częstotliwością zależną od pory dnia. Każdy obiekt klienta decyduje następnie o liczbie posiadanych produktów, zgodnie ze statystykami zebranymi z modelowanego rzeczywistego supermarketu. Następnie wybiera kolejkę do wejścia; na przykład, jeśli ma mniej niż 10 pozycji, wybierze kolejkę ekspresową, w przeciwnym razie wybierze kolejkę najkrótszą. (W symulacji obiekt klienta ustala, która kolejka jest najkrótsza, wysyłając komunikaty do kolejek, pytając o liczbę klientów i porównując odpowiedzi.) Następnie obiekt klienta wysyła wiadomość do obiektu reprezentującego wybraną kolejkę, pytając dołączyć do tej kolejki. Gdy obiekt kolejki ustali, że obiekt klienta dotarł na początek kolejki i po otrzymaniu komunikatu od obiektu kasjera, że jest gotowy dla następnego klienta, kolejka powiadamia klienta, że może podejść do kasjera. Klient następnie informuje kasjera o ilości posiadanych pozycji. Charakter symulacji jest taki, że obiekt kasjera nie wykonuje w rzeczywistości żadnej rzeczywistej pracy, ale po prostu decyduje, ile czasu należy czekać (udając, że tak powiem, że dzwoni do pozycji klienta) na podstawie liczby pozycji, które klient posiada, a po upływie tego czasu (symulowanego) powiadamia kolejkę, że jest gotowa na przyjęcie nowego klienta. Taki opis systemu w świecie rzeczywistym pod kątem obiektów, które zawiera i komunikatów, które one wymieniają, jest bardzo naturalny nie tylko dla symulacji zdarzeń dyskretnych, ale także dla wielu innych rodzajów modeli skomputeryzowanych (i, jak się okazuje, nawet dla bardziej abstrakcyjnych pojęć, które nie mają bezpośredniego związku w świecie rzeczywistym). Ten pogląd jest pierwszą podstawą paradygmatu obiektowego, w którym to, co się dzieje (działania) jest podporządkowane temu, co sprawia, że to się dzieje i komu się to dzieje (obiektom) i powinno być skontrastowane z imperatywnym poglądem programowania, w w których dominują procedury, działające na pasywnych strukturach danych. Ten paradygmat prowadzi do stylu programowania, w którym podstawową jednostką, zwaną klasą, jest opis pewnego typu obiektu, wraz z powiązanymi z nim operacjami (lub, równoważnie, komunikatami, które mogą obsłużyć). Kiedy program faktycznie działa, każdy Klasa jest tworzona przez prawdopodobnie wiele obiektów instancji. Druga podstawa paradygmatu obiektowego nazywana jest dziedziczeniem i oznacza zdolność programisty do definiowania relacji inkluzji między klasami. Na przykład klasa krów jest podklasą klasy ssaków, która z kolei jest podklasą klasy wszystkich zwierząt. Lub, aby pozostać nieco bliżej matematyki i obliczeń, klasa kwadratów jest podklasą klasy prostokątów, która jest podklasą wszystkich równoległoboków, która jest podklasą wszystkich czworokątów i tak dalej. Oznacza to, że kwadraty mają wszystkie właściwości prostokątów: wszystkie ich kąty są proste, a przeciwne boki są równe. Kwadraty mają również właściwości, które nie są wspólne dla wszystkich prostokątów: wszystkie ich boki są równe. Z punktu widzenia programisty squares może obsłużyć wszystkie komunikaty zdefiniowane dla prostokątów, takie jak żądania obliczenia obwodu lub obszaru. Ponadto kwadraty mogą obsługiwać własne komunikaty, które nie są wspólne dla prostokątów, takie jak żądanie zwrócenia długości boku kwadratu, która nie jest jednoznacznie zdefiniowana dla prostokątów. W ten sposób klasa squares dziedziczy wszystkie operacje z klasy rectangle, łącznie z kodem, który je implementuje. Prowadzi to do ważnego stylu programowania, w którym niektóre klasy mają tylko częściowy kod lub nawet nie

zawierają żadnego kodu. Takie klasy nazywane są abstrakcyjnymi i oczywiście nie opisują całkowicie zachowania swoich obiektów. Są jednak bardzo przydatne jako wysokie poziomy hierarchii dziedziczenia. Na przykład można zdefiniować klasę abstrakcyjną opisującą kolejki. Taka klasa opisuje, co potrafi kolejka, ale nie opisuje, jak to robi. To, co jest miłe w tym rozróżnieniu, to to, że „co" jest dokładnie informacją, której potrzebują inne klasy, aby korzystać z kolejek; tak naprawdę nie potrzebują „jak". Istnieje wiele sposobów implementacji kolejek, a różnią się one reprezentacją danych i używanymi algorytmami. Wszystkie takie implementacje można zdefiniować jako podklasy abstrakcyjnej klasy kolejki, co gwarantuje, że obsługują wszystkie niezbędne operacje. W ten sposób programy korzystające z klasy kolejki i programy, które ją implementują, mają dobrze zdefiniowany i wąski interfejs, który jest zawarty w abstrakcyjnej klasie kolejki. Ta niezależność pozwala na oddzielny rozwój odpowiednich klas, co ma kluczowe znaczenie dla rozwoju dużych programów. Pierwszym prawdziwie obiektowym językiem programowania był SMALLTALK, opracowany w latach 70. XX wieku. Od tego czasu powstało wiele innych języków obiektowych, z których najbardziej znane to C++ i JAVA. C++ jest oparty na C, z dodanymi funkcjami obiektowymi. W związku z tym cierpi na wyżej wymienione problemy związane z C (ale mimo to jest bardzo popularny). JAVA również bazuje na tradycji C/C++, ale nie przedłużenie jednego z nich; wiele problematycznych funkcji tych języków zostało usuniętych z JAVA (jak również niektóre z bardziej użytecznych...). Oto klasa Queue w JAVA ("interfejs" to termin JAVA na klasę abstrakcyjną, która nie ma implementacji).

```
public interface Queue

{

boolean empty();

Object front();

void add(Object x);

void remove();

}
```

Ta klasa definiuje cztery operacje (nazywane metodami w popularnej terminologii obiektowej): empty, która zwraca wartość logiczną (prawda lub fałsz) w zależności od tego, czy kolejka jest pusta, czy nie; front, który zwraca obiekt na początku kolejki (klasa Object jest szczytem hierarchii dziedziczenia i oznacza wszystkie obiekty w języku); add, który pobiera obiekt x i wstawia go z tyłu kolejki; i usuń, co usuwa obiekt z przodu kolejki. Ta definicja zawiera wszystkie informacje o kolejkach potrzebne pisarzowi klasy Customer, która reprezentuje klientów w naszym przykładzie symulacji supermarketu. Oczywiście do przeprowadzenia symulacji niezbędne jest posiadanie implementacji klasy Queue. Aby go zaimplementować, użyjemy połączonej listy do przechowywania jej elementów; lista zostanie zaimplementowana przy użyciu klasy Linkable, która opisuje w niej pojedynczy link.

```
public class Linkable

{

private Object −item;

private Linkable −next;

public Linkable(Object x)

{
```

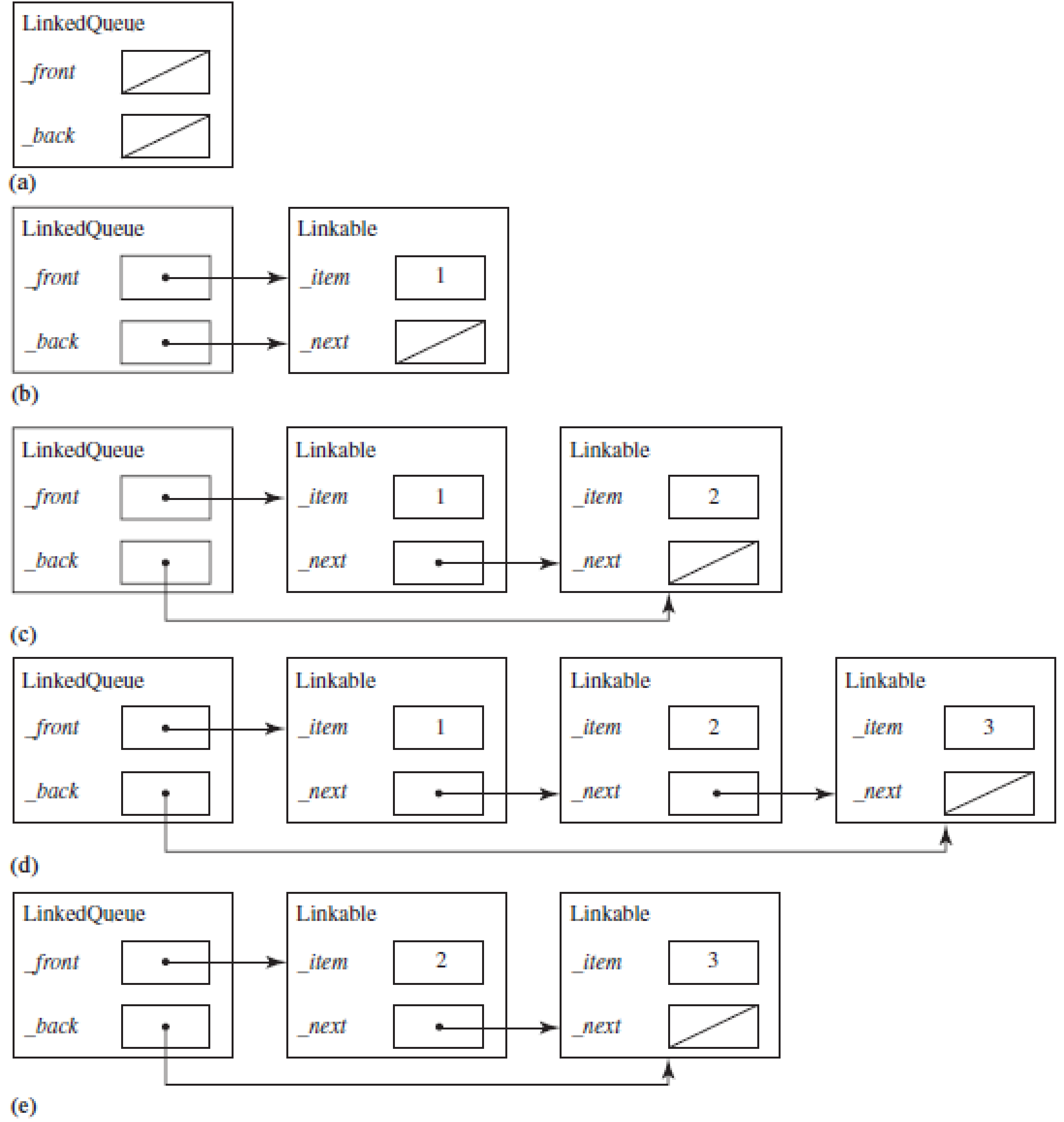
```
−item = x;
−next = null;
}
public Object item()
{
return −item;
}
public Linkable next()
{
return −next;
}
public void set−next(Linkable next)
{
−next = next;
}
}
```

Jest to prosta klasa, która przechowuje tylko dane iw tym sensie jest podobna do struktury danych w programie imperatywnym. Programy zorientowane obiektowo zazwyczaj zawierają klasy z bardziej złożonymi operacjami. Definicja klasy Linkable składa się z dwóch pól, które przechowują stan obiektu, konstruktora (którego nazwa w JAVA jest taka sama jak nazwa klasy), służącego do tworzenia nowych obiektów tej klasy oraz trzech metod. Obiekt przechowywany przez komórkę, którą można połączyć, jest przechowywany w zmiennej o nazwie - item, a odwołanie do następnej komórki znajduje się w zmiennej o nazwie - next. Zmienne te są zadeklarowane jako prywatne, co oznacza, że inne klasy nie mogą ich czytać ani modyfikować. Cały dostęp z zewnątrz do tych zmiennych musi być kierowany przez inne metody klasy Linkable. Zmienna −item jest ustawiana w konstruktorze; klienci mogą tylko odczytać jego wartość (za pomocą metody item), ale nie mogą jej modyfikować. Zmienna - next jest inicjowana do wartości null w konstruktorze i może być odczytywana i modyfikowana przez klientów (przy użyciu metod next i set-next). Oto definicja klasy LinkedQueue, która używa klasy Linkable w celu zaimplementowania interfejsu Queue.

```
public class LinkedQueue implements Queue
{
private Linkable −front = null, −back = null;
public boolean empty()
{
return −front == null;
```

```
}
public Object front()
{
return −front.item();
}
public void add(Object x)
{
Linkable new−back = new Linkable(x);
if (−front == null)
{
−front = new−back;
−back = new−back;
}
else
{
−back.set−next(new−back);
−back = new−back;
}
}
public void remove()
{
−front = −front.next();
}
}
```

Ta klasa ma dwa pola: jedno wskazuje na przód kolejki (która jest pierwszym elementem połączonej listy), a drugie wskazuje na jej tył (ostatni element listy). Kolejka jest uważana za pustą, gdy przedni wskaźnik ma wartość NULL (co oznacza, że nie wskazuje żadnego obiektu). Operacje modyfikujące kolejkę muszą utrzymywać poprawną relację między tymi wskaźnikami. Zdejmowanie przedniego elementu jest proste, wymaga jedynie przesunięcia przedniego wskaźnika na kolejny element. Dodawanie jest bardziej skomplikowane i wymaga rozważenia dwóch przypadków. Jeśli kolejka jest początkowo pusta, dodanie do niej elementu spowoduje utworzenie listy jednego elementu, więc zarówno przedni, jak i tylny wskaźnik powinny wskazywać na ten pojedynczy element. Jeśli kolejka nie jest pusta, dodanie nowego elementu wpływa tylko na wskaźnik wsteczny, który przesuwa się, aby wskazywać na nowo dodany element na końcu listy. Rysunek ilustruje działanie tego programu.



Część (a) pokazuje stan początkowy obiektu, który może zostać wygenerowany przez wyrażenie new LinkedQueue(). Jeśli wstawimy wartości 1, 2 i 3 (metodą add), a następnie usuniemy element przedni (metodą remove), otrzymamy szereg zmian pokazany w częściach (b)–(e). Istnieje wiele innych sposobów na zaimplementowanie interfejsu, takiego jak Queue (nie tylko sposób, w jaki zrobiliśmy to z klasą LinkedQueue). Klient nie musi być świadomy, która konkretna implementacja jest używana; wszystkie informacje potrzebne klientom są dostępne w abstrakcyjnym interfejsie. Ta właściwość, zwana modułowością, jest ważną cechą paradygmatu zorientowanego obiektowo, umożliwiającą, wręcz zachęcającą, rozbijanie złożonych problemów na stosunkowo niezależne mniejsze części i jest kluczowa dla rozwoju systemów wielkoskalowych. Jednym z najciekawszych języków obiektowych jest EIFFEL, który został zaprojektowany w oparciu o solidne podstawy teoretyczne, a jednocześnie jest praktyczny w dużych projektach programistycznych. W rozdziale 5 omówimy unikalną cechę EIFFEL, zwaną projektowaniem według umowy, która jest próbą przeniesienia omówionych w tym rozdziale metod dowodowych na programy o dużej skali.

Badania nad językami programowania

Języki programowania stanowią aktywny i szeroko adresowany obszar badawczy w informatyce, a my byliśmy w stanie dotknąć tylko kilku z związanych z tym zagadnień. Ludzie interesują się językami ogólnego i specjalnego przeznaczenia oraz ich precyzyjną definicją i efektywną implementacją. Opracowywane są wyrafinowane techniki kompilacji i tłumaczenia, proponowane są nowe struktury kontrolne i struktury danych, a także ustanawiane są potężne metody umożliwiające programistom definiowanie własnych. Poszukuje się języków, które zachęcają do „dobrego" stylu programowania, a ludzie starają się projektować je tak, aby zawierały różne zalecane sposoby myślenia o algorytmach. To samo tyczy się języków graficznych, czy, jak je dokładniej nazywamy, formalizmów wizualnych. Jeśli chodzi o paradygmaty, nie ma wątpliwości, że orientacja obiektowa w kilku postaciach cieszy się większym zainteresowaniem niż którakolwiek z pozostałych i jest przedmiotem ogromnej ilości badań i szeroko zakrojonej komercjalizacji. Jeśli chodzi o semantykę, informatycy są zainteresowani dostarczaniem narzędzi i metod tworzenia programów komputerowych o matematycznym znaczeniu.

Jest to konieczne zarówno w przypadku ręcznej kontroli prowadzonej przez człowieka, jak i komputerowej weryfikacji i analizy, jak wyjaśniono w rozdziale 5. Oczywiście dokładna składnia jest warunkiem wstępnym precyzyjnej semantyki, dlatego wyrażenie „semantyka algorytmów" jest nieco bez znaczenia. Dopiero gdy algorytm został zaprogramowany w formalnym języku programowania, można mu nadać formalne i jednoznaczne znaczenie. Istnieje kilka podejść do definicji semantycznych. Jedna, zwana semantyką operacyjną, opisuje znaczenie programu poprzez rygorystyczne zdefiniowanie sekwencji kroków podejmowanych podczas wykonywania oraz wpływ każdego kroku na stan programu. Stan jest jak migawka wszystkiego, co ma znaczenie w programie w danej chwili i zazwyczaj zawiera wartości wszystkich zmiennych, parametrów i struktur danych, a także bieżącą lokalizację sterowania (tj. procesora) w tekście programu. Bardziej abstrakcyjne podejście, zwane semantyką denotacyjną, opisuje znaczenie programu jako czysto matematycznego obiektu, zwykle rodzaju funkcji, która przechwytuje transformację od początkowego stanu początkowego do końcowego stanu końcowego, który pociąga za sobą program. W tym podejściu zwraca się mniej uwagi na faktyczne etapy realizacji programu, a więcej na jego ogólne, zewnętrznie obserwowalne efekty. Bez względu na podejście, uzyskanie odpowiednich definicji nie jest łatwe nawet w przypadku prostych języków, a dla tych zawierających skomplikowane funkcje może stać się naprawdę ogromnym zadaniem. Języki zapytań i manipulacji danymi dla baz danych można w rzeczywistości postrzegać jako języki programowania specjalnego przeznaczenia. Ludzie są zainteresowani tworzeniem tych, które są potężne, elastyczne i wydajne, a w tym konkretnym przypadku zespoły projektowe muszą również być wyczulone na fakt, że wielu użytkowników baz danych nie jest profesjonalnymi programistami, więc języki te powinny być wyjątkowo łatwe w użyciu. Badacze są zainteresowani tworzeniem środowisk programistycznych, a mianowicie przyjaznych dla użytkownika systemów interaktywnych, które umożliwiają programiście pisanie, edytowanie, zmienianie, wykonywanie, analizowanie, poprawianie i symulowanie programów. Niektóre z nich obejmują techniki wizualne (umożliwione przez nowoczesne graficzne terminale komputerowe i stacje robocze), takie jak animacja wykonania programu w połączeniu z obrazami struktur danych w miarę ich zmiany. Jak wspomniano pokrótce, istnieje nowszy kierunek formalizmów wizualnych, w którym te i inne problemy badawcze pojawiają się z pełną mocą, ale z dodatkowymi wymiarami, że tak powiem. (Jak na przykład najlepiej wizualizować rekurencję?) Ostatnia uwaga dotyczy uniwersalności języków programowania. W pewnym sensie technicznym wszystkie omawiane tutaj języki programowania, a także praktycznie wszystkie inne, są równoważne pod względem siły wyrazu. Każdy problem algorytmiczny, który można rozwiązać w jednym języku, można w zasadzie rozwiązać również w dowolnym innym języku. Różnice między językami są pragmatyczne i obejmują adekwatność do określonych zastosowań, przejrzystość i strukturę, wydajność implementacji oraz różne algorytmiczne sposoby myślenia. Biorąc pod uwagę znaczne różnice między językami programowania, może to być zaskoczeniem.

Metody algorytmiczne

Wygląda na to, że możemy teraz szczęśliwie kontynuować nasze algorytmiczne obowiązki. Wiemy, jak zbudowane są algorytmy i jak rozmieścić obiekty, którymi manipulują, a także wiemy, jak zapisać je do wykonania przez komputer. Dzięki temu możemy powiedzieć naszemu procesorowi, co powinien robić i kiedy. Jest to jednak zbyt naiwna ocena sytuacji i w miarę postępów zobaczymy różne przyczyny tego stanu rzeczy. Jeden z problemów polega na tym, że nie udostępniliśmy żadnych metod, które można wykorzystać do opracowania algorytmu. Bardzo dobrze mówi się o konstrukcjach, których algorytm może używać – to znaczy o kawałkach, z których może się składać – ale musimy powiedzieć coś więcej o sposobach wykorzystania tych kawałków do stworzenia całości. W tej części dokonamy przeglądu szeregu dość ogólnych metod algorytmicznych, które projektant może zastosować w celu znalezienia rozwiązania problemu algorytmicznego. Trzeba jednak zaznaczyć, że nie ma dobrych przepisów na wymyślanie receptur. Każdy problem algorytmiczny jest wyzwaniem dla projektanta algorytmu. Niektóre problemy są proste, inne skomplikowane, inne kuszące; niniejszy rozdział pokazuje tylko, że pewne algorytmy całkiem dobrze podążają za pewnymi ogólnymi paradygmatami. Morał, że projektant algorytmu może odnieść korzyść z przyjrzenia się im najpierw, próbując sprawdzić, czy można je wykorzystać lub przystosować do użycia w danej sytuacji. Ogólnie rzecz biorąc, projektowanie algorytmiczne to twórcza działalność, która może wymagać prawdziwej pomysłowości, ale która z pewnością może skorzystać na opanowaniu dostępnego zestawu technik i metod.

Wyszukiwania i przechodzenia

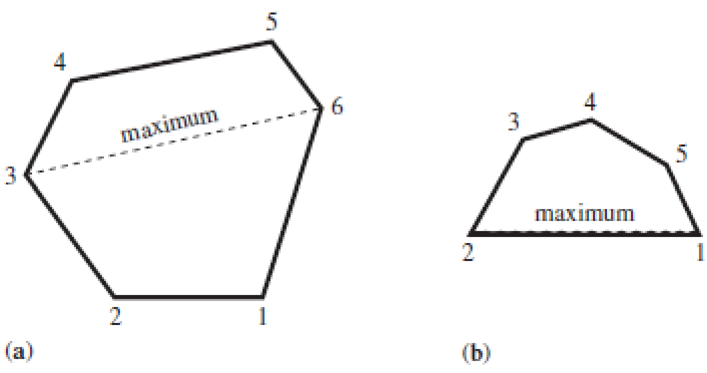
Wiele problemów algorytmicznych rodzi potrzebę przemierzania pewnych struktur. Czasami struktura, którą musimy przejść, jest wyraźnie obecna jako jedna ze struktur danych zdefiniowanych w algorytmie, ale czasami jest to jakaś niejawna abstrakcyjna struktura, której być może nie można faktycznie „zobaczyć", ale która istnieje pod powierzchnią. Czasami szuka się czegoś specjalnego w strukturze („kto jest dyrektorem działu public relations?"), a czasami trzeba wykonać jakąś pracę w każdym punkcie („oblicz średnie oceny wszystkich uczniów") . Na przykład prosty problem sumowania wynagrodzeń, o którym mowa w Części 1, wymaga prostego przejrzenia podanej listy pracowników. Z drugiej strony problem, który dotyczył tylko pracowników zarabiających więcej niż ich menedżerowie, można uznać za wymagający przechodzenia przez wyimaginowaną dwuwymiarową tablicę, w której pracownicy są wykreślani względem pracowników, a wyszukiwanie dotyczy określonego pracownika. /manager par. W takich przypadkach zadaniem jest znalezienie najbardziej naturalnego sposobu przechodzenia przez strukturę danych (czy to jawną, czy niejawną), a tym samym opracowanie algorytmu. Gdy zaangażowane są wektory lub tablice, zwykle pojawiają się pętle i pętle zagnieżdżone, jak wyjaśniono w Części 2, i w tym samym duchu, gdy zaangażowane są drzewa, pojawia się rekurencja, jak to miało miejsce w przykładzie sortowania drzew. Prawdą jest, że idea algorytmu sortowania drzew jest dość subtelna i nie można jej znaleźć po prostu wymyślając najlepszą strukturę kontrolną do przechodzenia przez daną strukturę danych. Jednak gdy już wpadnie na pomysł, drobiazgiem jest uświadomienie sobie, że druga wizyta, w której elementy są wyprowadzane w kolejności, jest niczym innym jak pewnym przechodzeniem drzewa binarnego, z którego nie jest zbyt trudno dojść do wniosku, że należy stosować rekurencję. Przechodzenie wywołane rekurencyjnym przejściem w jest czasami nazywane przeszukiwaniem w głąb ze śledzeniem wstecznym, ponieważ procesor „nurkuje" w drzewo, próbując zejść jak najgłębiej, a kiedy nie może przejść dalej, cofa się niechętnie, zawsze dążąc do wznowienia nurkowania. Jedyną dodatkową cechą jest tutaj wymóg, aby nurkowanie odbywało się maksymalnie w lewo. Istnieje kilka innych sposobów przechodzenia przez drzewa, z których jeden, podwójny do wyszukiwania w głąb, nazywa się wyszukiwaniem wszerz. Trawersowanie w kierunku wszerz oznacza, że poziomy drzewa są kolejno wyczerpane; najpierw korzeń, potem całe jego potomstwo, potem ich potomstwo i tak dalej.

Wyczerpujące poszukiwania, czyli procedura British Museum

Kiedy potrzebujesz znaleźć coś w strukturze danych, możesz po prostu zbadać wszystkie jej elementy jeden po drugim, aż znajdziesz to, czego szukasz. To właśnie zrobiliśmy, gdy szukaliśmy pracowników zarabiających więcej niż ich menedżerowie. Ten prosty pomysł nazywa się wyszukiwaniem wyczerpującym lub, bardziej barwnie, procedurą British Museum. Ten ostatni termin odnosi się do sposobu, w jaki poważnie myśląca osoba przegląda każdy eksponat w muzeum. Wyszukiwanie wyczerpujące jest czasem jedynym sposobem rozwiązania problemu algorytmicznego. Jednak często są znacznie lepsze sposoby. Na przykład, jeśli chciałbyś znaleźć numer w książce telefonicznej za pomocą wyszukiwania wyczerpującego, musiałbyś przejrzeć każde nazwisko w książce, aż znajdziesz to, którego szukasz (lub odkryjesz, że go tam nie ma). To może zająć dużo czasu. Zamiast tego szacujesz z grubsza, gdzie otworzyć książkę, na podstawie pierwszej litery imienia, a następnie szybko korygujesz swoje oszacowanie na podstawie nazwisk na otwieranej stronie i znajdujesz numer, którego szukasz w mniej niż minutę. Ta tak zwana procedura wyszukiwania interpolacyjnego, która jest znacznie bardziej wydajna niż wyczerpujące wyszukiwanie tego problemu, jest zorientowaną na człowieka wersją algorytmu. Dlatego w wielu przypadkach wyczerpujące poszukiwania nie są co najmniej najlepszym sposobem. Mimo to jest to przydatna procedura w przypadkach, w których nie ma innych rozwiązań, a także służy jako punkt odniesienia, z którym można porównać inne rozwiązania.

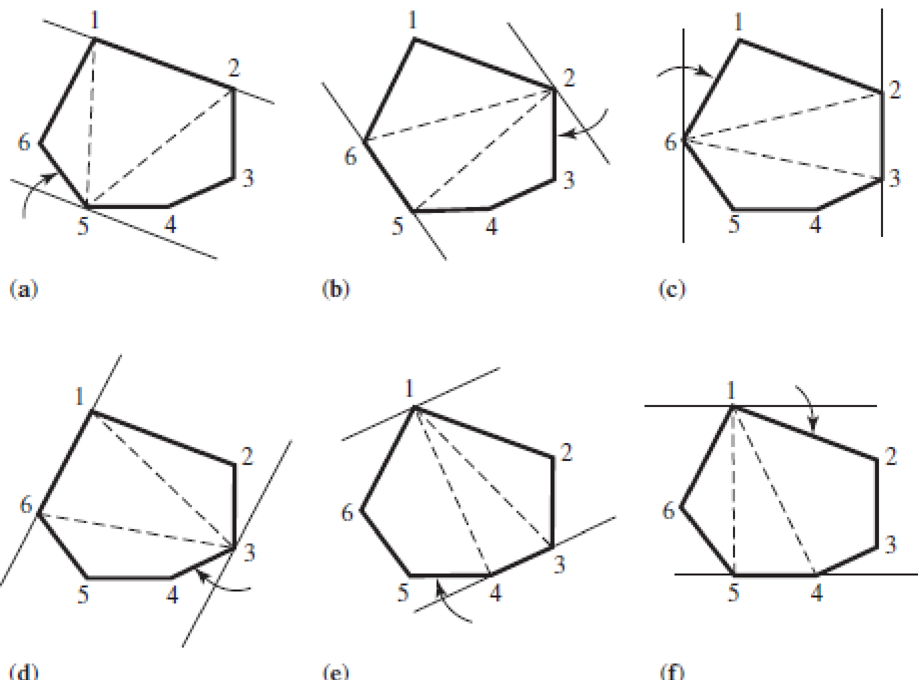
Maksymalna odległość wielokąta: przykład

Wiele interesujących problemów algorytmicznych wiąże się z pojęciami geometrycznymi, takimi jak punkty, linie i odległości, a zatem jest częścią przedmiotu znanego jako geometria obliczeniowa. Wiele problemów z tego obszaru jest zwodniczo łatwych do „rozwiązania” za pomocą ludzkiego układu wzrokowego, ale często prawdziwym wyzwaniem dla projektantów algorytmów. Oto bardzo prosty. Załóżmy, że otrzymaliśmy prosty wielokąt wypukły1, taki jak na rysunku



, i załóżmy, że interesuje nas wskazanie dwóch punktów maksymalnej odległości na jego granicy. Zakłada się, że wielokąt jest reprezentowany przez sekwencję współrzędnych wierzchołków, w kolejności zgodnej z ruchem wskazówek zegara. Ponieważ maksymalna odległość wyraźnie wystąpi dla dwóch wierzchołków (dlaczego?), nie ma potrzeby patrzenia na żadne punkty wzdłuż krawędzi wielokąta inne niż wierzchołki. Trywialne rozwiązanie polegałoby na rozważeniu wszystkich par wierzchołków w pewnej kolejności, utrzymywaniu bieżącego maksimum i pary osiągającej to maksimum w zmiennych. Każda nowa rozważana para jest poddawana prostemu obliczaniu odległości; nowa odległość jest porównywana z bieżącym maksimum, a zmienne są aktualizowane, jeśli nowa odległość okaże się większa, co oznacza, że w rzeczywistości jest to nowe maksimum. Jeśli przez chwilę zastanowimy się nad tym rozwiązaniem, stanie się jasne, że w rzeczywistości przemierzamy wyimaginowaną tablicę, w której wierzchołki są wykreślane względem wierzchołków. Wyimaginowany

element danych w punkcie <I, J > tej tablicy to odległość między wierzchołkami I i J . Przechodzenie można wtedy łatwo przeprowadzić za pomocą dwóch zagnieżdżonych pętli i rzeczywiście zachęcamy do zapisania wynikowego algorytmu. To rozwiązanie uwzględnia jednak zbyt wiele potencjalnych par. Czy nie powinno być możliwe uwzględnienie tylko „przeciwnych” par punktów, takich jak <1, 4>, <2, 5> i <3, > na rysunku 1(a)? Oznaczałoby to przechodzenie tylko przez wektor „specjalnych” par, a nie przez tablicę wszystkich par, co wyraźnie dałoby bardziej wydajny algorytm, który wymaga tylko jednej pętli. To nie jest tak proste, jak się wydaje, ponieważ pożądane przeciwne pary niekoniecznie są tymi, które mają równą liczbę wierzchołków po obu stronach; Rysunek 1(b) pokazuje wielokąt, w którym maksimum występuje w sąsiednich wierzchołkach, co zostałoby pominięte przez algorytm uwzględniający tylko przeciwnie ponumerowane. Lepsze rozwiązanie, które faktycznie wykorzystuje pojedynczą pętlę i uwzględnia tylko „właściwy” rodzaj przeciwstawnych par, pokazano na rysunku 2



Opiszmy, jak to działa. Zrobimy to za pomocą nieformalnych pojęć geometrycznych, chociaż szczegółowy opis algorytmu musiałby przełożyć je na manipulacje numeryczne za pomocą struktur danych i struktur kontrolnych, których tutaj nie będziemy przeprowadzać. Najpierw rysowana jest linia wzdłuż krawędzi między wierzchołkami 1 i 2. Następnie linia równoległa do niej jest stopniowo przesuwana w kierunku wielokąta spoza wielokąta po przeciwnej stronie pierwszej linii, aż do trafienia w jeden z wierzchołków; patrz Rysunek 2(a), z którego jasno wynika, że wierzchołek 5 jest pierwszym, do którego należy w ten sposób dotrzeć. Początkowe przybliżenie do maksimum jest teraz uważane za większą odległość między tym wierzchołkiem (w tym przypadku 5) a wierzchołkami 1 i 2. Następnie rozpoczyna się ruch zgodnie z ruchem wskazówek zegara, którego każdy krok obejmuje:

1. obracanie jednej z tych dwóch linii, aż znajdzie się wzdłuż następnej krawędzi wielokąta w kolejności zgodnej z ruchem wskazówek zegara (na rysunku 2(b) można zobaczyć, że dolna linia obraca się, aby dopasować się do krawędzi od 5 do 6); oraz

2. ustawienie drugiej linii, aby była do niej równoległa (na rysunku 2(b) górna linia jest regulowana).

Dokładnie, która z linii jest obrócona, a która jest dostosowana, określa się przez porównanie wysiłków potrzebnych do obrotu: linia o mniejszym kącie do następnej krawędzi to ta obrócona (na rysunku 2(a)

kąt między dolną linią a krawędź od 5 do jest mniejsza niż między linią górną a krawędzią od 2 do 3). Po zakończeniu rotacji na właśnie obróconej linii pojawia się nowy wierzchołek, którego nie było przed obrotem. Odległość między tym nowym wierzchołkiem a tym na dopasowanej linii jest obliczana i porównywana z bieżącym maksimum, jak poprzednio. Ta procedura jest wykonywana w pełnym okręgu wokół całego wielokąta. Po zakończeniu procedury aktualne maksimum jest żądaną maksymalną odległością. Można wykazać, że wszystkie działania wymagane przez ten algorytm polegają na prostych manipulacjach numerycznych na współrzędnych wierzchołków, które wynikają z elementarnej geometrii analitycznej. Rysunek 2 ilustruje kolejność przekształceń na liniach. Ten przykład został wybrany, aby dodatkowo zilustrować, że rozpoznanie potrzeby przechodzenia i ustalenie, co tak naprawdę ma zostać przebyte, jest ważne i może być bardzo pomocne, ale nie zawsze wystarcza, jeśli chodzi o rozwiązywanie skomplikowanych problemów algorytmicznych; pewien wgląd i duża wiedza na dany temat nie może zaszkodzić.

Dziel i rządź

Często problem można rozwiązać, sprowadzając go do mniejszych problemów tego samego rodzaju i rozwiązując je, a następnie, przy dodatkowej pracy, łącząc częściowe rozwiązania w celu uzyskania ostatecznego rozwiązania pierwotnego problemu. Jeśli mniejsze problemy są dokładnie tym samym problemem, który mamy pod ręką, ale odnoszą się do „mniejszych" lub „prostszych" danych wejściowych, wówczas w algorytmie można zastosować rekurencję. Ta metoda z oczywistych powodów nazywa się dziel i zwyciężaj. Kilka algorytmów w tej książce urzeczywistnia tę ideę „podziel i uderz". Widzieliśmy to już pośrednio w przykładzie z Wież Hanoi: algorytm rozwiązał problem dla pierścieni N, rozwiązując dwa zadania dla pierścieni N−1 we właściwej kolejności i o odpowiednich parametrach. Inne przypadki pojawiają się później. Oto dwa dodatkowe zastosowania dzielenia i zdobywania. Wyobraź sobie, że dostajesz pomieszaną książkę telefoniczną lub, żeby zabrzmieć głębiej, nieuporządkowaną listę L. Nie jesteśmy zainteresowani sortowaniem L, ale jedynie znajdowaniem największych i najmniejszych elementów, które się w niej pojawiają. Oczywiście możemy po prostu przejrzeć listę raz, zachowując bieżące minimum i bieżące maksimum w zmiennych, porównując każdy element z obydwoma w miarę postępu i aktualizując je, jeśli rozważany element jest mniejszy niż bieżące minimum lub większy niż bieżące maksimum. Jednak poniższy algorytm w prosty sposób wykorzystuje strategię dziel i zwyciężaj i, jak wyjaśniono w rozdziale 6, jest w rzeczywistości nieco lepszy. Schematycznie brzmi:

(1) jeśli L składa się z jednego elementu, to ten element jest traktowany zarówno jako minimum, jak i maksimum; jeśli składa się z dwóch elementów, to mniejszy jest przyjmowany za minimum, a większy za maksimum;

(2) w przeciwnym razie wykonaj następujące czynności:

(2.1) podzielić L na dwie połowy, Lleft i Lright;

(2.2) znajdź ich ekstremalne elementy MINleft, MAXleft, MINright i MAXright;

(2.3) wybierz mniejszy z MINleft i MINright; jest to minimalny element L;

(2.4) wybierz większy z MAXleft i MAXright; jest to maksymalny element L.

(Oczywiście podział w linii (2.1) należy zdefiniować w taki sposób, aby objąć przypadek listy L o nieparzystej długości, powiedzmy, przyjmując, że pierwsza połowa jest dłuższa od drugiej o jeden element.) Teraz, wiersz (2.2) błaga o wykonanie rekurencyjnie, ponieważ problemy do rozwiązania to właśnie problem min&max na mniejszych listach Lleft i Lright. Ta rekurencja nie jest tak prosta, jak się wydaje, ponieważ tutaj, w przeciwieństwie do procedury Wieże Hanoi, wywołanie rekurencyjne musi

dawać wyniki, które są używane w sequelu. W jakiś sposób procesor musi nie tylko zapamiętać swój adres zwrotny i sposób przywrócenia środowiska do stanu sprzed rozpoczęcia obecnego rekursywnego przedsięwzięcia, ale musi także być w stanie przywrócić pewne elementy ze swoich trudów. W takim przypadku najbardziej pomocne byłoby, gdyby procesor mógł wrócić z wywołania rekurencyjnego wraz z minimum i maksimum, które zostało wysłane do obliczeń. Poniższe jest wynikiem odpowiedniego rozszerzenia pojęcia podprogramu i zastosowania go do problemu min&max: subroute find-min&max-of L:

(1) jeśli L składa się z jednego elementu, to ustaw dla niego MIN i MAX; jeśli składa się z dwóch elementów, to ustaw MIN na mniejszy z nich i MAX na większy;

(2) w przeciwnym razie wykonaj następujące czynności:

(2.1) podzielić L na dwie połowy, Lleft i Lright;

(2.2) wywołaj find-min&max-of Lleft, umieszczając zwrócone wartości w MINleft i MAXleft;

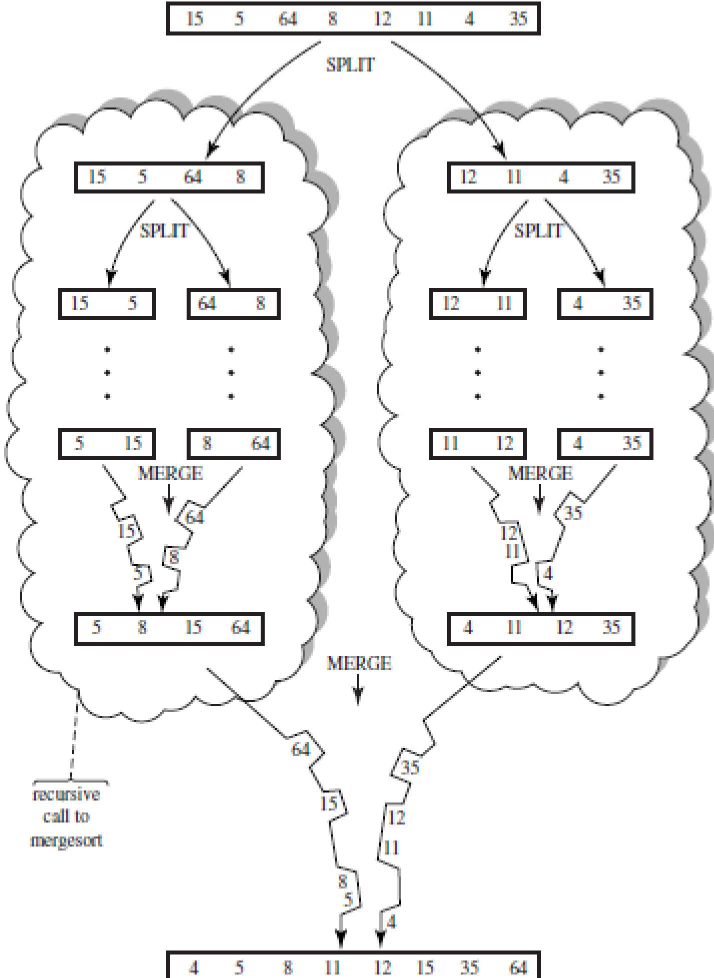
(2.3) wywołaj find- min&max-of Lright, umieszczając zwrócone wartości w MINright i MAXright;

(2.4) ustaw MIN na mniejsze z MINleft i MINright;

(2.5) ustaw MAX na większy z MAXleft i MAXright;

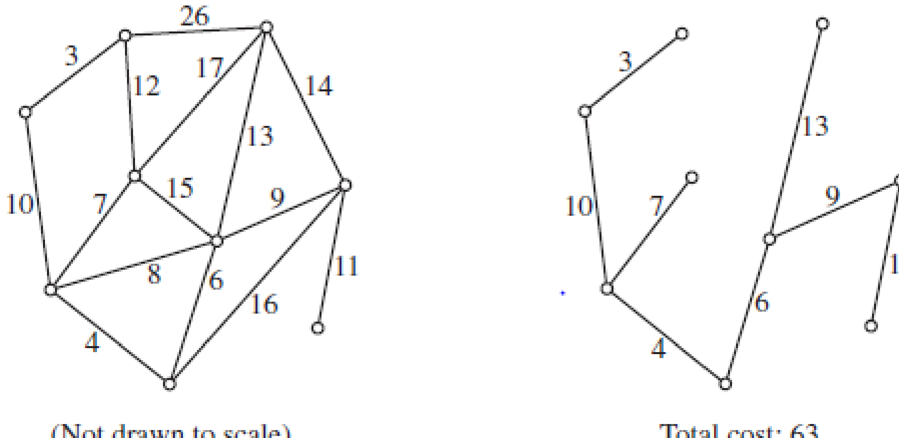
(3) powrót z MIN i MAX.

Paradygmat dziel i zwyciężaj może być z pożytkiem wykorzystany do sortowania listy, a nie tylko do znajdowania jej skrajnych elementów. Oto jak. Aby posortować listę L zawierającą co najmniej dwa elementy, podobnie dzielimy ją na połówki, Lleft i Lright, i sortujemy rekurencyjnie oba. Przypadek jednoelementowy jest traktowany oddzielnie, jak w przykładzie min&max. Aby uzyskać ostateczną posortowaną wersję L, łączymy posortowane połówki w jedną posortowaną listę. Aby scalić dwie posortowane listy, wielokrotnie usuwamy i wysyłamy do wyjścia mniejszy z dwóch elementów znajdujących się obecnie na początku dwóch list. Działanie tego algorytmu, zwanego sortowaniem przez scalanie, zilustrowano na rysunku i zachęcamy do szczegółowego zapisania algorytmu.

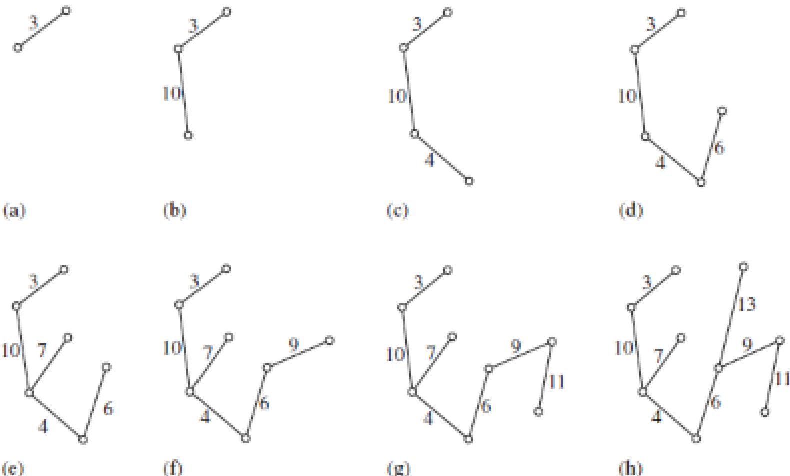


Chciwe algorytmy i wykonawcy kolei

Wiele problemów algorytmicznych wymaga uzyskania jakiegoś najlepszego wyniku z odpowiedniego zestawu możliwości. Weźmy pod uwagę sieć miast i leniwego wykonawcę kolei. Kontrahentowi zapłacono za ułożenie torów tak, aby można było dojechać do każdego miasta z każdego innego. Umowa nie określała jednak żadnych kryteriów, takich jak konieczność zapewnienia niektórych połączeń kolejowych bez międzylądowań, czy maksymalna liczba dozwolonych miast na ścieżce łączącej dowolne dwa inne. Dlatego nasz kontrahent, będąc leniwym, jest zainteresowany ułożeniem najtańszej (czyli najkrótszej) kombinacji odcinków szyny. Załóżmy, że nie wszystkie miasta można połączyć bezpośrednimi odcinkami kolei ze wszystkimi innymi z przyczyn obiektywnych, takich jak przeszkody fizyczne, oraz że odległości są podane tylko między tymi parami miast, które można połączyć. Zakładamy ponadto, że koszt bezpośredniego połączenia miasta A z B jest proporcjonalny do odległości między nimi. Nie dopuszczamy również węzłów kolejowych poza miastami. Taka sieć nazywana jest grafem oznaczonym lub w skrócie grafem. Wykresy są podobne do drzew, z tym wyjątkiem, że drzewa nie mogą się „zamykać"; oznacza to, że nie mogą zawierać cykli ani pętli, podczas gdy wykresy mogą. Rysunek przedstawia przykład grafu miasta i jego minimalnej linii kolejowej.



Zauważ, że wykonawca naprawdę dąży do tego, co czasami nazywamy minimalnym drzewem opinającym. Jest to drzewo, które „rozpina” graf w tym sensie, że dociera do każdego z jego węzłów (czyli w naszym przypadku miast) i jest najtańszym takim drzewem w tym sensie, że suma etykiety wzdłuż krawędzi (czyli w naszym przypadku odległości między miastami) są najmniejsze z możliwych. Łatwo zauważyć, że pożądanym rozwiązaniem musi być drzewo (to znaczy nie może zawierać cykli), ponieważ gdyby miało zawierać jakiś cykl, leniwy wykonawca mógłby uzyskać tańszą linię kolejową, która nadal łączyła wszystkie miasta, eliminując jeden z segmentów tego cyklu. Istnieje algorytmiczne podejście do takich problemów, zwane metodą zachłanną. Zaleca konstruowanie minimalnej krawędzi drzewa rozpinającego po krawędzi, wybierając jako następną krawędź najtańszą z możliwych w obecnej sytuacji. To tak, jakby przyjąć postawę „jedz i pij, bo jutro umrzemy”: rób tyle, ile możesz teraz, bo inaczej możesz żałować, że tego nie zrobiłeś. Rysunek ilustruje budowę takiego drzewa.

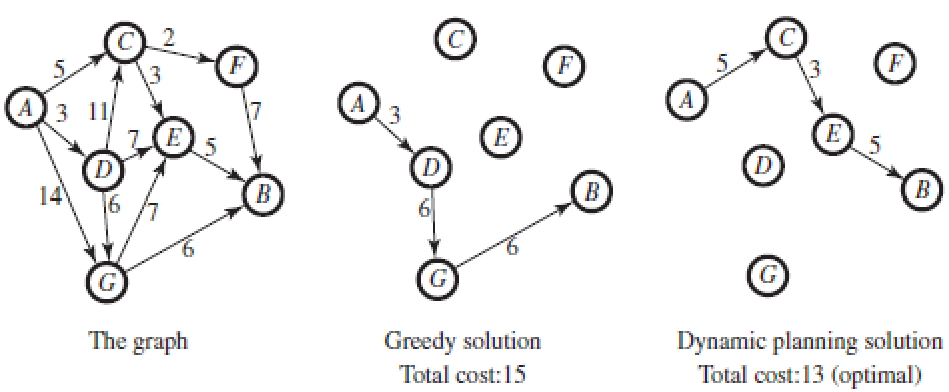


Zacznij od zdegenerowanego drzewa składającego się z najtańszej krawędzi na wykresie, zupełnie samej. Teraz na każdym etapie rozszerz zbudowane do tej pory drzewo, dodając najtańszą krawędź, która nie została jeszcze uwzględniona, o ile spowoduje to powstanie połączonej struktury, która w rzeczywistości jest drzewem. W szczególności nie powinna wprowadzać cyklu; jeśli tak, przejdź do następnej najtańszej krawędzi. Na przykład przejście z rysunku (e) do (f) polega na dodaniu krawędzi oznaczonej 9 zamiast krawędzi oznaczonej 8. Ta ostatnia wprowadziłaby cykl na wykresie. Można wykazać, że ta prosta strategia faktycznie tworzy minimalne drzewo opinające, i zachęcamy do szczegółowego zapisania algorytmu. Algorytmy zachłanne istnieją dla różnych interesujących problemów algorytmicznych. Zwykle są one dość łatwe do zdobycia, niektórych przypadkach są dość

intuicyjne. Najtrudniejszą częścią jest zwykle pokazanie, że strategia chciwości rzeczywiście daje najlepsze rozwiązanie, a jak pokazuje następna sekcja, są przypadki, w których chciwość w ogóle nie popłaca.

Dynamiczne programowanie i zmęczeni podróżnicy

Oto problem, który jest podobny do problemu z minimalnym drzewem opinającym, ale ten przeciwstawia się naiwnym zachłannym rozwiązaniom. To również obejmuje sieć miejską, ale zamiast leniwego wykonawcy kolei mamy zmęczonego podróżnika, który jest zainteresowany podróżowaniem z jednego miasta do drugiego. Chociaż obaj mają zadanie do wykonania i obaj chcą zminimalizować całkowity koszt jej wykonania, istnieje zasadnicza różnica: podczas gdy wykonawca musi połączyć wszystkie miasta za pomocą podsieci szyn, podróżny zazwyczaj podróżuje tylko przez niektóre z miasta. Jest zatem jasne, że podróżnik nie szuka minimalnego drzewa opinającego, ale minimalną ścieżkę; czyli najtańsza podróż prowadząca z miasta startowego do wybranego celu. Dla ułatwienia prezentacji zakładamy, że wszystkie linie na wykresie miasta są skierowane, co oznacza, że jeśli dwa miasta są połączone linią na wykresie, to ta linia reprezentuje połączenie w jedną stronę. Zakładamy również, że graf jest połączony, co oznacza, że graf nie składa się z oddzielnych, rozłącznych części. Dalej zakładamy, że graf miasta nie ma cykli, więc naprawdę zmęczony podróżnik nie będzie podatny na kręcenie się w kółko, ponieważ nie będzie w nim nikogo. Taki kompleks nazywa się skierowanym grafem acyklicznym lub DAG w skrócie. I tak otrzymujemy DAG, a podróżnik jest zainteresowany przejściem z punktu A do punktu B. Chciwe podejście do problemu może spowodować, że ścieżka zostanie zbudowana, zaczynając od A i stale dodając do bieżącej niepełnej ścieżki najtańsza krawędź prowadząca z miasta dotychczas osiągniętego do jakiegoś jeszcze nieodwiedzonego miasta, aż do osiągnięcia docelowego miasta B. Rysunek pokazuje przykład zastosowania tego naturalnie wyglądającego algorytmu do grafu, którego minimalna ścieżka od A do B ma długość 13.



The graph

Greedy solution
Total cost:15

Dynamic planning solution
Total cost:13 (optimal)

Algorytm znajduje ścieżkę o długości 15, która nie jest tak dobra. Chciwość nie popłaca, ponieważ sprytny algorytm musi być na tyle sprytny, aby wziąć krawędź długości 5 do C, a następnie długość 3 do E, mimo że nie są to lokalnie najlepiej wyglądające opcje. Inna, nie zachłanna metoda algorytmiczna, zwana planowaniem dynamicznym, umożliwia dokonywanie tak subtelnych wyborów. (Właściwie ta metoda nazywana jest programowaniem dynamicznym, a nie planowaniem, ale zdecydowaliśmy się użyć tego drugiego słowa, aby lepiej oddać ideę metody i uniknąć konfliktu z algorytmicznym użyciem terminu „programowanie”, który odnosi się do kodowania algorytmu do wykonania przez komputer, jak wyjaśniono w Części 3. Dynamiczne planowanie opiera się na doprecyzowaniu dość prymitywnego kryterium natychmiastowej chciwości i można je abstrakcyjnie opisać w następujący sposób. Załóżmy, że rozwiązanie jakiegoś problemu algorytmicznego składa się z sekwencji wyborów, które mają doprowadzić do jakiegoś optymalnego rozwiązania. Jak już pokazano, jest całkiem możliwe, że samo

wybranie najlepiej wyglądającego wyboru spośród każdego lokalnego zestawu możliwości nie doprowadzi do optymalnego rozwiązania. Jednak często zdarza się, że optimum można uzyskać, rozpatrując wszystkie kombinacje (a) dokonania jednego wyboru oraz (b) znalezienia optymalnego rozwiązania mniejszego problemu reprezentowanego przez pozostałe wybory. Na przykład na rysunku długość najkrótszej drogi z A do B jest najmniejszą z trzech liczb uzyskanych przez wybranie najpierw jednego z miast C, G i D (tych bezpośrednio osiągalnych z A), a następnie dodanie jego odległości z A do długość najkrótszej drogi prowadzącej z niej do B. Symbolicznie, oznaczając długość najkrótszej drogi z X do B przez L(X), możemy napisać:

L(A) = minimum: 5 + L(C), 14 + L(G), 3 + L(D)

Innymi słowy, znajdujemy najkrótszą drogę z A do B, znajdując trzy „prostsze" najkrótsze ścieżki (te z każdego z C, G i D do B), łącząc ich rozwiązania z bezpośrednią krawędzią z A, a następnie wybierając najlepszy z trzech wyników. Oznacza to, że możemy znaleźć optymalne rozwiązanie, znajdując najpierw optymalne rozwiązania trzech „mniejszych" problemów, a następnie dokonując kilku uzupełnień i porównań. Proces ten można następnie kontynuować, pisząc na przykład:

L(D) = minimum: 7 + L(E), 6 + L(G), 11 + L(C)

Gdy takie wyprowadzenia dają klauzule określające L(B) (czyli minimalną odległość od B do siebie) nie są potrzebne dalsze prace, ponieważ nawet wyczerpany podróżnik wie, że L(B) wynosi po prostu 0, ponieważ najlepszym sposobem na przejście z punktu B do punktu B jest pozostanie tam, gdzie jesteś. Te obserwacje prowadzą do dynamicznego algorytmu planowania dla ogólnego problemu zmęczonego podróżnika (czasami zwanego problemem najkrótszej ścieżki), który działa od punktu końcowego B wstecz do A. W przykładzie z rysunku ?? najpierw oblicza najkrótsze ścieżki z F i E do B, czyli L(F) i L(E) (są to jedyne miasta, które prowadzą donikąd poza B). Następnie oblicza L(G) i L(C) (tu G i C są jedynymi miastami, które prowadzą tylko do B, F i E, czyli do miast już rozpatrzonych), potem L(D), a na końcu L (A). Każde obliczenie jest przeprowadzane na podstawie wyników już dostępnych, tak aby np. L(G) było minimum 6 (bezpośrednia odległość od G do B) i 7 + L(E). Wykonując tę procedurę, śledzimy również ścieżkę wsteczną, która jest stopniowo konstruowana od B do A; jest to poszukiwana optymalna ścieżka. Planowanie dynamiczne można traktować jako „dziel i zwyciężaj" doprowadzone do granic możliwości: wszystkie podproblemy są rozwiązywane w kolejności rosnącej, a wyniki są przechowywane w pewnej strukturze danych, aby ułatwić łatwe rozwiązania większych. Metodę można zastosować do wielu bardziej skomplikowanych problemów, które do przechowywania częściowych rozwiązań wymagają struktur danych bardziej złożonych niż zwykłe wektory. Możesz spróbować opracować algorytm dynamicznego planowania dla ściśle powiązanego problemu „podróżującego sprzedawcy"

Mnóstwo pracy i terminowa praca

Wiele algorytmów opiera się na sprytnym doborze struktur danych o właściwościach dostosowanych do potrzeb konkretnego algorytmu. Na przykład algorytm sortowania drzewa opisany w rozdziale 2 wykorzystuje drzewa wyszukiwania binarnego, które ograniczają umieszczanie elementów w taki sposób, że mniejsze elementy trafiają na lewo, a większe na prawo. Istnieje wiele rodzajów struktur danych, niektóre o ciekawych nazwach, takich jak „stosy Fibonacciego" lub „czerwono-czarne drzewa", z których każda nadaje się do określonego celu algorytmicznego. Załóżmy, że potrzebujemy algorytmu, który odbiera elementy w określonej kolejności i musi być w stanie dostarczyć najmniejszy z nich przy każdym zapytaniu. Rozważmy na przykład harmonogram drukarni. W każdej chwili klient może przyjechać z jakimś drukiem. Każda praca ma swój własny termin; niektóre są pilne, a inne mogą poczekać. Kiedy kserokopiarka staje się dostępna, kierownik sklepu musi znaleźć najpilniejszą pracę do wykonania. Tutaj elementami, które algorytm musi obsłużyć, są zadania drukowania uporządkowane

według ich terminów. Odpowiednią strukturą danych dla tego problemu jest sterta, która jest drzewem binarnym z tą właściwością, że wartość każdego węzła jest mniejsza niż wartości wszystkich jego potomków. Dzięki tej właściwości najmniejszy element sterty zawsze znajduje się w korzeniu drzewa i jest natychmiast dostępny. Kiedy rzeczywiście uzyskujemy dostęp do tego elementu, musimy go usunąć ze sterty (jak w przykładzie z drukarnią). Ale robiąc to, musimy zachować charakterystyczną właściwość sterty, zastępując ten najmniejszy element mniejszym z jego potomstwa, które właśnie stało się najmniejszym elementem sterty. Jeśli jednak po prostu przesuniemy ten nowy minimalny element do korzenia, stworzymy „dziurę" w drzewie, którą musi wypełnić mniejsze z jego potomstwa. To rzeczywiście się dzieje, a proces jest kontynuowany w dół, aż dotrzemy do liścia. Tyle o usunięciu najmniejszego elementu. Wkładanie nowego elementu do pryzmy jest podobne, z tą różnicą, że zaczyna się on od jednego z liści i przesuwa się w kierunku korzenia, aż do znalezienia właściwego miejsca (czyli do momentu, gdy wejście w górę naruszyłoby charakterystyczną właściwość hałdy). Skuteczna implementacja stert wykorzystuje wektor. Sterta zawierająca N elementów zajmie komórki od 1 do N wektora, przy czym dwa potomstwo węzła znajdującego się w komórce I znajduje się w komórkach 2 · I i 2 · I + 1. W tej reprezentacji , usunięcie elementu minimum nie może być wykonane w sposób opisany powyżej, ponieważ może to spowodować powstanie „dziury" w środku wektora. Zamiast tego element z ostatniej pozycji w wektorze zastępuje pierwszy element (ten właśnie usunięty), a następnie jest „bąbelkowany" w dół, aż znajdzie się we właściwym miejscu w stosie. Ta reprezentacja wektorowa ma pewne właściwości, które sprawiają, że algorytmy manipulacji stertą są dość wydajne

Niedestrukcyjne algorytmy

Inny elegancki przykład użycia struktur danych pochodzi z paradygmatu programowania funkcyjnego, omówionego w Części 3. Paradygmat ten sprawia, że programy są łatwiejsze do zrozumienia i zrozumienia, za cenę wyższego poziomu abstrakcji, który używa tylko niemodyfikowalnych obiektów. Zazwyczaj algorytmy są opisywane w trybie imperatywnym, przy użyciu struktur danych, które algorytm może modyfikować w miarę postępu. Z drugiej strony algorytmy funkcjonalne muszą traktować swoje struktury danych z większym szacunkiem, ponieważ nie można ich modyfikować. W tym widoku zamiast zmieniać struktury danych, zawsze tworzymy nowe. Na przykład operacja dodania elementu do stosu zwraca nowy stos, zawierający wszystkie elementy oryginalnego stosu oraz nowy element na górze. Pierwotny stos nie jest zmieniany i w razie potrzeby jest dostępny do dalszych obliczeń. Ponieważ do stosów można uzyskać dostęp tylko z jednego końca, dość łatwo jest zaimplementować stosy za pomocą połączonych list. Dodanie elementu do stosu to po prostu dodanie elementu na początek listy (a dokładniej stworzenie kolejnego linku, który wskazuje na oryginalną listę, która, jak wspomniano, nie ulega zmianie). Podobnie usunięcie górnego elementu stosu oznacza po prostu przejście do następnego elementu na liście . W przeciwieństwie do stosów kolejki są trudniejsze do zaimplementowania w języku funkcjonalnym, ponieważ umożliwiają dostęp (a zatem wymagają modyfikacji) z obu stron. Usunięcie elementu jest łatwe i odbywa się tak, jak w przypadku stosu. Jednak dodanie elementu oznacza dodanie go na końcu listy, jak w przykładzie JAVA, który widzieliśmy w Części 3. W języku imperatywnym można to łatwo zrobić, modyfikując wskaźnik „następny" ostatniego łącza na liście, ale jest to niemożliwe w języku funkcjonalnym. Oczywiście moglibyśmy skopiować całą listę i dodać nowy element na końcu nowej listy, ale byłaby to strata czasu i pamięci. Sprytny pomysł pozwala nam na realizację kolejek funkcjonalnych z taką samą wydajnością jak przy implementacji imperatywnej. Sztuczka polega na zaimplementowaniu każdej kolejki jako dwóch stosów. Stos „front" zawiera elementy z przodu kolejki, uporządkowane tak, że górny element stosu jest elementem przednim kolejki, a stos „back" zawiera elementy z tyłu kolejki, w odwrotnej kolejności order - górny element stosu będący ostatnim elementem kolejki . Dodanie elementu z tyłu kolejki jest teraz łatwe: jest umieszczany na tylnym stosie. Usuwanie elementu z kolejki jest równie proste: wystarczy usunąć

element znajdujący się na górze przedniego stosu. A co, jeśli przedni stos jest pusty, a tylny nie? W tym przypadku najpierw zamieniamy stos tylny w stos przedni, przesuwając wszystkie jego elementy do stosu przedniego w odwrotnej kolejności. Tylny stos staje się teraz pusty, co jest w porządku, ponieważ nigdy nie musimy niczego z niego usuwać. (W rzeczywistości nigdy nie pozwalamy, aby stos przedni stał się pusty, gdy stos tylny jest niepusty, wykonując to odwrócenie, gdy ostatni element jest usuwany ze stosu przedniego). tylny stos. Zauważ jednak, że każdy element kolejki przejdzie z tylnego stosu do przedniego stosu co najwyżej raz. Kosztem tej operacji można zatem „obciążyć" przenoszony element, tak że po zsumowaniu i uśrednieniu z pełnego wykonania algorytmu, każda operacja wstawiania lub usuwania zajmuje określoną ilość czasu. Ta metoda księgowa, zwana kosztem zamortyzowanym, jest ważną techniką analizy wydajności algorytmów

Algorytmy on-line

Załóżmy, że niektórzy rodzice po raz pierwszy w życiu zabierają swoje dzieci do ośrodka narciarskiego. Nie da się powiedzieć, jak bardzo polubią jeździć na nartach, a na ile będą chcieli je uprawiać. Narty można wypożyczyć lub kupić. Jeśli okaże się, że dzieci będą chciały dużo jeździć na nartach, taniej byłoby raz na zawsze kupić narty. Jeśli jednak nie, taniej byłoby po prostu wypożyczyć narty, kiedy tylko masz na to ochotę. Gdyby rodzice z góry wiedzieli, ile dzieci będą chciały jeździć na nartach, wybór byłby oczywisty. Jednak informacje te są nieznane i pozostaje nam przyjęcie pewnej strategii ogólnej formy, która wymaga rozpoczęcia od wypożyczenia nart kilka razy, aby zobaczyć, jak się sprawy mają, a następnie podjęcia decyzji o faktycznym zakupie. Jaka byłaby najlepsza strategia rodziców? Rozwiązanie tego jest przykładem interesującej klasy problemów, których rozwiązanie wymaga tak zwanych algorytmów on-line. Nazwa pochodzi od faktu, że te algorytmy muszą podejmować decyzje na bieżąco, nie znając wszystkich istotnych informacji; konkretnie, nie wiedzą, jakie wnioski mogą zostać złożone w ich sprawie w przyszłości. Pierwszym pytaniem przy analizie algorytmów on-line jest to, jak przeanalizować ich koszt. Zwykłą metodą jest porównanie każdego algorytmu z algorytmem wszechwiedzącym off-line - tym, który potrafi poprawnie przewidzieć przyszłość (w naszym przypadku, do jakiego stopnia dzieci będą cieszyć się jazdą na nartach). Oczywiście żaden algorytm on-line nie może działać lepiej niż ten algorytm offline; najlepszy algorytm on-line to ten, który jest najbliżej tego celu. W naszym przykładzie okazuje się, że najlepszym algorytmem on-line dla problemu z nartami jest wypożyczenie, aż koszt wypożyczenia będzie równy kosztowi zakupu, a następnie kupno. Ten algorytm jest mniej niż dwa razy gorszy od algorytmu wszechwiedzącego. Aby to zobaczyć, załóżmy, że koszt zakupu nart jest równy kosztowi wypożyczeń M i zastanów się, ile razy dzieci będą chciały jeździć na nartach. Jeśli jeżdżą na nartach mniej niż M razy, oba algorytmy zapłacą tę samą kwotę (wszystkowiedzący off-line wie z góry, że będzie jeździł mniej niż M razy, więc wynajmie, a nie kupi). Jeśli dzieci będą chciały jeździć na nartach więcej niż M razy, algorytm on-line wypożyczy $M-1$ razy, a następnie kupi, za łączny koszt równy $2 \cdot M - 1$ wypożyczeń. Algorytm off-line, znając przyszłość, kupi natychmiast, za cenę wynajmu M. W każdym razie algorytm on-line nigdy nie przekracza dwukrotności kosztu algorytmu off-line. Można wykazać, że żadna inna strategia nie okaże się ogólnie lepsza.4 Strategia, w ramach której rodzice kupią narty po wypożyczeniu K, gdzie K jest ściśle mniejsze niż $M - 1$, może kosztować $K + M$ wynajem jednostek, podczas gdy algorytm off-line płaci tylko $K + 1$. (Ile razy dzieci muszą chcieć jeździć na nartach, aby tak się stało?) Jeśli K jest większe niż $M - 1$, algorytm on-line może ponownie kosztowało rodziców $K+M$, podczas gdy algorytm off-line wzywa do natychmiastowego zakupu nart, za cenę tylko M wypożyczeń. W obu przypadkach koszt algorytmu on-line jest co najmniej dwukrotnością kosztu algorytmu wszechwiedzącego off-line.

Badania nad metodami algorytmicznymi

Naprawdę bardzo niewiele jest powszechnie akceptowanych paradygmatów, które są wystarczająco ogólne, by zasługiwały na specjalną nazwę i tytuł „metoda algorytmiczna", a większość bardziej

znanych została już opisana. Mimo to, bez szczególnego dążenia do ogólnych paradygmatów, informatycy nieustannie poszukują lepszych metod rozwiązywania coraz bardziej złożonych problemów algorytmicznych. Trochę trudno jest tu dalej omawiać te próby, gdyż kwestie efektywności wkradają się na bardzo wczesnym etapie, a efektywność została omówiona szczegółowo dopiero w rozdziale 6. Ponadto pojęcia współbieżności i probabilizmu, omówione w rozdziałach 10 i 11, stają się coraz bardziej kluczowe dla najnowszych osiągnięć w projektowaniu algorytmicznym. Omawiając te tematy, zobaczymy kilka dodatkowych sposobów na wymyślenie dobrych algorytmów.

Ćwiczenia

W kolejnych ćwiczeniach drzewo jest podawane przez (wskaźnik do) jego korzenia. Węzeł V drzewa z zewnętrznym stopniem N ma N potomstwa, oznaczony jako pierwszy do N-tego i zawiera pewną pozycję danych. Liść jest węzłem bez potomstwa. Drzewo binarne ogranicza liczbę potomstwa każdego węzła do maksymalnie dwóch. Głębokość węzła drzewa jest następująca: głębokość korzenia wynosi 0, a jeśli głębokość V wynosi N to głębokość jego potomstwa wynosi N + 1. Dla węzła V dostępne operacje obejmują pobranie jego zawartość, sprawdzenie, czy ma I-te potomstwo, a jeśli tak, przypisanie wskaźnika do tego potomstwa.

1. Rozważ problem sumowania wynagrodzeń pracowników zarabiających więcej niż ich bezpośredni przełożony, zakładając, że każdy pracownik ma jednego przełożonego. Pracownicy są oznaczeni etykietami 1, 2 itd. Napisz algorytmy rozwiązujące problem dla każdej z poniższych reprezentacji danych wejściowych:

(a) Dane wejściowe są liczbą całkowitą N i dwuwymiarową tablicą A, gdzie N to liczba pracowników, A[I, 1] to wynagrodzenie I pracownika, a A[I, 2] to wynagrodzenie etykietę swojego menedżera.

(b) Dane wejściowe są podawane przez drzewo binarne skonstruowane w następujący sposób: Korzeń drzewa reprezentuje pierwszego pracownika. Dla każdego węzła V drzewa reprezentującego I pracownika,

- V zawiera wynagrodzenie I pracownika;

- pierwsze potomstwo V to listek zawierający etykietę kierownika I pracownika; oraz

- jeśli jest więcej niż I pracowników, drugie potomstwo V jest węzłem reprezentującym I + 1 pracownika.

2.

(a) Napisz algorytm, który mając dane drzewo T , oblicza sumę głębokości wszystkich węzłów T .

(b) Napisz algorytm, który mając drzewo T i dodatnią liczbę całkowitą K, oblicza liczbę węzłów w T na głębokości K.

(c) Napisz algorytm, który przy danym drzewie T sprawdza, czy ma on jakiś liść na równej głębokości.

3. Napisz algorytmy, które rozwiązują następujące problemy, wykonując przechodzenie wszerz danych drzew. Możesz założyć dostępność kolejki Q. Operacje na Q obejmują dodanie przedmiotu z tyłu, odzyskanie i usunięcie przedmiotu z przodu oraz testowanie Q pod kątem pustki.

(a) Mając drzewo T, którego węzły zawierają liczby całkowite, wypisz listę składającą się z sumy zawartości węzłów na głębokości 0, sumy zawartości węzłów na głębokości 1 itd.

(b) Mając dane drzewo T , znajdź głębokość K z maksymalną liczbą węzłów w T . Jeśli jest kilka takich Ks, zwróć ich maksimum.

Wyrażenie arytmetyczne utworzone przez nieujemne liczby całkowite i standardową jednoargumentową operację „−” oraz operacje binarne „+”, „−”, „×” i „/” może być reprezentowane przez drzewo binarne w następujący sposób:

- Liczba całkowita I jest reprezentowana przez liść zawierający I .

- Wyrażenie −E, gdzie E jest wyrażeniem, jest reprezentowane przez drzewo, którego korzeń zawiera „−”, a jego pojedyncze potomstwo jest korzeniem poddrzewa reprezentującego wyrażenie E.

- Wyrażenie E ∗ F, gdzie E i F są wyrażeniami, a „-” jest operacją binarną, jest reprezentowane przez drzewo, którego korzeń zawiera „∗”, jego pierwsze potomstwo jest korzeniem poddrzewa reprezentującego wyrażenie E, a jego drugie potomstwo jest korzeniem poddrzewa reprezentującego F. Zauważ, że symbol „-” oznacza zarówno operacje jednoargumentowe, jak i binarne, a węzły drzewa zawierające ten symbol mogą mieć stopień zewnętrzny 1 lub 2.

4. Zaprojektuj algorytm sprawdzający, czy dane drzewo reprezentuje wyrażenie arytmetyczne.

5. (a) Zaprojektuj algorytm, który oblicza wartość wyrażenia arytmetycznego, biorąc pod uwagę jego reprezentację w postaci drzewa. Zauważ, że dzielenie przez zero jest niezdefiniowane.

(b) Rozszerz swój algorytm, aby najpierw wydrukować wyrażenie reprezentowane przez drzewo wejściowe, a następnie znak równości „=” i jego ocenę. Wyrażenie drukowane powinno być w całości w nawiasach, tj. para pasujących nawiasów powinna obejmować każde zastosowanie operacji binarnej. Mówimy, że dwa wyrażenia arytmetyczne E i F są izomorficzne, jeśli E można uzyskać z F przez zastąpienie niektórych nieujemnych liczb całkowitych innymi. Na przykład wyrażenia (2 + 3) × 6 − (−4) i (7 + 0) × 6 − (−9) są izomorficzne, ale żadne z nich nie jest izomorficzne z żadnym z (−2 + 3) × 6 − (−4) i (7 + 0) + 6 − (−9).

Mówi się, że wyrażenie E jest zrównoważone, jeśli każda operacja binarna w nim jest zastosowana do dwóch wyrażeń izomorficznych. Na przykład wyrażenia -5, (1 + 2) ∗ (3 + 5) i ((−3)/(−4))/((−1)/(−100)) są zrównoważone, natomiast 12 + (3 + 2) i (−3) ∗ (−3) nie są.

6. Zaprojektuj algorytm sprawdzający, czy dwa wyrażenia są izomorficzne, biorąc pod uwagę ich reprezentację w formie drzewa.

7. Zaprojektuj algorytm sprawdzający, czy wyrażenie jest zrównoważone, biorąc pod uwagę jego reprezentację w postaci drzewa. (Wskazówka: wykonaj najpierw przejście wszerz drzewa.)

8. Udowodnij, że maksymalna odległość między dowolnymi dwoma punktami wielokąta występuje między dwoma wierzchołkami.

9. Napisz program implementujący algorytm maksymalnej odległości wielokątnej.

10. Zaprojektuj algorytm, który przy danych (wierzchołkach) niekoniecznie wypukłego wielokąta znajduje parę wierzchołków o minimalnej odległości.

11. Napisz algorytmy, które znajdują dwa maksymalne elementy w danym wektorze składającym się z N odrębnych liczb całkowitych (załóżmy, że N > 1).

(a) Za pomocą metody iteracyjnej.

(b) Stosowanie metody dziel i zwyciężaj.

12. Napisz szczegółowo opisany w tekście zachłanny algorytm znajdowania minimalnego drzewa opinającego. Problem plecaka całkowitoliczbowego polega na znalezieniu sposobu na wypełnienie

plecaka o określonej pojemności niektórymi elementami z danego zestawu dostępnych przedmiotów różnego typu w najbardziej opłacalny sposób. Dane wejściowe do problemu składają się z:

- C, całkowita nośność plecaka;

- dodatnia liczba całkowita N, liczba typów pozycji;

- wektor Q, gdzie Q[I] jest dostępną liczbą elementów typu I;

- wektor W, gdzie W[I] jest wagą każdego elementu typu I spełniającego $0 < W[I] \leq C$;

oraz

- wektor P, gdzie P[I] jest zyskiem z przechowania przedmiotu typu I w plecaku.

Wszystkie wartości wejściowe są nieujemnymi liczbami całkowitymi. Problem polega na wypełnieniu plecaka elementami, których łączna waga nie przekracza C, tak aby łączny zysk z plecaka był maksymalny. Wynikiem jest wektor F, gdzie F[I] zawiera liczbę elementów typu I, które są wkładane do plecaka. Problem plecakowy jest odmianą problemu plecakowego, w którym zamiast dyskretnych przedmiotów są materiały. Różnica polega na tym, że zamiast pracować z liczbą całkowitą liczb, możemy włożyć do plecaka dowolną ilość materiału I, która nie przekracza dostępnej ilości Q[I]. Wektory W i P zawierają teraz odpowiednio wagę i zysk jednej jednostki ilościowej materiału I . Wszystkie wartości wejściowe i wyjściowe są teraz nieujemnymi liczbami rzeczywistymi, niekoniecznie liczbami całkowitymi.

13. (a) Zaprojektuj dynamiczny algorytm planowania dla problemu plecakowo-całkowitego.

(b) Jaki jest wynik twojego algorytmu dla danych wejściowych?

- N = 5

- C = 103

- Q = [3,1,4,5,1]

- W = [10,20,20,8,7]

- P = [17,42,35,16,15]

a jaki jest całkowity zysk z plecaka?

14. (a) Zaprojektuj zachłanny algorytm dla problemu plecakowego.

(b) Jaki jest wynik twojego algorytmu dla danych wejściowych podanych w ćwiczeniu 4.13(b) i jaki jest teraz całkowity zysk z plecaka?

15. (a) W jaki sposób odniesiesz łączne zyski uzyskane dla danego wejściowego typu liczb całkowitych, gdy zostaniesz poddany problemowi plecakowemu i problemowi plecakowemu typu Integer?

(b) Rozważ modyfikację algorytmu, który zaprojektowałeś w ćwiczeniu 4.14(a), która daje w F część całkowitą wielkości obliczonych przez oryginalny algorytm. Udowodnij, że zmodyfikowany algorytm nie rozwiązuje problemu plecakowego. Oznacza to, że podaj dane wejściowe w postaci liczb całkowitych, dla których (zmodyfikowany) algorytm zachłanny wygeneruje akceptowalne wypełnienie liczb całkowitych, które nie jest maksymalnie opłacalne. Znajdź takie dane wejściowe z N, liczbą typów, jak najmniejszą. (Wskazówka: poprawne rozwiązania problemu z plecakiem całkowitym, w

przeciwieństwie do problemu z plecakiem, mogą pozostawić dostępne przedmioty poza plecakiem, nawet jeśli nie jest on pełny.)

Poprawność algorytmów

* Na początku lat 60. jeden z amerykańskich statków kosmicznych z serii Mariner wysłany na Wenus zaginął na zawsze kosztem milionów dolarów z powodu błędu w programie komputerowym kontroli lotu.

* W 1981 roku jedna ze stacji telewizyjnych relacjonujących wybory w prowincji Quebec w Kanadzie, przez swoje błędne programy komputerowe przekonała, że w rzeczywistości przewodniczy jej mała partia, początkowo uważana za pozbawioną jakichkolwiek szans. Te informacje i wynikające z nich odpowiedzi komentatorów zostały przekazane milionom widzów.

* W serii incydentów w latach 1985-1987 kilku pacjentów otrzymało ogromne przedawkowanie promieniowania z systemów radioterapii Therac-25; troje z nich zmarło z powodu powstałych komplikacji. Sprzętowe blokady bezpieczeństwa z poprzednich modeli zostały zastąpione testami bezpieczeństwa oprogramowania, ale wszystkie te incydenty wiązały się z błędami programistycznymi.

* Kilka lat temu pewna duńska dama otrzymała, około swoich 107 urodzin, skomputeryzowany list od lokalnych władz szkolnych z instrukcjami dotyczącymi procedury rejestracji w pierwszej klasie szkoły podstawowej. Okazało się, że na pole „wiek" w bazie danych przypisano tylko dwie cyfry.

* Na przełomie tysiącleci problemy z oprogramowaniem stały się nagłówkami wiadomości wraz z tak zwanym problemem roku 2000, czyli błędem Y2K. Obawiano się, że 1 stycznia 2000 r. rozpęta się piekło, ponieważ komputery używające dwóch cyfr do przechowywania lat błędnie zakładają, że rok podany jako 00 to 1900, podczas gdy w rzeczywistości był to rok 2000. z perspektywy czasu, całkiem udane) wysiłki mające na celu poprawienie tych programów musiały zostać podjęte przez firmy programistyczne na całym świecie.

To tylko kilka z licznych opowieści o błędach oprogramowania, z których wiele zakończyło się katastrofami, często z utratą życia. Nie sposób przecenić wagi kwestii poprawności. Przez cały czas naiwnie zakładaliśmy, że algorytmy i programy, które piszemy, robią dokładnie to, co zamierzamy zrobić. To nie ma żadnego uzasadnienia

Błędy językowe

Jeden z najczęstszych rodzajów błędów występujących przy przygotowywaniu programów komputerowych wynika z nadużywania składni języka programowania. Spotkaliśmy się z nimi w poprzednim rozdziale. Zapis:

for Y from 1 until N do

zamiast:

for Y from 1 to N do

jak wymaga tego język, jest zły, ale nie jest to błąd algorytmu. Błędy składniowe są jedynie uciążliwym przejawem tego, że algorytmy realizowane przez komputer muszą być prezentowane w stroju formalnym. Kompilatory i interpretery są tworzone w celu wykrywania błędów składniowych i powiadamiają programistę, który zazwyczaj będzie w stanie je poprawić przy niewielkim wysiłku. Co więcej, i tutaj kompilatory mają przewagę nad interpreterami, sprytny kompilator sam spróbuje poprawić pewne rodzaje błędów składniowych, umożliwiając pożądane tłumaczenie na język maszynowy. Kompilator nie jest ograniczony, tak jak interpreter, do przeglądania jednej linii lub jednej instrukcji programu na raz. Zwykle jest również zaprogramowany do wykrywania bardziej subtelnych

błędów, które zamiast naruszać lokalne reguły składniowe języka, powodują sprzeczności między prawdopodobnie odległymi częściami programu, zwykle między definicją a instrukcją operacyjną. Przykłady obejmują operacje arytmetyczne zastosowane do zmiennych nienumerycznych, odwołania do 150. elementu wektora, którego indeksy zostały zdefiniowane w zakresie od 1 do 100, oraz wywołania podprogramów z niewłaściwą liczbą parametrów. Wszystko to jednak oznacza również nieprawidłowe użycie języka. Niezależnie od tego, czy kompilator lub interpreter wykryje taki błąd z wyprzedzeniem, próba uruchomienia programu zakończy się niepowodzeniem, gdy zostanie osiągnięta obraźliwa część; program zostanie przerwany, to znaczy zostanie zatrzymany, a użytkownikowi wyświetli się odpowiedni komunikat. W przeciwieństwie do błędów omówionych w następnym podrozdziale i pomimo potencjalnie nieprzyjemnego charakteru awarii programu, błędy językowe nie są uważane za najpoważniejsze. Często są one wykrywane automatycznie i zazwyczaj można je stosunkowo łatwo skorygować.

Błędy logiczne

Przypomnijmy algorytm liczenia „pieniądze" wprowadzony w Części 2. Problem polegał na liczeniu zdań zawierających wystąpienia słowa „pieniądze". Rozwiązanie polegało na przeprowadzeniu wyszukiwania słowa „pieniądze", a następnie wyszukania końca zdania, które umownie jest zawsze oznaczane w tekście wejściowym przez kombiację „. " , a mianowicie kropka, po której następuje spacja. Po pomyślnym przeprowadzeniu obu wyszukiwań początkowo wyzerowany licznik jest zwiększany, a wyszukiwanie „pieniądze" jest wznawiane od początku następnego zdania . Co by się stało, gdyby algorytm użył „." (bez spacji) zamiast „. "? Załóżmy na razie, że nie mówimy o algorytmie, ale o jego wersji formalnej, jako programie nie zawierającym błędów językowych. Oczywiste jest, że nowa wersja, która różni się od oryginału jedynie brakiem miejsca, również nie zawiera błędów językowych. Dla obserwatora, łącznie z kompilatorami i interpreterami, nowy program jest doskonały. Nie tylko nie ma dostrzegalnych błędów składniowych, ale za każdym razem, gdy zostanie uruchomiony na tekście wejściowym, program posłusznie zatrzymuje się i wyświetla liczbę jako końcową wartość swojego licznika. Oczywiście w nowym programie jest błąd. Wynika to z faktu, że kropki mogą występować w zdaniach. Rozważ następujące:

Całkowita suma pieniędzy na moim koncie bankowym to 322.56 dolarów, naprawdę niezwykła suma, biorąc pod uwagę mój talent do zarabiania pieniędzy. Jestem bogatym człowiekiem.

Po włączeniu tego dwuzdaniowego tekstu nasza zmodyfikowana wersja wygeneruje 2, mimo że słowo „pieniądze" pojawia się tylko w pierwszym zdaniu. Program jest oszukany przez przecinek dziesiętny pojawiający się w 322.56 USD. Nowy program jest poprawny pod względem językowym i faktycznie rozwiązuje problem algorytmiczny, ale niestety nie dokładnie ten, który zamierzaliśmy rozwiązać. Program zawiera to, co nazwiemy błędem logicznym, co skutkuje nie niepoprawnym składniowo lub bezsensownym programem, ale programem, który robi coś innego niż to, do czego był przeznaczony. Błędy logiczne mogą być notorycznie nieuchwytne. Chociaż może nie być zbyt trudno zauważyć, że spacja została pominięta w „. " w programie do liczenia pieniędzy, następujący błąd nie jest tak łatwy do znalezienia. Załóżmy, że różne wskaźniki do tekstu są używane do wyszukiwania „pieniądze" i „. " Szukaj. Oczywiście raz ". ", a licznik został zwiększony, pierwszy wskaźnik powinien zostać przekierowany na pozycję drugiego przed wznowieniem wyszukiwania słowa „pieniądze". Niezastosowanie się do tego stanowi błąd logiczny, który również da 2 po uruchomieniu w poprzednim przykładzie, ale z innego powodu; tym razem granice zdań są prawidłowo wykrywane, ale licznik jest zwiększany dwukrotnie w obrębie pierwszego zdania. (Dlaczego?) Takie błędy nie wskazują, że coś jest nie tak z programem per se, ale że coś jest nie tak z połączeniem programu i konkretnego problemu algorytmicznego; program, który sam w sobie jest w porządku, nie rozwiązuje poprawnie tego problemu. Błędy logiczne mogą być spowodowane niezrozumieniem semantyki języka programowania

(„Myślałem, że X*Y oznacza X podniesione do potęgi Y , a nie X razy Y ” lub „Byłem pewien, że kiedy pętla formy ponieważ Y od 1 do N jest uzupełnione, wartość Y to N, a nie N + 1”), w takim przypadku możemy nazwać je błędami semantycznymi. Jednak o wiele bardziej typowe jest napotkanie „prawdziwych” błędów logicznych; to znaczy błędy w procesie logicznym używanym przez projektanta algorytmu do rozwiązania problemu algorytmicznego. Nie mają one nic wspólnego z programem napisanym później w celu zaimplementowania algorytmu. Stanowią one wady samego algorytmu, gdy są rozważane jako proponowane rozwiązanie problemu. Są to błędy algorytmiczne i to właśnie one nas tutaj interesują. Nieprzekierowanie licznika „pieniądze” do następnego zdania jest błędem algorytmicznym. Czy zatem błędne powiązanie we wczesnej wersji Rysunku 2.4 przedstawionej na Rysunku 5.1 dotyczyło jednego z wyjść z „czy pensja P jest wyższa niż Q?” na „przejść P do następnego pracownika” zamiast „czy P jest na końcu listy?”

Komputery nie błądzą

Analogia między algorytmami a recepturami zawodzi, jeśli chodzi o kwestie poprawności. Kiedy próba gotowania lub pieczenia nie powiedzie się, mogą być dwa powody:

1. winę ponosi „sprzęt” lub

2. przepis jest nieprecyzyjny i niejasny.

W przeważającej części pierwszy z nich jest tak naprawdę powodem, zwłaszcza po tym, jak zdecydowaliśmy, jak rzeczywiście zrobiliśmy, że kucharze i piekarze są częścią sprzętu. Ale jeśli są problemy z przepisem, a nie z piekarzem, piekarnikiem czy sztućcami, to zazwyczaj mają one do czynienia z założeniami autora na temat kompetencji jego użytkowników. „Ubijaj białka na pieniste” wymaga od piekarza pewnej wiedzy na temat piany jajecznej, bez której efektem pieczenia może nie być mus czekoladowy, ale czekoladowy bałagan! W przeciwieństwie do przepisów, algorytmy napisane do wykonania przez komputer kończą w formalnym, jednoznacznym języku programowania, który prawie całkowicie eliminuje przyczynę (2). Co więcej, powód (1) można również odrzucić. Ogólnie rzecz biorąc, komputery nie popełniają błędów! Błąd sprzętowy to taka rzadkość we współczesnych komputerach, że gdy nasz wyciąg bankowy jest błędny i bankier mamrocze coś w tym sensie, że komputer się pomylił, możemy być pewni, że to nie komputer popełnił błąd - to chyba jeden pracowników banku. Albo w jednym z programów wprowadzono nieprawidłowe dane, albo sam program, oczywiście napisany przez człowieka, zawierał błąd. Nieprawidłowo działający program nie jest wynikiem problemu z komputerem. Jeśli dane wejściowe zostaną sprawdzone i okaże się, że są poprawne, problem dotyczy programu i jego podstawowego algorytmu.

Testowanie i debugowanie

Błędy algorytmiczne mogą pozostać niewykryte przez wieki. Czasami nigdy nie są wykrywane. Jest całkiem możliwe, że dane wejściowe, dla których błąd generuje nieprawidłowe dane wyjściowe, po prostu nie wystąpią w okresie życia algorytmu. Ewentualnie takie dane wejściowe mogą się rzeczywiście pojawić, ale nieprawidłowe dane wyjściowe mogą nigdy nie zostać zauważone. Niektóre błędy logiczne pojawiają się, gdy procesor nie może wykonać instrukcji z jakiegoś nieoczekiwanego powodu. Na przykład próba podzielenia X przez Y nie powiedzie się, jeśli Y będzie w tym czasie równe 0. Podobnie niepowodzenie będzie wynikać z próby zejścia w dół drzewa z węzła, który akurat jest liściem (czyli nie ma dokąd zejść). Nie są to błędy językowe, a kompilator generalnie nie będzie w stanie ich wcześniej wykryć. Nazywa się je błędami czasu wykonania i wynikają z błędów logicznych w projekcie algorytmu. Często wada polega po prostu na zapomnieniu o odrębnym traktowaniu specjalnych przypadków „z pogranicza”, takich jak zera (w liczbach) i liście (w drzewach). Projektant może wypróbować algorytm na kilku typowych i nietypowych danych wejściowych i nie znaleźć błędu.

W rzeczywistości programista zwykle testuje program na wielu wejściach, czasami nazywanych zestawami testowymi, i stopniowo usuwa z niego błędy językowe i większość błędów logicznych. Nie może być jednak pewien, że program (i leżący u jego podstaw algorytm) jest całkowicie wolny od błędów, po prostu dlatego, że większość problemów algorytmicznych ma nieskończone zestawy danych wejściowych, a zatem nieskończenie wiele kandydujących zestawów testowych, z których każdy może ujawnić nowy błąd. Błędy logiczne, jak ktoś kiedyś powiedział, są jak syreny. Sam fakt, że ich nie widziałeś, nie oznacza, że nie istnieją. Proces wielokrotnego wykonywania algorytmu lub uruchamiania programu w celu znalezienia i wyeliminowania błędów nazywa się debugowaniem. Nazwa ma ciekawą historię. Jeden z pierwszych komputerów, które zbudowano, pewnego dnia przestał działać, a później okazało się, że w kluczowej części obwodu zablokował się duży owad. Od tego czasu błędy, zwykle błędy logiczne, są pieszczotliwie nazywane błędami. Debugowanie programu, zwłaszcza złożonego i długiego, może być dość dużym przedsięwzięciem. Nawet jeśli zaobserwuje się, że program generuje nieprawidłowe dane wyjściowe w określonym przypadku testowym, może nie być dostępnych informacji o źródle błędu. Istnieje jednak kilka technik zawężania możliwości. Wersję programu można uruchomić ze sztucznie wstawionymi instrukcjami dla wydruków pośrednich. Pokazują one częściowe wyniki i wartości debuggera podczas wykonywania. Alternatywnie, jeśli program jest zinterpretowany, a nie skompilowany, jego wykonanie można śledzić wiersz po wierszu, umożliwiając wykrycie podejrzanych wartości pośrednich, zwłaszcza podczas pracy z wyświetlaczem interaktywnym. Wspomnieliśmy również o środowiskach programistycznych. Obsługują one wiele rodzajów narzędzi testujących i symulacyjnych, aby pomóc projektantowi algorytmów i programiście w uzyskaniu właściwego działania. Należy jednak ponownie podkreślić, że żadna z tych metod nie gwarantuje wolnych od błędów algorytmów, które generują prawidłowe dane wyjściowe na dowolnym legalnym wejściu. Jak ktoś kiedyś to ujął, testowanie i debugowanie nie może służyć do wykazania braku błędów w oprogramowaniu, a jedynie ich obecności.

Nieskończone pętle

Nieprawidłowość algorytmu jako rozwiązania problemu algorytmicznego może zatem objawiać się albo wykonaniem, które kończy się normalnie, ale z nieprawidłowymi danymi wyjściowymi, albo przerwaniem wykonania. Jakby co gorsza, algorytm uruchomiony na jednym z legalnych danych wejściowych problemu może w ogóle się nie zakończyć! Oczywiście jest to również błąd. Nieskończone obliczenia lub nieskończone pętle, jak się je czasami nazywa, mogą mieć charakter oscylacyjny, powtarzający w kółko pewną liczbę instrukcji na tych samych wartościach, lub nieoscylacyjny, ale rozbieżny charakter, co skutkuje coraz większym lub coraz mniejszym wartości. Przykładem oscylującej pętli jest procedura wyszukiwania, w której nie ma instrukcji do przekazania odpowiedniego wskaźnika; algorytm kontynuuje wyszukiwanie w tym samym obszarze struktury danych. Przykładem rozbieżności jest pętla, która zwiększa X o 1 w każdym przejściu i która jest instruowana, aby zakończyć, gdy X osiągnie 100. Jeśli pętla zostanie błędnie osiągnięta z początkową wartością 17,6 w X, pętla „chybi" 100 i zwiększaj X w nieskończoność. Oczywiście, prawdziwe komputery mają ograniczoną pamięć i generalnie przerywają programy drugiego rodzaju, gdy wartość X przekroczy pewne ostateczne maksimum, ale algorytm, na którym oparty jest program, mimo to dopuszcza nieskończoną pętlę. Testowanie i debugowanie może również pomóc w wykrywaniu potencjalnych nieskończonych pętli. Wypisując wartości pośrednie, debugger może zauważyć podejrzane oscylacje lub nienormalny wzrost lub spadek wartości, które pozostawione bez zmian mogą prowadzić do braku zakończenia. Tak jak poprzednio, zawsze jest więcej danych wejściowych niż możemy przetestować, dlatego żadna taka metoda nie gwarantuje znalezienia wszystkich potencjalnych nieskończonych pętli. Dla nich brak zakończenia jest błogosławieństwem, a zakończenie wskazuje na obecność błędu. Na razie jednak poprawny algorytm musi kończyć się normalnie na wszystkich legalnych danych wejściowych i za każdym razem generować właściwe dane wyjściowe.

Poprawność częściowa i całkowita

Jak omówiono w Części 1, problem algorytmiczny można zwięźle podzielić na dwie części:

1. specyfikację zestawu danych wejściowych prawnych; oraz

2. związek między wejściami a pożądanymi wyjściami.

Na przykład może być wymagane, aby każdy wpis prawny składał się z listy L słów w języku angielskim. Relacja między danymi wejściowymi a pożądanymi wynikami może określać, że wynik musi być listą zawierającą słowa w L posortowane w rosnącym porządku leksykograficznym. W ten sposób określiliśmy problem algorytmiczny, który wymaga algorytmu A, który sortuje każdą listę legalnych danych wejściowych L. Aby ułatwić precyzyjne potraktowanie problemu poprawności dla algorytmów, badacze rozróżniają dwa rodzaje poprawności, w zależności od tego, czy zakończenie jest, czy nie jest zawarty. W jednym przypadku zakłada się a priori, że program się kończy, a w drugim tak nie jest. Mówiąc dokładniej, mówi się, że algorytm A jest częściowo poprawny (w odniesieniu do definicji legalnych danych wejściowych i pożądanego związku z wynikami), jeśli dla każdego legalnego wejścia X, jeśli A kończy się po uruchomieniu na X, to określona relacja zachodzi między X i wynikowy zestaw wyjściowy. Zatem częściowo poprawny algorytm sortowania może nie kończyć się na wszystkich dozwolonych listach, ale zawsze, gdy tak się dzieje, wynikiem jest poprawnie posortowana lista. Mówimy, że A kończy działanie, jeśli zatrzyma się po uruchomieniu na jednym z dozwolonych danych wejściowych. Oba te pojęcia wzięte razem - częściowa poprawność i zakończenie - dają całkowicie poprawny algorytm, który poprawnie rozwiązuje problem algorytmiczny dla każdego legalnego wejścia: proces uruchamiania A na dowolnym takim wejściu X rzeczywiście kończy się i daje wyniki spełniające pożądaną zależność

Konieczność udowodnienia poprawności

Teraz wiemy dokładnie, co chcielibyśmy ustalić w obliczu problemu algorytmicznego i proponowanego rozwiązania, i dysponujemy różnymi technikami testowania i debugowania, które mogą nam w tym pomóc. Jednak żadna z tych technik nie jest niezawodna i podobnie jak przykłady przytoczone na początku części, folklor informatyczny pełen jest opowieści o katastrofach, niektóre z nich śmiertelne, inne powodujące utratę niewiarygodnych ilości pieniędzy, a wszystkie następujące od błędów algorytmicznych, zwykle w dużych i złożonych systemach oprogramowania. Twierdzenie, że dany algorytm jest poprawny w odniesieniu do problemu algorytmicznego, jest być może mniej głębokie niż twierdzenie, że syreny są wyimaginowane, ale może być tak ważne, że zależy od tego wiele żyć lub fortun. Wystarczy pomyśleć o systemach komputerowych, które kontrolują broń nuklearną lub wielomilionowe transakcje, aby docenić ten punkt. Niepokój budzą jednak nie tylko liczne nagłaśniane i niepublikowane przypadki błędów. Powszechnie uważa się, że ponad 70% (!) wysiłku i kosztów opracowania złożonego systemu oprogramowania jest w taki czy inny sposób przeznaczane na korygowanie błędów. Obejmuje to opóźnienia spowodowane błędnie przyjętymi specyfikacjami (tj. niejasnymi i nieprecyzyjnymi definicjami problemów algorytmicznych), szeroko zakrojone testowanie i debugowanie samych algorytmów, a co najgorsze, zmiany i przepisywanie już działających systemów (ogólnie określane jako konserwacja), ponieważ wynik nowo odkrytych błędów. Odnosząc się do dużych systemów oprogramowania używanych komercyjnie, sytuację opisano ładnie, mówiąc, że oprogramowanie jest udostępniane do użytku nie wtedy, gdy wiadomo, że jest poprawne, ale gdy tempo odkrywania nowych błędów spada do poziomu, który menedżerowie uznają za akceptowalny. Ta sytuacja jest wyraźnie zła. Potrzebujemy sposobów na udowodnienie, że algorytm jest bez wątpienia poprawny. Nikt nie pyta, czy może istnieć jakiś „nieodkryty” trójkąt równoboczny o nierównych kątach. Ktoś udowodnił raz na zawsze, że wszystkie trójkąty równoboczne mają jednakowe kąty i od tej pory wątpliwości nie pozostały. Czy można coś zrobić, aby ułatwić takie dowody? Czy sam

komputer może pomóc zweryfikować poprawność naszych algorytmów? Właściwie to, czego najbardziej chcielibyśmy, to automatyczny weryfikator; mianowicie jakiś rodzaj superalgorytmu, który przyjąłby jako dane wejściowe opis problemu algorytmicznego P i algorytm A, który jest proponowany jako rozwiązanie, i określałby, czy rzeczywiście A rozwiązuje P. Być może wskazałby również błędy, jeśli odpowiedź brzmiała nie. Nie można skonstruować takiego weryfikatora. Na razie jednak zignorujmy kwestię skorzystania z pomocy komputera. Czy możemy sami udowodnić poprawność naszych algorytmów? Czy jest jakiś sposób, w jaki możemy wykorzystać formalne, matematyczne techniki do realizacji tego celu? Tutaj mamy lepsze wieści.

Niezmienniki i zbieżności

Rzeczywiście istnieją metody weryfikacji programu. W rzeczywistości, w pewnym sensie technicznym, każdy poprawny algorytm można rygorystycznie wykazać, że jest poprawny! Zanim zaczniemy ilustrować przykładem, powiedzmy coś o samych metodach dowodowych. Próbując ustalić częściową poprawność, nie jesteśmy zainteresowani wykazaniem, że pewne pożądane rzeczy się zdarzają, ale wykazaniem, że pewne niepożądane rzeczy się nie zdarzają. Nie obchodzi nas, czy wykonanie kiedykolwiek osiągnie punkt końcowy, ale jeśli tak się stanie, nie będziemy w sytuacji, w której wyniki będą różnić się od oczekiwanych. W związku z tym chcemy uchwycić zachowanie algorytmu poprzez dokładne stwierdzenie, co robi w określonych momentach. Aby udowodnić częściową poprawność, dołączamy zatem twierdzenia pośrednie do różnych punktów kontrolnych w tekście algorytmu. Dołączenie asercji do punktu kontrolnego oznacza, że wierzymy, że ilekroć wykonanie osiągnie dany punkt, w każdym wykonaniu algorytmu na dowolnym legalnym wejściu, asercja będzie prawdziwa. Obejmuje to oczywiście punkty, które są osiągane wiele razy w ramach jednego wykonania, w szczególności te w pętlach. Z tego powodu takie twierdzenia są powszechnie nazywane niezmiennikami; pozostają prawdziwe bez względu na to, jak często są osiągane. Na przykład algorytm sortowania może być taki, że w pewnym momencie tekstu sortowana jest dokładnie połowa listy wejściowej, w którym to przypadku możemy dołączyć do tego punktu twierdzenie „połowa listy jest posortowana". Bardziej typowe jest jednak to, że niezmienniki zależą od wartości zmiennych dostępnych w punkcie kontrolnym. W związku z tym do pewnego momentu możemy dołączyć twierdzenie, że „częściowa lista od pierwszej lokalizacji do X jest posortowana", gdzie X jest zmienną, która zwiększa się w miarę sortowania większej ilości listy. Początkowe stwierdzenie, a mianowicie to, które jest dołączone do punktu początkowego algorytmu, jest zwykle formułowane w celu uchwycenia wymagań dotyczących legalnych danych wejściowych, i podobnie, końcowe stwierdzenie, związane z punktem końcowym, ujmuje pożądaną relację wyników do wejść . Załóżmy teraz, że możemy ustalić, że wszystkie załączone przez nas twierdzenia są rzeczywiście niezmiennikami, co oznacza, że są prawdziwe, gdy tylko zostaną osiągnięte. Wtedy w szczególności ostateczne stwierdzenie jest również niezmiennikiem. Ale to oznacza, że algorytm jest częściowo poprawny. Dlatego wszystko, co musimy zrobić, to ustalić niezmienność naszych twierdzeń. Odbywa się to poprzez ustalenie pewnych lokalnych własności naszych stwierdzeń, czasami nazywanych warunkami weryfikacji, tak że przechodzenie lokalnie od punktu kontrolnego do punktu kontrolnego nie powoduje żadnych naruszeń własności niezmienności. Takie podejście do udowadniania poprawności jest czasami nazywane niezmienną metodą asercji lub metodą Floyda, od nazwiska jednego z jej wynalazców. Jak zabieramy się za wybór punktów kontrolnych i pośrednich asercji oraz jak ustalamy warunki weryfikacji? Przykład podany w następnej sekcji powinien rzucić nieco światła na te pytania. Przechodząc od częściowej poprawności do zakończenia, naszym głównym interesem jest pokazanie, że w końcu wydarza się coś dobrego (nie, że złe rzeczy się nie zdarzają); mianowicie, że algorytm rzeczywiście osiąga swój punkt końcowy i kończy pomyślnie. Aby udowodnić takie stwierdzenie, używamy punktów kontrolnych jak poprzednio, ale teraz znajdujemy pewną ilość zależną od zmiennych algorytmu i struktur danych i pokazujemy, że jest ona zbieżna. Rozumiemy przez to, że ilość spada w miarę postępu egzekucji z jednego punktu

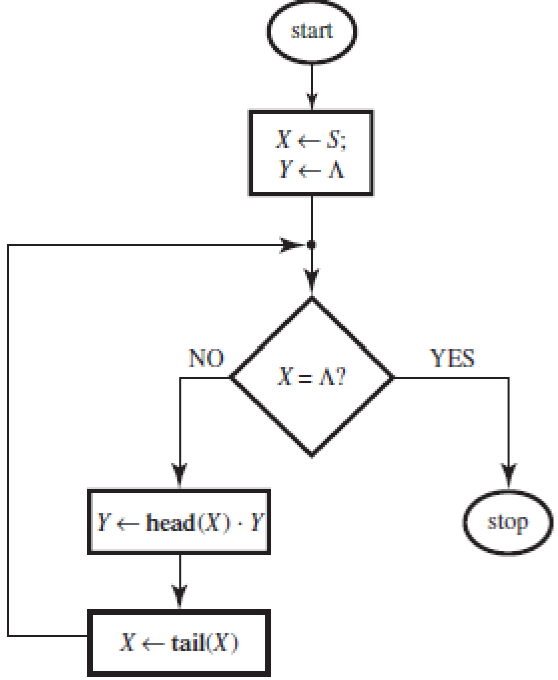
kontrolnego do drugiego, ale nie może się zmniejszać w nieskończoność – musimy pokazać, że istnieje pewna granica, poniżej której nigdy nie może zejść. Nie ma więc możliwości, aby algorytm działał w nieskończoność, ponieważ zbieżność, jak to się czasem nazywa, musiałaby wtedy zmniejszać się w nieskończoność, przecząc temu ograniczeniu. Na przykład w algorytmie sortowania liczba elementów, które nie znajdują się jeszcze na swoich końcowych pozycjach na posortowanej liście, może się zmniejszać w miarę postępu wykonywania, ale nigdy nie jest mniejsza niż 0. Gdy liczba ta osiągnie 0, algorytm przypuszczalnie się kończy. Jak wybrać takie zbieżności i jak pokazano, że są zbieżne? Ponownie przykład pomoże odpowiedzieć na te pytania.

Odwracanie ciągu symboli: Przykład

Prawnym wejściem do poniższego problemu jest ciąg znaków S składający się z symboli, powiedz słowo lub tekst w języku angielskim. Celem jest wytworzenie odwróconego obrazu S, oznaczonego jako reverse(S), składającego się z symboli S w odwrotnej kolejności. Na przykład:

reverse(“ajj$dt8”) = “8td$jja”

Rysunek 1 przedstawia prosty schemat blokowy algorytmu A, który rozwiązuje problem.



Używa unikalnego pustego ciągu , który nie zawiera żadnych symboli (i który możemy oznaczyć pustymi cudzysłowami „”), oraz funkcji head(X) i tail(X), które dla dowolnego ciągu X oznaczają , odpowiednio, pierwszy symbol X i ciąg X z usuniętą głową. Mamy więc:

head(“ajj$dt8”) = “a”

i :

tail(“ajj$dt8”) = “jj$dt8”

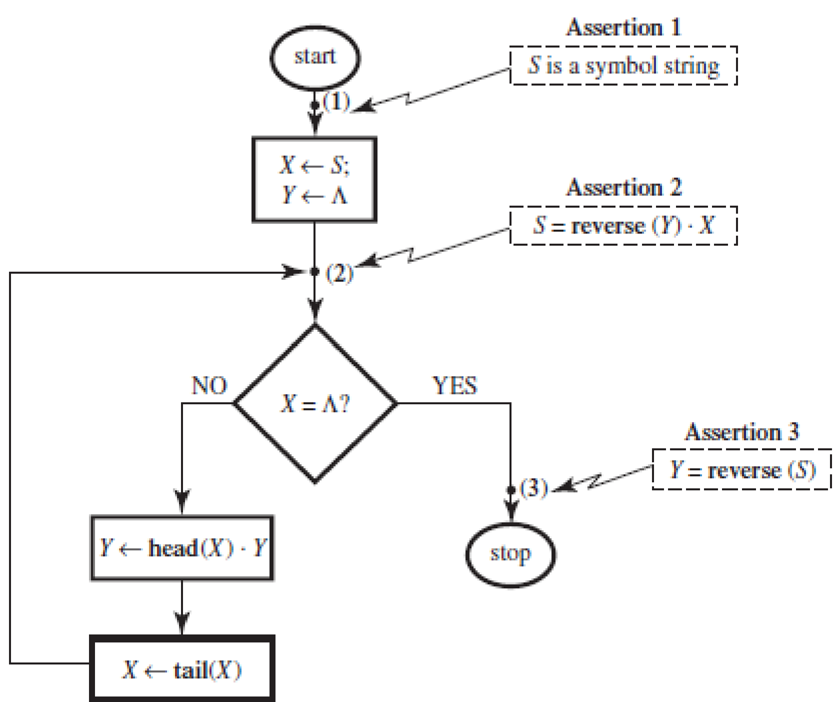
Używamy również specjalnego symbolu „·” dla konkatenacji ciągów lub załącznika. Zatem:

„ajj$dt8” · „tdd9tr” = „ajj$dt8tdd9tr”

W ten sposób będziesz w stanie łatwo zweryfikować, że przyczepienie głowy do ogona dowolnego sznurka daje sam sznurek. W symbolach:

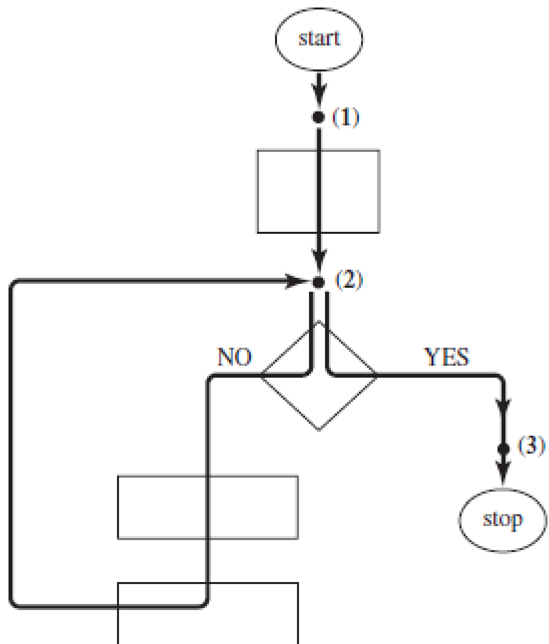
head(S) · tail(S) = S

Algorytm odwracania S polega na wielokrotnym „odklejaniu” symboli S jeden po drugim i dołączaniu każdego po kolei do początku nowo skonstruowanego ciągu Y . Nowy ciąg Y zaczyna się początkowo jako pusty. Procedura kończy się, gdy nie ma już nic do odklejenia S. Aby nie zniszczyć S w procesie, odklejanie odbywa się w zmiennej X, która jest inicjalizowana na wartość S. Twierdzi się, że ten algorytm poprawnie generuje odwrotność (S) w zmiennej Y . Oznacza to, że algorytm A z rysunku 1 jest całkowicie poprawny w odniesieniu do problemu algorytmicznego, który wymaga, aby wynik Y był odwróconym obrazem ciągu wejściowego S. Najpierw ustalimy, że A jest częściowo poprawny, używając metody asercji pośredniej , a następnie osobno, że również się kończy. Dlatego najpierw pokazujemy, że jeśli zdarzy się, że A zakończy się na łańcuchu wejściowym S, to wytworzy odwrotność(S) w Y . W tym celu rozważmy rysunek 2, na którym zidentyfikowano trzy punkty kontrolne.



Jak już wyjaśniono, twierdzenie 1 ujmuje wymagania dotyczące zbioru danych wejściowych, a twierdzenie 3 zawiera pożądaną zależność między łańcuchem wejściowym S i wyjściem Y , a mianowicie, że Y ma być równe odwrotności (S). Jednak znaczenie rysunku 2 jest w asercji 2, która ma uchwycić sytuację tuż przed ponownym obejściem pętli lub zakończeniem. Twierdzenie 2 stwierdza, że w punkcie kontrolnym (2) bieżące wartości X i Y razem tworzą oryginalny łańcuch S, w tym sensie, że Y zawiera pewną początkową część S w odwrotnej kolejności, a X zawiera resztę S nieodwróconą, co jest dokładnie to samo, co powiedzenie, że konkatenacja reverse(Y) z X daje S. Chcemy pokazać, że wszystkie trzy twierdzenia są niezmiennikami, co oznacza, że w każdym wykonaniu A na dowolnych legalnych danych wejściowych są one prawdziwe, gdy tylko zostaną osiągnięte. Sztuczka polega na

rozważeniu wszystkich możliwych „przeskoków" od punktu kontrolnego do punktu kontrolnego, które procesor może wykonać podczas wykonywania algorytmu. W tym przypadku, biorąc pod uwagę formę tego konkretnego schematu blokowego, możliwe są trzy przeskoki: punkt (1) do punktu (2), punkt (2) do punktu (3) i punkt (2) z powrotem do punktu (2).



Pierwszy z nich przechodzi dokładnie raz w każdym wykonaniu A; drugi jest pokonywany co najwyżej raz, ponieważ na jego końcu algorytm się kończy; trzeci można przemierzyć wiele razy (ile?). Zauważ, że segmenty algorytmiczne odpowiadające tym ścieżkom są wolne od pętli. Składają się z prostych sekwencji elementarnych instrukcji i testów i nie zawierają iteracji. Jak zobaczymy, oznacza to, że można sobie z nimi dość łatwo poradzić. Musimy teraz pokazać, że dla każdego z tych prostych segmentów, jeśli założymy, że twierdzenie dołączone do jego punktu początkowego jest prawdziwe i że odcinek jest faktycznie przebyty, to twierdzenie dołączone do jego punktu końcowego również będzie prawdziwe, gdy zostanie osiągnięte. Twierdzi się, że to wystarczy, aby ustalić częściową poprawność, ponieważ nastąpi niezmienność wszystkich trzech twierdzeń. Dlaczego? Cóż, powód jest zakorzeniony w fakcie, że każde legalne wykonanie A składa się z sekwencji segmentów oddzielonych punktami kontrolnymi

(1) ⟶ (2) ⟶ (2) ⟶ (2) ---⟶ (2) ⟶ (3)
start *stop*

W ten sposób, jeśli prawdziwość początkowego stwierdzenia każdego z tych segmentów implikuje prawdziwość końcowego twierdzenia i jeśli pierwsze twierdzenie całej sekwencji jest tym, które odpowiada legalnemu wejściu, a zatem zakłada się, że jest prawdziwe po pierwsze, prawdziwość twierdzeń rozprzestrzenia się w całej sekwencji, sprawiając, że wszystkie twierdzenia są prawdziwe w miarę postępu egzekucji. W szczególności, jak wyjaśniono, ostateczne Twierdzenie 3 będzie prawdziwe po rozwiązaniu, co stanowi częściową poprawność A. Podsumowując, musimy teraz pokazać, że prawdziwość twierdzeń rzeczywiście rozchodzi się naprzód po prostych ścieżkach między punktami kontrolnymi. Szczegóły tych dowodów nie zostaną tutaj przedstawione, ale zachęcamy do ich uzupełnienia. Pomaga jednak uważne zanotowanie, w symbolach, a nie słowach, tego, co dokładnie

ma być udowodnione. Patrząc na rysunki 2 i 3 i przypominając sobie trzy możliwe „przeskoki" między punktami kontrolnymi, okazuje się, że istnieją trzy stwierdzenia do udowodnienia:

(1 → 2): dla dowolnego ciągu S, po wykonaniu dwóch instrukcji X ← S; Y ← Λ,

zachowana zostanie równość S = reverse(Y) · X .

(2 → 3): if S = reverse(Y) · X, and X = Λ, then Y = reverse(S).

(2→2): if S = reverse(Y) · X, and X ≠ Λ, to po wykonaniu instrukcji

Y ←head(X) · Y ; X ←tail(X), ta sama równość, czyli S = reverse(Y) · X,

będzie obowiązywać dla nowych wartości X i Y . Formalne ustalenie, że te trzy stwierdzenia są prawdziwe, kończy dowód, że algorytm jest częściowo poprawny. Musimy teraz pokazać, że algorytm kończy się dla dowolnego ciągu wejściowego S. W tym celu ponownie rozważmy punkt kontrolny (2). Jedynym sposobem, w jaki wykonanie algorytmu może się nie zakończyć, jest nieskończenie częste przechodzenie przez punkt (2). Wykazano, że jest to niemożliwe przez wykazanie zbieżności (tj. wielkości zależnej od bieżących wartości zmiennych), która z jednej strony zmniejsza się za każdym razem, gdy punkt kontrolny (2) jest ponownie odwiedzany, ale z drugiej strony nie może stać się coraz mniejszy. Zbieżność, która działa w naszym przypadku, to po prostu długość łańcucha X. Za każdym razem, gdy pętla jest wykonywana, X jest skracane dokładnie o jeden symbol, ponieważ staje się on ogonem swojej poprzedniej wartości. Jednak jego długość nie może być mniejsza niż 0, ponieważ gdy X ma długość 0 (czyli staje się pustym ciągiem), pętla nie jest dalej przemierzana i algorytm się kończy. Na tym kończy się dowód, że algorytm odwrotny jest całkowicie poprawny. Może przyszło ci do głowy, że ten dowód nie jest wart zachodu, ponieważ poprawność programu wydaje się wystarczająco oczywista, a dowód wydaje się być żmudnie techniczny. Jest w tym trochę prawdy, a przykład został wybrany, aby zilustrować samą technikę dowodową, a nie potrzebę jej w tym konkretnym przykładzie. Niemniej jednak nietrudno wyobrazić sobie wersję tego samego algorytmu z jakimś subtelnym błędem w jednym z ekstremalnych przypadków lub wersję bardziej skomplikowaną, w której, powiedzmy, pozycje parzyste mają być odwrócone, a nieparzyste. nie. W takich przypadkach weryfikacja programu przez samo patrzenie jest niebezpiecznie nieodpowiednia a formalne dowody są koniecznością. Na marginesie, podczas weryfikacji poprawności nie musimy pracować na schematach blokowych. Chociaż wizualna natura schematu blokowego może czasami być pomocna w wyborze punktów kontrolnych i wnioskowaniu o dynamice algorytmu, istnieją proste sposoby dołączania pośrednich asercji do punktów w standardowych formatach tekstowych algorytmu. Nawet w tym małym przykładzie widać problematyczną naturę programowania imperatywnego. Wszystkie twierdzenia, które miały zostać udowodnione, zostały sformułowane w terminach różnych czasów, z rozróżnieniem między „nowymi" i „pierwotnymi" wartościami zmiennych. Napisanie tego algorytmu w funkcjonalnym języku programowania jest proste, a stwierdzenia do udowodnienia byłyby prostsze. Na przykład ostatni staje się:

(2 → 2): if S = reverse(Y) · X, and X ≠ Λ, then

S = reverse(head(X) · Y) · tail(X).

Nie rozwiązuje to zasadniczego problemu udowadniania poprawności, ale usuwa nieco kłopotliwy element jawnego zajmowania się zmianami wartości w czasie, co nie zawsze w naturalny sposób łączy się z naszymi matematycznymi oczekiwaniami dotyczącymi zmiennych.

Co zawiera dowód?

Dowód poprawności przedstawiony w ostatniej sekcji jest jednym z najprostszych w swoim rodzaju, nie będąc całkowicie trywialnym. Jest to co najmniej zniechęcające. Omówmy jego składniki. Podstawowym elementem zarówno częściowego dowodu poprawności, jak i zakończenia, jest wybór punktów kontrolnych w tekście algorytmu. Na ogół składają się one z punktów początkowych i końcowych oraz dostatecznie wielu lokalizacji pośrednich, aby każda pętla tekstu algorytmu zawierała przynajmniej jedną taką lokalizację. Po wybraniu punktów kontrolnych musimy dołączyć do nich asercje pośrednie, których niezmienność należy ustalić poprzez udowodnienie lokalnych warunków weryfikacji. Obejmują one tylko segmenty algorytmiczne wolne od pętli, ponieważ uważaliśmy, aby „otworzyć" wszystkie pętle z punktami kontrolnymi. Musimy również wykazać zbieżność i pokazać, że faktycznie jest zbieżna. Które z tych działań można zautomatyzować algorytmicznie? Innymi słowy, na co możemy oczekiwać znacznej pomocy od komputera? Znalezienie zestawu punktów kontrolnych do pokrycia każdej z pętli, nawet pewnego rodzaju zestawu minimalnego, może być w pełni zautomatyzowane. Co więcej, pod pewnymi warunkami technicznymi, wiele z lokalnych kontroli warunków weryfikacji, jak również lokalne zmniejszenie wartości zbieżności, może być również zautomatyzowane. Jednak sedno takich dowodów można znaleźć gdzie indziej. Polega na doborze odpowiednich niezmienników i zbieżności. Tutaj można wykazać, że nie istnieje żaden ogólny algorytm, który potrafiłby automatycznie znaleźć niezmienniki i zbieżności, które „działają", co oznacza, że spełniają lokalne warunki potrzebne do wytworzenia dowodu. Właściwy wybór niezmiennika jest sztuką delikatną i subtelną i może wymagać większej pomysłowości niż ta związana z zaprojektowaniem algorytmu. Rzeczywiście, projektant algorytmu może posiadać odpowiednią intuicję wymaganą do stworzenia dobrego algorytmu, ale może być zagubiony, gdy zostanie poproszony o precyzyjne sformułowanie „co się dzieje" w pewnym momencie. Paradoksalnie zawsze istnieją adekwatne niezmienniki i zbieżności, tak że, jak wspomniano wcześniej, poprawny algorytm można w zasadzie zawsze udowodnić. Wydaje się to przeczyć naszym twierdzeniom, że procesu weryfikacji nie można zautomatyzować, a projektant algorytmu może nie być w stanie udowodnić poprawności. To nie. A kluczem jest fraza „w zasadzie". Chociaż dowody istnieją, komputer nie zawsze może je znaleźć, a ludzie często też są zagubieni. W przypadku dużych i złożonych systemów oprogramowania weryfikacja po fakcie jest często niemożliwa, po prostu z powodu niewykonalnej wielkości i złożoności zadania. W przypadku przedstawienia dużego oprogramowania, które było nieustannie zmieniane, poprawiane, łatane i aktualizowane, dostarczanie formalnych i precyzyjnych twierdzeń, które całkowicie charakteryzują jego zachowanie w różnych punktach, jest zasadniczo wykluczone. W takich przypadkach należy zastosować alternatywną metodę, zwaną weryfikacją as-you-go, która zostanie omówiona później.

Wieże Hanoi: Przykład

Niezależnie od powyższej dyskusji, weryfikacja jest czasami nieco łatwiejsza niż oczekiwano. Mogłoby się wydawać, że subtelna natura rekurencji może utrudnić weryfikację podprogramów rekurencyjnych niż zwykłych algorytmów iteracyjnych. Nie zawsze tak jest. Przypomnij sobie problem Wież Hanoi i następujące rozwiązanie rekurencyjne, przedstawione w Części 2:

podprogram przesuń N z X na Y za pomocą Z:

(1) jeśli N wynosi 1, to wypisz „przenieś X do Y";

(2) w przeciwnym razie (to znaczy, jeśli N jest większe niż 1) wykonaj następujące czynności:

(2.1) wywołaj przeniesienie N − 1 z X do Z za pomocą Y ;

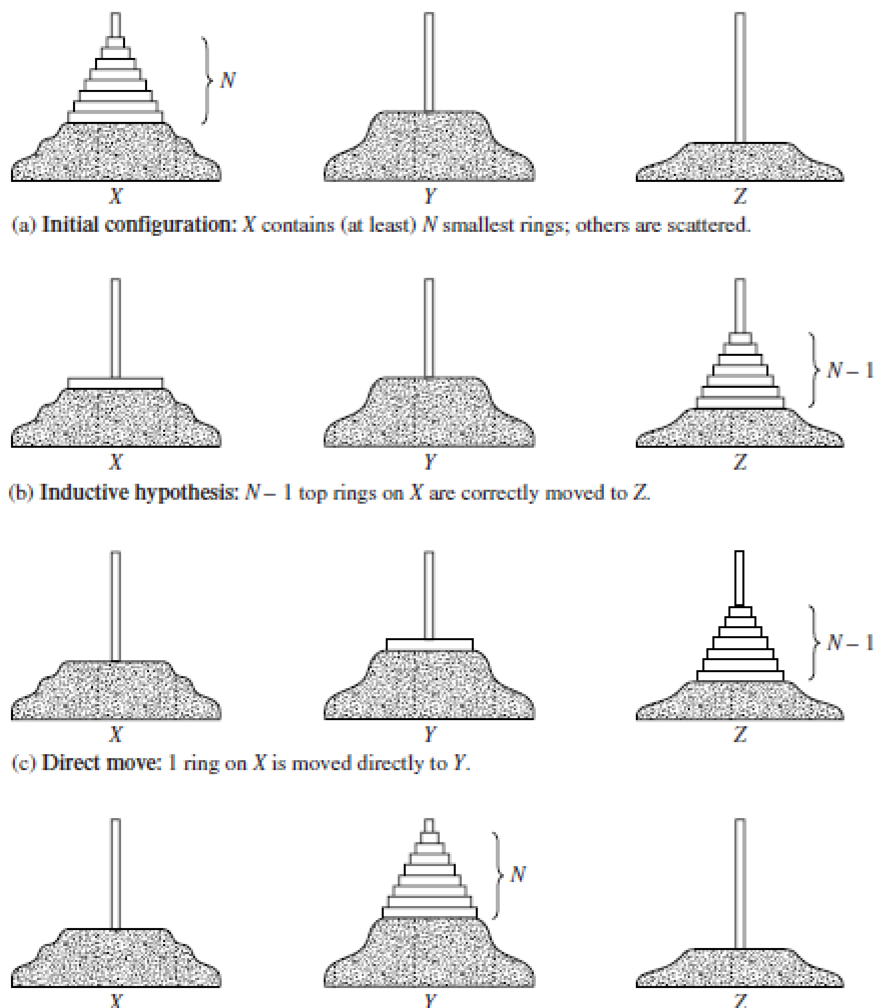
(2.2) wyjście „przesuń X do Y";

(2.3) wywołanie przesunięcia N − 1 z Z do Y przy użyciu X;

(3) powrót.

To, że ta procedura kończy się dla każdego N (gdzie wartości X, Y i Z to A, B i C w pewnej kolejności) wydaje się dość łatwe do zauważenia: jedyny możliwy rodzaj nieskończonego obliczenia jest uzyskiwany przez wykonanie nieskończonego głęboka kaskada wywołań rekurencyjnych. Jednak obserwując, że głębokość drzewa wywołań rekurencyjnych nie może być większa niż N, ponieważ jedyne, co dzieje się z N podczas wykonywania, to spadek o 1 za każdym razem, gdy wykonywane jest wywołanie rekurencyjne niższego poziomu, stwierdzamy, że N musi „ trafić" w pewnym momencie na wartość 1. Ale gdy N jest dokładnie 1, klauzula ucieczki rekurencji zostaje osiągnięta, powodując nie kolejne, głębsze wywołanie rekurencyjne, ale prostą instrukcję, po której następuje powrót, co pociąga za sobą wznoszenie drzewa. W konsekwencji drzewo wywołań rekurencyjnych jest skończone i wykonanie musi się zakończyć. Zauważ, że ten dowód zakończenia nie wymaga żadnego „zrozumienia" działania procedury; wykorzystaliśmy tylko powierzchowne obserwacje dotyczące zachowania N i obecności odpowiedniej klauzuli korekcyjnej. Jednak dowód nie jest do końca słuszny! Jeśli początkowa wartość N wynosi zero lub mniej, można ją zmniejszyć nieskończoną liczbę razy bez uderzania w 1, a procedura rzeczywiście nie zakończy się, jeśli otrzyma taką wartość dla N. To, co musimy tutaj zrobić, to dodać do ta procedura specyfikacja legalnych danych wejściowych, które wykluczają niedodatnie wartości N. (Co ciekawe, porównanie tego algorytmu z wersją podaną w PROLOGU w części 3 pokazuje, że wersja PROLOGa w rzeczywistości obsłuży przypadek N = 0 poprawnie.) Aby udowodnić częściową poprawność, używamy wariantu pośredniej metody asercji, który pasuje do nieiteracyjnego charakteru algorytmów rekurencyjnych. Zamiast próbować sformułować lokalną sytuację w danym punkcie, staramy się sformułować nasze oczekiwania dotyczące całej procedury rekurencyjnej tuż przed jej wejściem. Jest to następnie używane w cyklicznie wyglądającym, ale doskonale zdrowym stylu, aby się utrzymać! Oto możliwe sformułowanie rutyny ruchu. Dla dowolnego N prawdziwe jest następujące stwierdzenie, nazwij je (S):

Załóżmy, że nazwy kołków A, B i C są skojarzone w pewnej kolejności ze zmiennymi X, Y i Z. Następnie kończące wykonanie wywołania przenosi N z X na Y za pomocą Z wymienia sekwencję pierścieni. instrukcje przemieszczania, które, jeśli zostaną uruchomione (i będą wiernie przestrzegane) w dowolnej legalnej konfiguracji pierścieni i kołków, w której co najmniej N najmniejszych pierścieni jest na kołku X, prawidłowo przenosi te N pierścieni z X na Y, prawdopodobnie używając Z jako tymczasowego przechowywania . Co więcej, sekwencja jest zgodna z zasadami problemu Wież Hanoi i pozostawia nietknięte wszystkie inne pierścienie. Jeśli teraz możemy pokazać, że (S) jest prawdziwe dla wszystkich N, ustalimy częściową poprawność naszego rozwiązania, ponieważ szczególnie interesuje nas wywołanie przeniesienia N z A do B za pomocą C, gdzie A zawiera N pierścieni i B i C są puste. W tym przypadku (S) jest tylko przeformułowaniem wymagań problemu, co możesz zweryfikować. Zdanie (S) można ustalić klasyczną metodą indukcji matematycznej. Oznacza to, że najpierw pokazujemy bezpośrednio, że zdanie jest prawdziwe, gdy N wynosi 1, a następnie pokazujemy, że przy założeniu, że jest prawdziwe dla pewnego danego N − 1 (gdzie N − 1 wynosi co najmniej 1, więc N wynosi co najmniej 2), musi to być również prawdziwe dla samego N. Wynika z tego, że (S) musi być prawdziwe dla wszystkich N, ponieważ oddzielnie udowodniona prawda dla przypadku N = 1 implikuje prawdę dla N = 2, co z kolei implikuje prawdę dla N = 3 itd., ad nieskończoność. Przyjrzyjmy się teraz uważnie szczegółom dowodu.



(a) **Initial configuration:** *X* contains (at least) *N* smallest rings; others are scattered.

(b) **Inductive hypothesis:** *N* – 1 top rings on *X* are correctly moved to *Z*.

(c) **Direct move:** 1 ring on *X* is moved directly to *Y*.

(d) **Inductive hypothesis:** *N* – 1 top rings on *Z* are correctly moved to *Y*.

To (S) obowiązuje, gdy N wynosi 1, jest trywialne: stwierdzenie (S) w tym przypadku jest po prostu stwierdzeniem, że kiedy podprogram zostanie wywołany, gdy N wynosi 1, tworzona jest sekwencja, która prawidłowo przenosi najwyższy pierścień z X na Y . Od razu widać, że zostało to osiągnięte przez pierwszą linię podprogramu. Załóżmy teraz, że zdanie (S) obowiązuje dla pewnego arbitralnego N − 1. Teraz musimy pokazać, że dotyczy ono również dla N. W związku z tym załóżmy, że wywołanie procedury ruchu zostało właśnie wykonane z liczbą N i pewnym powiązanie kołków A, B i C ze zmiennymi X, Y i Z. Trzy kołki zawierają pewien legalny układ pierścieni, przy czym kołek X zawiera co najmniej N najmniejszych pierścieni (patrz (a)). Teraz, ponieważ wartość N nie wynosi 1, pierwszą rzeczą, jaką robi podprogram, jest wywołanie siebie rekurencyjnie w linii (2.1) z parametrem N − 1. Zgodnie z hipotezą indukcyjną, to znaczy zakładając, że (S) obowiązuje dla wywołania do procedury z N − 1, to wywołanie, jeśli zostanie zakończone pomyślnie, poprawnie i legalnie przesunie najwyższe N − 1 dzwonków z X na Z, pozostawiając wszystko inne bez zmian (patrz (b)). Jednakże, ponieważ X miał gwarancję, że na początku będzie zawierało co najmniej N dzwonków, po zakończeniu połączenia on-line (2.1) na X nadal pozostaje co najmniej jeden dzwonek, a instrukcja online (2.2) przenosi ten dzwonek bezpośrednio do Y (patrz (c)). Ponieważ wszystkie pierścienie na Y przed tym ruchem (jeśli były) musiały być większe niż N-ty poruszany pierścień, ten ruch jest prawidłowy. To samo założenie dotyczące N − 1, ale teraz zastosowane do następnego wywołania, w linii (2.3) podprogramu, można

zobaczyć, aby uzupełnić obraz, przesuwając pierścienie N − 1 z Z do Y, tuż nad pojedynczym pierścień, który został przeniesiony oddzielnie (patrz (d)). Powinieneś dokładnie przeanalizować ten dowód, zauważając, dlaczego legalność wszystkich ruchów jest również gwarantowana przez cały proces, a nie tylko ostateczny wynik. Możliwe jest również wykazanie (używając tej samej hipotezy indukcyjnej dla dwóch wywołań), że żadne dzwonki inne niż górne N na kołku X nie są dotykane tą procedurą, tak że (S) zostało ustalone w całości dla tego wywołania z N, zakładając to dla wywołań z N − 1. Jak już wspomniano, ustala to, że zdanie (S) obowiązuje dla wszystkich N. Częściowa poprawność rekurencyjnego algorytmu Wież Hanoi jest w ten sposób udowodniona, a ponieważ już udowodniliśmy zakończenie, algorytm jest całkowicie poprawny.

Więcej o wieżach Hanoi: proste rozwiązanie iteracyjne

W Części 2 obiecaliśmy zaprezentować niezwykle prosty do wykonania algorytm iteracyjny dla problemu Wieże Hanoi. Powód omawiania tego tutaj, a nie w części 2, wynika z faktu, że nie jest do końca jasne, dlaczego to działa, a fakt, że to działa, wymaga dowodu. Rzeczywiście, nie będziemy tutaj przedstawiać dowodu jego poprawności, a zachęcamy do próby skonstruowania dowodu, pokazując, że algorytmy iteracyjne i rekurencyjne są naprawdę równoważne. Można to zrobić przez indukcję na N. Aby opisać algorytm, załóżmy, że trzy kołki są ułożone w okrąg i że wszystkie pierścienie N są ułożone na jednym z nich (nazwy kołków są nieistotne). Oto algorytm:

(1) wykonaj następujące czynności wielokrotnie, aż przed krokiem (1.2) wszystkie pierścienie zostaną prawidłowo ułożone na innym kołku:

(1.1) przesuń najmniejszy pierścień z obecnego kołka na następny kołek w kolejności zgodnej z ruchem wskazówek zegara;

(1.2) sprawiają, że jedyny możliwy ruch, który nie obejmuje najmniejszego pierścienia.

Powinno być jasne, że krok (1.2) jest dobrze zdefiniowany i jednoznaczny, ponieważ jeden z kołków musi mieć najmniejszy pierścień na górze, a z dwóch pozostałych kołków jeden ma mniejszy pierścień na górze niż drugi; stąd jedynym ruchem, który nie obejmuje najmniejszego pierścienia, jest przeniesienie tego mniejszego pierścienia na drugi kołek. Ten algorytm może być łatwo wykonany przez małe dziecko, nawet jeśli w grę wchodzi wiele dzwonków. Zauważ mimochodem, że jeśli N jest nieparzyste, pierścienie wylądują na następnym kołku w kolejności zgodnej z ruchem wskazówek zegara, a jeśli N jest parzyste, trafią na następny kołek w kolejności przeciwnej do ruchu wskazówek zegara.

Weryfikacja po fakcie a weryfikacja na bieżąco

Alternatywą dla udowodnienia poprawności już ukończonego algorytmu jest wspólne opracowanie algorytmu i dowodu. Chodzi o to, aby dopasować zdyscyplinowaną, stopniową, krok po kroku konstrukcję algorytmu lub systemu oprogramowania do stopniowej konstrukcji segmentów dowodu, które ostatecznie skumulują się, tworząc kompletny dowód całego systemu. Najwyraźniej łatwiej to powiedzieć niż zrobić. Jednak oczywistym mechanizmem, który zachęca do takiej praktyki, jest podprogram. Jak wyjaśniono w części 2, złożony program można starannie zbudować z wielu procedur, z których niektóre są zagnieżdżone w innych, co skutkuje dobrze ustrukturyzowanym i rozwarstwionym oprogramowaniem. Dobra praktyka projektowa nakazuje, aby każda procedura była dokładnie analizowana pod kątem jej ogólnego przeznaczenia, a następnie weryfikowana jako oddzielna całość. W zasadzie po wykonaniu tej czynności cały system również zostanie zweryfikowany. Powodem jest to, że to, co zostało udowodnione o danej procedurze, może być użyte do udowodnienia rzeczy o innych procedurach, które to nazywają. Jeśli, na przykład, zweryfikujemy, że procedura wyszukiwania

z Części 2 poprawnie odnajduje X w tekście wejściowym i wyprowadza licznik, jeśli dojdzie do końca tekstu, wówczas ten dowód może zostać użyty do zweryfikowania znajdowania „pieniędzy”. algorytm. Tutaj również istnieje wiele możliwych pułapek. Nie jest łatwo poprawnie zidentyfikować interfejs procedury i jej zamierzone zachowanie w każdych okolicznościach, ale dobry projekt algorytmiczny zaleca, abyśmy nie tworzyli podprogramu, chyba że możemy to zrobić. Oczywiście procedura rekurencyjna wymaga cyklicznego procesu weryfikacji, który obejmuje założenie, że jest poprawny dla własnych wywołań, tak jak zrobiono to w przypadku Wież Hanoi. Oczywiście ten proces nie zawsze jest tak prosty, jak się wydaje. Krótko mówiąc, dobry krokowy projekt algorytmiczny, w połączeniu z informacyjną i precyzyjną dokumentacją decyzji projektowych, może stanowić podstawę konstrukcji zweryfikowanego oprogramowania, poprzez rozbicie ogólnego dowodu na części, które odzwierciedlają rozpad samego oprogramowania. Kiedy współbieżność i reaktywność są wprowadzane do algorytmów, jak to ma miejsce w częściach 10 i 14, sprawy stają się trudniejsze, a nawet mały i niewinnie wyglądający algorytm może powodować ogromne problemy, jeśli chodzi o weryfikację. Porozmawiamy o tym moście, ale nie do końca go przekroczymy, kiedy do niego dotrzemy.

Projekt według umowy

Paradygmat zorientowany obiektowo przenosi ideę bieżącej weryfikacji o krok naprzód, w formie metodologii zwanej projektowaniem na podstawie umowy. Jak wspomniano w części 3, każda klasa w programie zorientowanym obiektowo opisuje pewien abstrakcyjny zestaw obiektów z powiązanymi z nimi zachowaniami lub metodami. Taki opis może zawierać instrukcje, jak przeprowadzić każdą metodę, ale nie musi. W rzeczywistości często bardzo przydatne jest posiadanie całkowicie abstrakcyjnych klas, które określają tylko, co należy zrobić, ale nie jak. Wcielenie JAVA klasy Queue jest takim przykładem, w którym nie określa zachowania każdej metody, a jedynie sposób jej wywołania. Co więcej, zastępując słowo „Kolejka” słowem „Stos” otrzymalibyśmy całkowicie legalny opis stosów zamiast kolejek. Brakuje znaczenia każdej metody: jak wpływa ona na stan obiektu i co (jeśli w ogóle) zwraca. W podejściu projektowania po kontrakcie specyfikacja ta nazywana jest kontraktem i składa się z trzech rodzajów stwierdzeń: niezmienników klasy, warunków wstępnych metody i warunków końcowych metody. Niezmienniki klasy określają warunki, które muszą być spełnione dla każdego obiektu klasy przed i po wykonaniu każdej operacji. Warunki wstępne metody określają legalne dane wejściowe do każdej metody (lub innymi słowy, co musi być przechowywane przed wykonaniem metody), a warunki końcowe metody określają, co musi być przechowywane po wykonaniu metody. W kontekście poprzednich dyskusji w tej części, warunki wstępne metody i warunki końcowe są tak naprawdę tylko twierdzeniami na początku i na końcu procedur, a niezmienniki klasy są analogiczne do niezmienników pętli. Na przykład w klasie Queue warunkiem wstępnym dla metody front byłoby to, że kolejka nie jest pusta. Warunkiem końcowym add(X) jest to, że front zwróci X, jeśli kolejka była pusta przed wywołaniem add, w przeciwnym razie zwróci tę samą wartość, którą zwróciłby przed tą operacją. Natomiast warunek końcowy dla odpowiedniej metody w klasie Stack powiedziałby, że po wywołaniu metody add(X), front zwróci X, niezależnie od poprzedniego stanu obiektu. Pozwala to uchwycić istotną różnicę między stosami a kolejkami. Metodologia projektowania po kontrakcie wymaga sprecyzowania kontraktu przed napisaniem programu wdrożeniowego oraz starannego modyfikowania kontraktu w miarę zmian specyfikacji. Stąd już tylko krok do weryfikacji programu, a pojawiają się narzędzia, które mogą w tym pomóc. Projektowanie według umowy jest integralną cechą języka programowania EIFFEL, który obsługuje również niezmienniki pętli i konwergencję. Chociaż nie jest to jeszcze część oficjalnego języka JAVA, istnieje wiele narzędzi, które dodają tę możliwość również do JAVA. Pozwalają one kompilatorowi generować kod, który faktycznie sprawdza asercje podczas działania programu, ostrzegając programistę o napotkanych naruszeniach. Jednak chociaż takie narzędzia są przydatne do testowania programów, głównym wkładem metody projektowania według umowy jest zwracanie uwagi na poprawność programu podczas jego pisania i modyfikowania. W

związku z tym może być praktykowane nawet przez programistów używających języków obiektowych, które nie mają jeszcze obsługi narzędzi do sprawdzania asercji w czasie wykonywania. Określanie zachowania funkcji lub podprogramów za pomocą warunków wstępnych i warunków końcowych jest możliwe w każdym języku programowania. Zasadniczo istnieją dwa powody, dla których projektowanie na podstawie umowy jest szczególnie odpowiednie dla paradygmatu zorientowanego obiektowo. Po pierwsze, podział programu na klasy i metody oraz możliwość klas abstrakcyjnych dostarcza naturalnych punktów do umieszczania asercji. Po drugie, styl zorientowany obiektowo daje realną obietnicę możliwości ponownego wykorzystania kodu, co oznacza możliwość używania tej samej klasy – kodu implementującego i wszystkiego – w wielu różnych aplikacjach. Jednak, aby to zadziałało, niezbędne jest jasne określenie, co mają oznaczać klasy i metody. Bez tego programista próbujący ponownie wykorzystać czyjąś klasę może popełniać takie błędy, jak mylenie kolejki ze stosem. Spektakularnym tego przykładem jest utrata pierwszego startu Ariane 5 w 1996 roku. Awarię modułu nawigacyjnego przypisuje się praktyce ponownego wykorzystywania programów z Ariane 4 bez zrozumienia założeń, na których opierały się oryginalne programy; te założenia nie były już prawdziwe dla Ariane 5, ale nie zostały udokumentowane w kodzie.

Interaktywna weryfikacja i kontrola dowodowa

Pomimo tego, że generalnie automatyczna weryfikacja algorytmiczna nie wchodzi w rachubę, przy pomocy komputera można wiele zrobić. W tym miejscu przeanalizujemy tylko kilka kwestii, ponieważ większość z nich jest zbyt techniczna, aby można ją było tutaj szczegółowo omawiać. Komputer można zaprogramować do pomocy w weryfikacji po fakcie. Możliwy scenariusz to interaktywny program, który próbuje zweryfikować dany algorytm pod kątem formalnego opisu problemu algorytmicznego. Od czasu do czasu prosi użytkownika, aby dostarczył mu, powiedzmy, niezmiennik kandydata (przypomnij sobie, że jest to zadanie, którego na ogół nie może wykonać samodzielnie). Następnie kontynuuje i próbuje ustalić odpowiednie warunki weryfikacji, a następnie przechodzi do dodatkowych punktów w algorytmie, jeśli się powiedzie, lub cofa do poprzednich decyzji, jeśli się nie powiedzie. Ponieważ cała procedura jest interaktywna, użytkownik może ją zatrzymać, gdy wydaje się, że idzie w złym kierunku, i spróbować skierować ją z powrotem na właściwe tory. Komputer jest tutaj używany jako szybki i skrupulatny praktykant, sprawdzający szczegóły lokalnych warunków logicznych, śledzący przeszłe niezmienniki, twierdzenia i komentarze, i nigdy niestrudzony w wypróbowywaniu innej możliwości. Taki proces może również prowadzić do odkrycia ukrytych błędów lub brakujących założeń. Na przykład skomputeryzowany weryfikator nie zaakceptuje podanego wcześniej błędnego dowodu zakończenia programu Wieże Hanoi. Zbadanie przyczyny odrzucenia dowodu doprowadziłoby programistę do uświadomienia sobie, że brakowało odpowiedniego warunku wstępnego. Ten pomysł może być również wykorzystany do pomocy w działaniach weryfikacyjnych na bieżąco. Tutaj, wcześniej ustalone dowody części algorytmu (powiedzmy, podprogramów) mogą być śledzone przez automatycznego ucznia i użyte w algorytmicznych próbach weryfikacji większych porcji (powiedzmy, wywoływanie podprogramów). Taki interaktywny system może być częścią całego środowiska programistycznego, które może obejmować również narzędzia do edycji, debugowania i testowania. Użytkownik może być zainteresowany weryfikacją niektórych mniejszych i lepiej zidentyfikowanych algorytmów w opracowywanym systemie oprogramowania i pozostawić mniej łatwe w zarządzaniu części do konwencjonalnego debugowania i testowania. Inną możliwością weryfikacji wspomaganej komputerowo jest sprawdzanie dowodu. Tutaj człowiek wytwarza coś, co wydaje się dowodem – niezmienniki, zbieżności i wszystko – a komputer jest zaprogramowany do generowania i weryfikacji długich sekwencji logicznych i symbolicznych manipulacji, które składają się na pełnoprawny dowód. Temu tematowi poświęcono i nadal poświęcono wiele pracy, istnieje kilka systemów, które mogą pomóc w projektowaniu i weryfikacji nietrywialnych programów. Jak omawiamy w częściach 10 i 14, potężne metody (wiele z nich opiera się na technice zwanej sprawdzaniem modelu) zostały

opracowane do weryfikacji nawet złożonych systemów oprogramowania i sprzętu. Przy pewnych rozsądnych założeniach, takich jak skończona liczba stanów zachowania systemu i proste, dobrze zdefiniowane właściwości do weryfikacji, działają one całkiem dobrze w praktyce. Są więc zachęcające znaki. Wciąż jednak pozostaje wiele problemów, a stosowanie sprawnych i szeroko stosowanych pomocy weryfikacyjnych jeszcze trochę potrwa, zanim stanie się powszechną praktyką.

Badania poprawności algorytmicznej

Oprócz tematów omówionych w poprzednim częściach, badacze są zainteresowani opracowaniem nowych i użytecznych metod weryfikacji, a metody niezmienne i zbieżne są tylko przykładami najbardziej podstawowych z nich. Różne konstrukcje językowe uruchamiają różne metody, co stanie się widoczne po wprowadzeniu współbieżności i probabilizmu w Częściach 10 i 11. Jedną z kwestii, której w ogóle nie zajęliśmy się, jest kwestia języków specyfikacji, czasami zwane językami asercyjnymi. Jak formalnie określamy problem algorytmiczny? Jak zapisać twierdzenia pośrednie, aby były jednoznaczne i poddawały się manipulacji algorytmicznej? Alternatywy różnią się nie tylko pragmatyzmem, ale także podstawową matematyką. W przypadku niektórych języków asercji ustalenie lokalnych warunków weryfikacji segmentów pozbawionych pętli między punktami kontrolnymi jest algorytmicznie wykonalne, podczas gdy w przypadku innych można wykazać, że problem jest tak trudny, jak sam globalny problem weryfikacyjny, a zatem nie można go rozwiązać algorytmicznie. W obu przypadkach ludzie są zainteresowani rozwojem technik dowodzenia twierdzeń, które stanowią podstawę algorytmów sprawdzania warunków. Wiele tematów badawczych dotyczy zarówno weryfikacji i debugowania, jak i wydajnej i pouczającej kompilacji. Dobrym przykładem jest analiza przepływu danych, w ramach której opracowywane są metody symbolicznego śledzenia przepływu danych w algorytmie. Oznacza to, że możliwe zmiany, jakie zmienna może przejść podczas wykonywania, są analizowane bez faktycznego uruchamiania algorytmu. Taka analiza może ujawnić potencjalne błędy, takie jak indeksy wykraczające poza granice tablic, możliwe wartości zerowe dla dzielników w operacjach arytmetycznych i tak dalej. Badaczy interesują także bardziej złożone właściwości algorytmów. Czasami może być ważne, aby wiedzieć, czy dwa algorytmy są równoważne. Jednym z powodów może być dostępność prostego, ale nieefektywnego algorytmu oraz kandydata projektanta na bardziej wydajny, ale bardziej skomplikowany algorytm. Pragniemy tutaj udowodnić, że oba algorytmy zakończą się na tej samej klasie danych wejściowych i dadzą identyczne wyniki. Jednym z podejść do badania tak bogatszych klas własności algorytmicznych jest formułowanie logiki programów lub logik algorytmicznych, które są analogiczne do klasycznych systemów logiki matematycznej, ale które umożliwiają wnioskowanie o algorytmach i ich skutkach. Podczas gdy systemy klasyczne są czasami nazywane statycznymi, logiki algorytmiczne mają charakter dynamiczny; prawdziwość stwierdzeń zawartych w tych formalizmach zależy nie tylko od obecnego stanu świata (jak w stwierdzeniu „pada deszcz”), ale od relacji między stanem obecnym a innymi możliwymi. Jako przykład rozważ podstawową konstrukcję takiej dynamicznej logiki:

after (A, F)

co oznacza, że po wykonaniu algorytmu A, twierdzenie F z konieczności będzie prawdziwe. Twierdzenie F może stwierdzać, że lista L jest posortowana, w którym to przypadku instrukcja może być użyta do sformalizowania częściowej poprawności procedury sortującej. Siła takiego konstruktu tkwi jednak w możliwości wielokrotnego użycia go w wypowiedzi i połączenia z innymi obiektami logicznymi. Rozważmy następujące, gdzie „→” oznacza „implikacje”:

jeśli F1 → after (A, F2) i F2 → after (B, F3)

wtedy F1 → after (A; B, F3)

Stwierdzenie to stwierdza, że jeśli (1) zawsze, gdy F1 jest prawdziwe, F2 jest prawdziwe po wykonaniu A, oraz jeśli także (2) zawsze, gdy F2 jest prawdziwe, F3 jest prawdziwe po wykonaniu B, to możemy wywnioskować (3) zawsze, gdy F1 jest prawdziwe, F3 jest prawdziwe po wykonaniu A; B (czyli A, po którym następuje bezpośrednio B). To stwierdzenie może wydawać się oczywiste, ale jest dość subtelne. W rzeczywistości stanowi matematyczne uzasadnienie wstawiania twierdzeń pośrednich do algorytmu. F2 można traktować jako twierdzenie pośrednie dołączone do punktu oddzielającego A od B, a stwierdzenie to mówi, że ustalenie warunków lokalnych na A i B oddzielnie sprowadza się do ustalenia bardziej globalnego warunku na A; B. W takiej logice można sformułować (i udowodnić) bardziej skomplikowane stwierdzenia dotyczące równoważności i zakończenia programów oraz innych interesujących właściwości. Logika programów pomaga zatem oprzeć teorię weryfikacji algorytmicznej na solidnych podstawach matematycznych. Ponadto pozwalają nam badać pytania dotyczące możliwości zautomatyzowania metod sprawdzania terminacji, poprawności, równoważności i tym podobnych. Jedna z najskuteczniejszych metod weryfikacji, zwana modelem sprawdzania może być użyta do sprawdzenia programu pod kątem formuły logicznej, ale tutaj również istnieją nieodłączne ograniczenia takiego wysiłku, które zostaną omówione w Częściach 7 i 8. Ponadto Część 10 omawia weryfikację systemów współbieżnych, a Część 12 omawia ogólne dodawanie interakcji do dowodów matematycznych, a także dowody, które są zarówno interaktywne, jak i probabilistycznie sprawdzalne. Części 13 i 14, które omawiają duże i złożone systemy, również dotykają niektórych kwestii poprawności. Inny obiecujący kierunek badań dotyczy automatycznej lub półautomatycznej syntezy algorytmicznej. W tym przypadku przedmiotem zainteresowania jest synteza działającego algorytmu lub programu na podstawie specyfikacji problemu algorytmicznego. Podobnie jak w przypadku weryfikacji, ogólny problem syntezy jest algorytmicznie nierozwiązywalny, ale części procesu mogą być wspomagane przez komputer. Jednym z problemów związanych zarówno z syntezą, jak i dowodami równoważności jest kwestia transformacji programu, w której części algorytmu lub programu są przekształcane w sposób zapewniający zachowanie równoważności. Na przykład możemy być zainteresowani takimi przekształceniami, aby algorytm był zgodny z zasadami jakiegoś języka programowania (na przykład zastąpienie rekurencji przez iterację, gdy język nie pozwala na rekurencję) lub w celach wydajnościowych, jak pokazano w Części 6. Badania w zakresie poprawności i logiki algorytmów ładnie komponują się z badaniami nad semantyką języków programowania. Nie da się udowodnić czegokolwiek na temat programu bez rygorystycznego i jednoznacznego znaczenia dla tego programu. Im bardziej złożone są używane przez nas języki, tym trudniej jest zapewnić im semantykę i tym trudniej wymyślić metody dowodowe i zbadać je za pomocą logik algorytmicznych. Jest to kolejny powód, dla którego języki funkcjonalne są bardziej podatne na dowody poprawności niż języki imperatywne.

Twierdzenie o czterech kolorach

Wydaje się właściwe zamknięcie tej części opowieścią o pewnym filozoficznym znaczeniu. Problem czterokolorowy został sformułowany w 1852 roku i przez około 120 lat uważany był za jeden z najciekawszych otwartych problemów w całej matematyce. Obejmuje mapy, takie, jakie można znaleźć w atlasie: diagramy składające się z podziałów skończonej części płaszczyzny przez zamknięte regiony oznaczające kraje. Załóżmy, że chcemy pokolorować taką mapę, przypisując kolor każdemu krajowi, ale w taki sposób, aby żadne dwa kraje, które dzielą część granicy, nie były pokolorowane tym samym kolorem. Ile kolorów jest potrzebnych do pokolorowania dowolnej mapy? Na pierwszy rzut oka wydaje się, że możemy konstruować coraz bardziej skomplikowane mapy, wymagające coraz większej liczby kolorów. Kiedy jednak pobawimy się kilkoma przykładami, okazuje się, że zawsze wystarczą cztery kolory. Problem czterech kolorów pyta, czy to zawsze prawda. Z jednej strony nikt nie był w stanie udowodnić, że cztery kolory były wystarczające do pokolorowania dowolnej mapy, z drugiej strony nikt nie był w stanie wykazać mapy wymagającej pięciu. Przez lata nad tym problemem pracowało wiele

osób, a w dziedzinie matematyki zwanej topologią pojawiło się wiele głębokich i pięknych wyników. Opublikowano kilka „dowodów", że cztery kolory wystarczą, ale później okazało się, że zawierają one subtelne błędy. W 1976 roku problem został ostatecznie rozwiązany przez dwóch matematyków. Udowodnili to, co jest obecnie znane jako twierdzenie o czterech kolorach, które twierdzi, że cztery kolory rzeczywiście wystarczą. Co to ma wspólnego z nami? Cóż, dowód z 1976 roku został osiągnięty przy pomocy komputera. Dowód można uznać za składający się z grubsza z dwóch części. W pierwszym, dwaj badacze wykorzystali kilka wcześniej ustalonych wyników, z pewnym dodatkowym rozumowaniem matematycznym, aby udowodnić, że ogólny problem można sprowadzić do wykazania, że skończoną liczbę szczególnych przypadków można pokolorować czterema kolorami. Napisano wówczas programy komputerowe, które miały skrupulatnie generować wszystkie takie przypadki (okazało się, że jest ich około 1700) i przejrzeć je wszystkie w celu znalezienia czterokolorowości. Twierdzenie zostało ustalone, gdy programy się zakończyły, odpowiadając pozytywnie na pytanie: wszystkie 1700 przypadków okazało się czterokolorowych. Czy możemy umieścić tradycyjne Q.E.D. (oznaczające quod erat demonstrandum, swobodnie tłumaczone jako „to, co miało być udowodnione") na końcu dowodu? Problem w tym, że nikt nigdy nie zweryfikował programów użytych w dowodzie. Możliwe, że algorytmy skonstruowane w celu przeprowadzenia subtelnego generowania przypadków były wadliwe. Mało tego, nikt jeszcze nigdy nie zweryfikował poprawności kompilatora użytego do tłumaczenia programów na kod wykonywalny maszynowo. Nikt nie zweryfikował żadnego z innych odpowiednich programów systemowych, które mogłyby wpłynąć na prawidłowe działanie programów, a z tego powodu (choć nie jest to nasze zmartwienie) nikt nie sprawdził, czy sprzęt działa tak, jak powinien. W rzeczywistości społeczność matematyczna zaakceptowała twierdzenie. Być może ma to związek z faktem, że od 1976 r. pojawiło się wiele dodatkowych dowodów, z których wszystkie wykorzystują komputer, ale niektóre z nich muszą sprawdzać tylko mniejszy i łatwiejszy w zarządzaniu zestaw przypadków oraz używać krótszych i jaśniejszych programów. Filozoficzne pozostaje jednak pytanie: czy absolutna prawda matematyczna, w przeciwieństwie do praktycznych aplikacji komputerowych, będzie mogła zależeć od nieco wątpliwej wydajności niezweryfikowanego oprogramowania?

Wydajność algorytmów

Poproszony o zbudowanie mostu nad rzeką, łatwo jest zbudować „niepoprawny" most. Most może nie być wystarczająco szeroki dla wymaganych pasów, może nie być wystarczająco mocny, aby przenosić ruch w godzinach szczytu lub w ogóle nie docierać na drugą stronę! Jednak nawet jeśli jest „poprawny", w tym sensie, że w pełni spełnia wymagania operacyjne, nie każdy kandydat na projekt mostu będzie akceptowalny. Możliwe, że projekt wymaga zbyt dużej siły roboczej lub zbyt wielu materiałów lub komponentów. Doprowadzenie do końca może również zająć zbyt dużo czasu. Innymi słowy, chociaż da to dobry most, projekt może być zbyt drogi. Dziedzina algorytmiki jest podatna na podobne problemy. Algorytm może być zbyt kosztowny, a zatem niedopuszczalny. I tak, chociaż Część 5 była poświęcona udowodnieniu, że niepoprawne algorytmy są złe i że metody są potrzebne do ustalenia poprawności algorytmicznej, ta Część argumentuje, że nawet poprawny algorytm może pozostawiać wiele do życzenia. Ponieważ siła robocza i inne powiązane koszty nie są tutaj istotne, pozostają nam dwa kryteria pomiaru oszczędności - materiały i czas. W informatyce nazywa się je miarami złożoności przestrzeni pamięci (lub po prostu przestrzeni) i czasu. Pierwsza z nich jest mierzona kilkoma czynnikami, w tym liczbą zmiennych oraz liczbą i rozmiarami struktur danych wykorzystywanych podczas wykonywania algorytmu. Druga jest mierzona liczbą elementarnych czynności wykonywanych przez procesor w takim wykonaniu. Zarówno przestrzeń, jak i czas wymagany przez algorytm zazwyczaj różnią się w zależności od danych wejściowych, a zatem wydajność algorytmu odzwierciedla sposób, w jaki te zasoby są zużywane w miarę zmian danych wejściowych. Oczywiste jest, że algorytm sumowania wynagrodzeń trwa dłużej na większych listach. Nie oznacza to, że nie możemy precyzyjnie sformułować jego wykonania w czasie; oznacza to jedynie, że sformułowanie będzie musiało uwzględniać fakt, że czas działania algorytmu zależy lub jest funkcją długości listy wejściowej. To samo dotyczy miejsca w pamięci. W rozwiązaniu dynamicznego planowania problemu zmęczonego podróżnika z Części 4 obliczyliśmy wiele optymalnych podróży częściowych i musieliśmy je przechowywać w wektorze, ponieważ były one wykorzystywane w obliczeniach innych. Ich liczba, a co za tym idzie ilość pamięci wykorzystywanej przez algorytm, zależy bezpośrednio od liczby węzłów w wejściowym grafie miasta. Chociaż w tym rozdziale koncentrujemy się na mierze czasu, należy zauważyć, że kwestie całkiem analogiczne do poruszonych tutaj dotyczą również miary przestrzeni.

Potrzebne są ulepszenia

Naszą dyskusję na temat efektywności czasowej w algorytmice należy rozpocząć od całkowitego odrzucenia mitu dotyczącego szybkości komputera. Niektórzy uważają, że komputery są tak niewiarygodnie szybkie, że nie ma problemu z czasem. Cóż, ta opinia jest bezpodstawna. Oto dwa krótkie przykłady, o których powiemy więcej w dalszej części. Załóżmy, że interesuje nas znalezienie najkrótszej trasy dla podróżnika, który chce odwiedzić każde z, powiedzmy, 200 miast. W tej chwili nie ma komputera, który mógłby znaleźć trasę w czasie krótszym niż miliony lat obliczeniowych! Podobnie żaden komputer nie jest w stanie rozłożyć na czynniki (czyli znaleźć liczb pierwszych, które dzielą) dużych liczb całkowitych, powiedzmy o długości 300 cyfr, w czasie krótszym niż miliony lat. Status obu tych problemów może się zmienić, ale na razie nikt nie zna ich sensownych rozwiązań. Bardziej szczegółowe omówienie rzeczywistego czasu, jaki zająłby współczesnemu komputerowi rozwiązanie tych problemów, znajduje się w Części 7. Czas jest kluczowym czynnikiem w prawie każdym użyciu komputerów. Nawet w codziennych zastosowaniach, takich jak programy do prognozowania pogody, systemy zarządzania zapasami i programy do wyszukiwania bibliotek, istnieje szerokie pole do ulepszeń. Czas to pieniądz, a czas komputera nie jest wyjątkiem. Co więcej, jeśli chodzi o komputery, czas może być czynnikiem krytycznym. Niektóre rodzaje systemów skomputeryzowanych, takie jak sterowanie lotem, naprowadzanie pocisków i programy nawigacyjne, określane są jako systemy czasu rzeczywistego. Muszą reagować na bodźce zewnętrzne, takie jak naciskanie przycisków, w czasie

„rzeczywistym"; to znaczy prawie natychmiast. Niezastosowanie się do tego może okazać się śmiertelne.

Ulepszenia po fakcie

Istnieje wiele standardowych sposobów na poprawienie czasu działania danego algorytmu. Niektóre z nich są włączane do kompilatorów, zamieniając je w kompilatory optymalizujące, które w rzeczywistości rekompensują pewne złe decyzje ze strony programisty. Wiele zadań kompilatora optymalizującego można traktować jako przeprowadzanie pewnych rodzajów przekształceń programu. Jednym z najpowszechniej stosowanych jest modyfikowanie programu lub algorytmu poprzez przenoszenie instrukcji z wnętrza pętli na zewnątrz. Czasami jest to proste, jak w poniższym przykładzie. Załóżmy, że nauczyciel, w interesie, aby klasa zarejestrowała w miarę dobre wyniki na egzaminie, chce znormalizować listę ocen, dając uczniowi, który uzyskał najlepszy wynik na egzaminie, 100 punktów i odpowiednio podnosząc resztę. Wysokopoziomowy opis algorytmu może wyglądać tak:

(1) obliczyć maksymalny wynik w MAX;

(2) pomnóż każdy wynik przez 100 i podziel go przez MAX.

(Musimy założyć, że istnieje co najmniej jedna niezerowa klasa, aby podział był dobrze zdefiniowany.) Jeśli lista jest podana w wektorze L(1), ... , L(N), obie części można wykonać za pomocą prostych pętli biegnących przez wektor. Pierwsza szuka maksimum w jakiś standardowy sposób, a druga może być napisana:

(2) dla I od 1 do N wykonuję:

(2.1) L(I) L(I) × 100/MAX

Zauważ, że dla każdego stopnia L(I) algorytm wykonuje jedno mnożenie i jedno dzielenie w pętli. Jednak ani 100, ani wartość MAX nie zmienia się w pętli. Stąd czas wykonania tej drugiej pętli można skrócić prawie o dwie (!) obliczając stosunek 100/MAX przed rozpoczęciem pętli. Odbywa się to po prostu wstawiając oświadczenie:

WSPÓŁCZYNNIK ← 100/MAKS.

między krokami (1) i (2) oraz pomnożenie L(I) przez FACTOR w pętli. Wynikowy algorytm to:

(1) obliczyć maksymalny wynik w MAX;

(2) WSPÓŁCZYNNIK←100/MAX;

(3) dla I od 1 do N wykonuję:

(3.1) L(I) L(I) × WSPÓŁCZYNNIK.

Powodem prawie 50% poprawy jest to, że treść drugiej pętli, która pierwotnie składała się z dwóch operacji arytmetycznych, teraz składa się tylko z jednej. Oczywiście nie we wszystkich implementacjach takiego algorytmu czas będzie zdominowany przez instrukcje arytmetyczne; możliwe, że aktualizacja wartości L(I) jest bardziej czasochłonne niż dzielenie liczbowe. Mimo to następuje znaczna poprawa czasu działania i wyraźnie im dłuższa lista, tym więcej czasu zyskuje ta zmiana. Jak wspomniano, modyfikacje takie jak ta są dość proste i wiele kompilatorów jest w stanie przeprowadzić je automatycznie. Jednak nawet proste usunięcie instrukcji czasami wymaga sprytnych sztuczek. Spójrzmy na inny przykład.

Wyszukiwanie liniowe: przykład

Załóżmy, że szukamy elementu X na nieuporządkowanej liście (np. numer telefonu w pomieszanej książce telefonicznej). Standardowy algorytm wywołuje prostą pętlę, w ramach której przeprowadzane są dwa testy: (1) „czy znaleźliśmy X?" oraz (2) „czy doszliśmy do końca listy?" Pozytywna odpowiedź na którekolwiek z tych pytań powoduje zakończenie działania algorytmu - pomyślnie w pierwszym przypadku i bezskutecznie w drugim. Ponownie możemy założyć, że testy te dominują w wydajności czasowej algorytmu wyszukiwania. Drugi test można wykonać poza pętlą, korzystając z następującej sztuczki. Przed rozpoczęciem wyszukiwania wymagany element X jest fikcyjnie dodawany na końcu listy. Następnie wykonywana jest pętla wyszukiwania, ale bez testowania końca listy; w pętli sprawdzamy tylko, czy X został znaleziony. Skraca to również czas całego algorytmu o około 50%. Teraz, odkąd dodaliśmy X do listy, X zawsze zostanie znaleziony, nawet jeśli nie pojawił się na oryginalnej liście. Jednak w tym przypadku znajdziemy się na końcu nowej listy, gdy konfrontujemy się z X po raz pierwszy, podczas gdy znajdziemy się gdzieś w jej obrębie, jeśli X pojawił się na oryginalnej liście. W konsekwencji, po osiągnięciu X, test na znalezienie się na końcu listy jest przeprowadzany raz, a algorytm zgłasza sukces lub porażkę w zależności od wyniku. (Nawiasem mówiąc, częsty błąd może wystąpić tutaj, jeśli zapomnimy usunąć fikcyjnego X z końca listy przed zakończeniem.) Tym razem musieliśmy być nieco bardziej kreatywni, aby zaoszczędzić 50%.

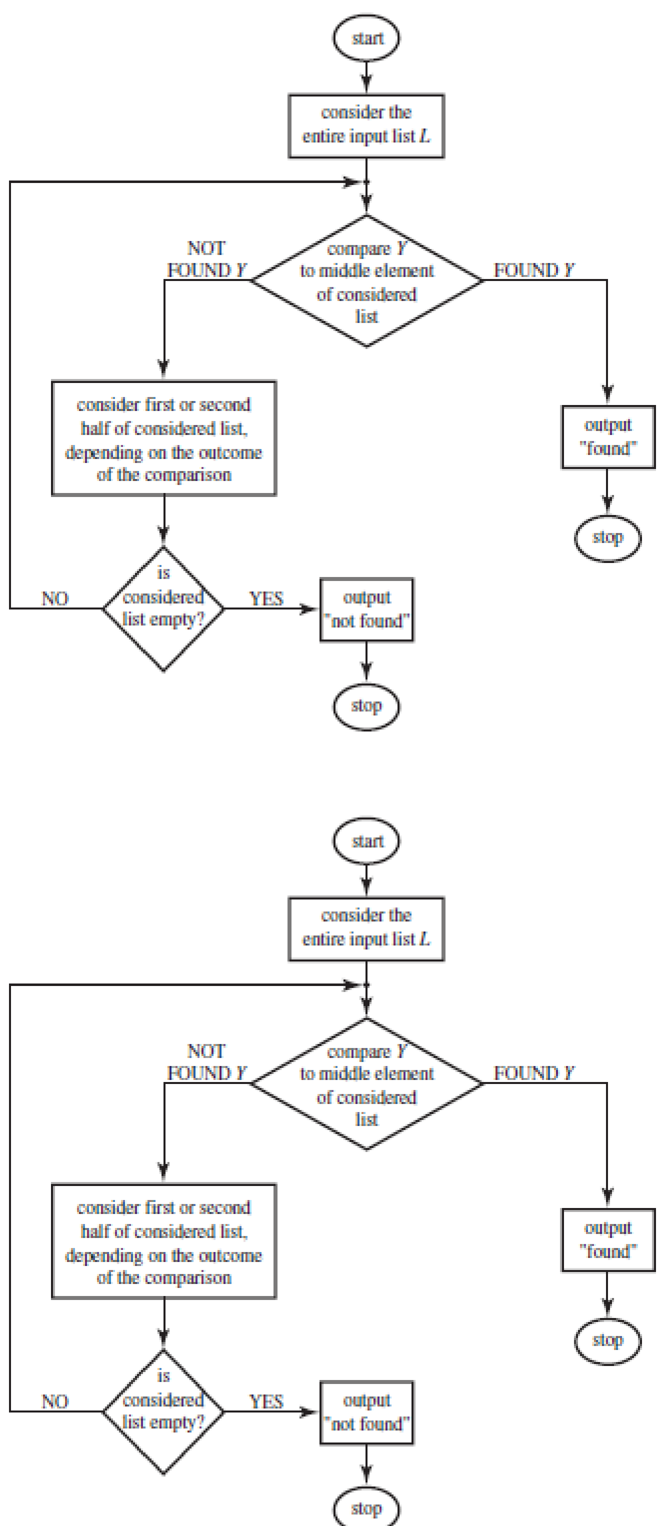
Ulepszenia rzędu wielkości

Skrócenie czasu działania algorytmu o 50% jest imponujące. Jak to jednak bywa, w wielu przypadkach potrafimy zrobić dużo lepiej. Kiedy mówimy „lepiej", nie mamy na myśli tylko stałych spadków o 50%, 60%, a nawet 90%, ale spadki, których stopa staje się coraz lepsza wraz ze wzrostem wielkości danych wejściowych. Jednym z najbardziej znanych przykładów jest wyszukiwanie elementu na uporządkowanej lub posortowanej liście (na przykład wyszukiwanie numeru telefonu w normalnej książce telefonicznej). Mówiąc dokładniej, załóżmy, że dane wejściowe składają się z nazwy Y oraz listy L nazwisk i ich numerów telefonów. Zakłada się, że lista, która ma długość N, jest posortowana alfabetycznie według nazw. Naiwny algorytm, który wyszukuje numer telefonu Y, to ten opisany wcześniej dla nieposortowanej listy: przeglądaj listę L po jednym nazwisku, porównując Y z bieżącą nazwą na każdym kroku i sprawdzając koniec listy listy w tym samym czasie lub stosując sztuczkę opisaną w poprzedniej sekcji i sprawdzając ją tylko raz, gdy zostanie znalezione Y. Nawet jeśli sztuczka zostanie zastosowana, aby skrócić czas działania o 50%, nadal może istnieć przypadek, w którym konieczne jest pełne N porównań; mianowicie, gdy Y w ogóle nie pojawia się w L lub gdy pojawia się na ostatniej pozycji. Mówimy, że algorytm, nazwijmy go A, ma najgorszy możliwy czas działania, który jest liniowy w N lub, używając równoważnego terminu, jest rzędu N. Jest to opisane bardziej zwięźle, mówiąc, że A działa w czas O(N) w najgorszym przypadku, gdzie duże-O oznacza „kolejność". Notacja O(N) jest dość subtelna. Zauważ, że powiedzieliśmy „A działa w czasie O(N)". Nie doprecyzowaliśmy stwierdzenia, aby odnosiło się tylko do liczby porównań, chociaż porównania między Y a nazwiskami w L były jedynym rodzajem instrukcji, który liczyliśmy. Powód tego zostanie wyjaśniony później. Na razie wystarczy powiedzieć, że używając notacji duże-O, jak to się czasem nazywa, nie obchodzi nas, czy algorytm przyjmuje czas N (czyli wykonuje N instrukcji elementarnych), czas 3N (czyli wymaga trzykrotnie większej liczby instrukcji elementarnych), czyli 10N, a nawet 100N. Co więcej, nawet jeśli zajmuje tylko ułamek N, powiedzmy N/6, nadal mówimy, że działa w czasie O(N). Jedyną rzeczą, która ma znaczenie, jest to, że czas działania rośnie liniowo wraz z N. Oznacza to, że istnieje pewna stała liczba K, taka, że algorytm działa w czasie nie większym niż K × N w najgorszym przypadku. Rzeczywiście, wersja, która sprawdza koniec listy za każdym razem działa mniej więcej w czasie 2N, a podstępna wersja zmniejsza to do mniej więcej N. Jednak obie mają czas działania wprost proporcjonalny do N. Są to zatem algorytmy czasu liniowego , mający najgorszy czas działania O(N). Termin „najgorszy przypadek" oznacza, że algorytm mógłby prawdopodobnie działać znacznie mniej na niektórych wejściach, być może na większości wejść. Mówimy tylko, że algorytm nigdy nie działa dłużej niż K × N

czasu i że jest to prawdą dla dowolnego N i dla dowolnego wejścia o długości N, nawet najgorszych. Oczywiście, jeśli spróbujemy ulepszyć algorytm wyszukiwania w czasie liniowym, rozpoczynając porównania od końca listy, nadal będą występować równie złe przypadki - Y nie pojawia się w ogóle lub Y pojawia się jako pierwsze imię na liście. W rzeczywistości, jeśli algorytm wymaga dokładnego przeszukania listy, kolejność, w jakiej nazwy są porównywane z Y, nie ma znaczenia. Taki algorytm nadal będzie miał czas działania O(N) w najgorszym przypadku. Mimo to, możemy lepiej przeszukiwać uporządkowaną listę, nie tylko pod względem stałego czynnika „ukrytego wewnątrz" dużego O, ale pod względem samego oszacowania O(N). Jest to poprawa rzędu wielkości i teraz zobaczymy, jak można ją osiągnąć.

Wyszukiwanie binarne: Przykład

Dla konkretności załóżmy, że książka telefoniczna zawiera milion nazwisk, czyli N to 1 000 000 i nazwijmy je X1, X2, ... , X1 000 000. Pierwsze porównanie przeprowadzone przez nowy algorytm nie następuje między Y a imieniem lub nazwiskiem w L, ale między Y i drugim imieniem (lub, jeśli lista jest parzysta, to nazwisko z pierwszej połowy listy), czyli $X_{500\,000}$. Zakładając, że porównywane nazwy okażą się nierówne, co oznacza, że jeszcze nie skończyliśmy, istnieją dwie możliwości: (1) Y poprzedza $X_{500\,000}$ w kolejności alfabetycznej, oraz (2) $X_{500\,000}$ poprzedza Y . Ponieważ lista jest posortowana alfabetycznie, jeśli tak jest w przypadku (1), wiemy, że jeśli w ogóle pojawia się na liście, to musi być w pierwszej połowie, a jeśli tak jest w przypadku (2), to musi pojawić się w drugiej połowie. Możemy więc ograniczyć nasze kolejne poszukiwania do odpowiedniej połowy listy. W związku z tym następne porównanie będzie pomiędzy Y a środkowym elementem tej połowy: X250.000 w przypadku (1) i $X_{750\,000}$ w przypadku (2). I znowu, wynikiem tego porównania będzie zawężenie możliwości do połowy tej nowej, krótszej listy; to znaczy do listy, której długość jest równa jednej czwartej oryginału. Ten proces jest kontynuowany, zmniejszając długość listy lub ogólniej, rozmiar problemu o połowę na każdym kroku, aż do znalezienia Y, w którym to przypadku procedura kończy się, zgłaszając sukces lub trywialną pustą listę zostanie osiągnięty, w takim przypadku kończy się, zgłaszając niepowodzenie . Ta procedura nazywa się przeszukiwaniem binarnym i jest tak naprawdę zastosowaniem paradygmatu dziel i zwyciężaj omówionego w Części 4. Różnica między tym przykładem a wprowadzonymi w nim (przeszukiwanie min&max i sortowanie przez scalanie) polega na tym, że po dzieleniu potrzebujemy tylko podbij jedną z części, a nie obie. Rysunek zawiera schematyczną iteracyjną wersję wyszukiwania binarnego, ale w rzeczywistości możliwe jest również zapisanie prostej wersji rekurencyjnej.



Zachęcamy do tego. Jaka jest złożoność czasowa wyszukiwania binarnego? Aby na to odpowiedzieć, najpierw policzmy porównania. Próba odgadnięcia, ile porównań będzie wymagał algorytm wyszukiwania binarnego w najgorszym przypadku w naszej książce telefonicznej zawierającej milion nazwisk, jest dość pouczające. Przypomnijmy, że naiwne poszukiwania wymagałyby miliona porównań.

Cóż, w najgorszym przypadku (jaki jest przykład jednego? ile jest najgorszych przypadków?) algorytm będzie wymagał tylko 20 porównań! Co więcej, im większe staje się N, tym bardziej imponująca jest poprawa. Międzynarodowa książka telefoniczna zawierająca, powiedzmy, miliard nazwisk, wymagałaby co najwyżej 30 porównań zamiast miliarda! Przypomnij sobie, że każde porównanie zmniejsza długość listy wejściowej L o połowę, a proces kończy się, gdy lista stanie się pusta lub wcześniej. Stąd najgorszą liczbę porównań uzyskuje się, obliczając, ile razy liczba N może być wielokrotnie podzielona przez 2, zanim zostanie zredukowana do 0. (Zasady gry polegają na ignorowaniu ułamków). logarytm o podstawie 2 z N i jest oznaczony przez log2N. W rzeczywistości log2 N zlicza liczbę potrzebną do zredukowania N do 1, a nie do 0, więc tak naprawdę szukamy $1 + \log_2 N$. można śmiało powiedzieć, że algorytm wykonuje porównania O(log N) w najgorszym przypadku. Możemy wyczuć rodzaj ulepszeń, jakie oferuje wyszukiwanie binarne, analizując poniższą tabelę, która przedstawia kilka wartości N w stosunku do liczby porównań wymaganych przez wyszukiwanie binarne w najgorszym przypadku:

N : $1 + \log_2 N$

10 : 4

100 : 7

1000 : 10

milion : 20

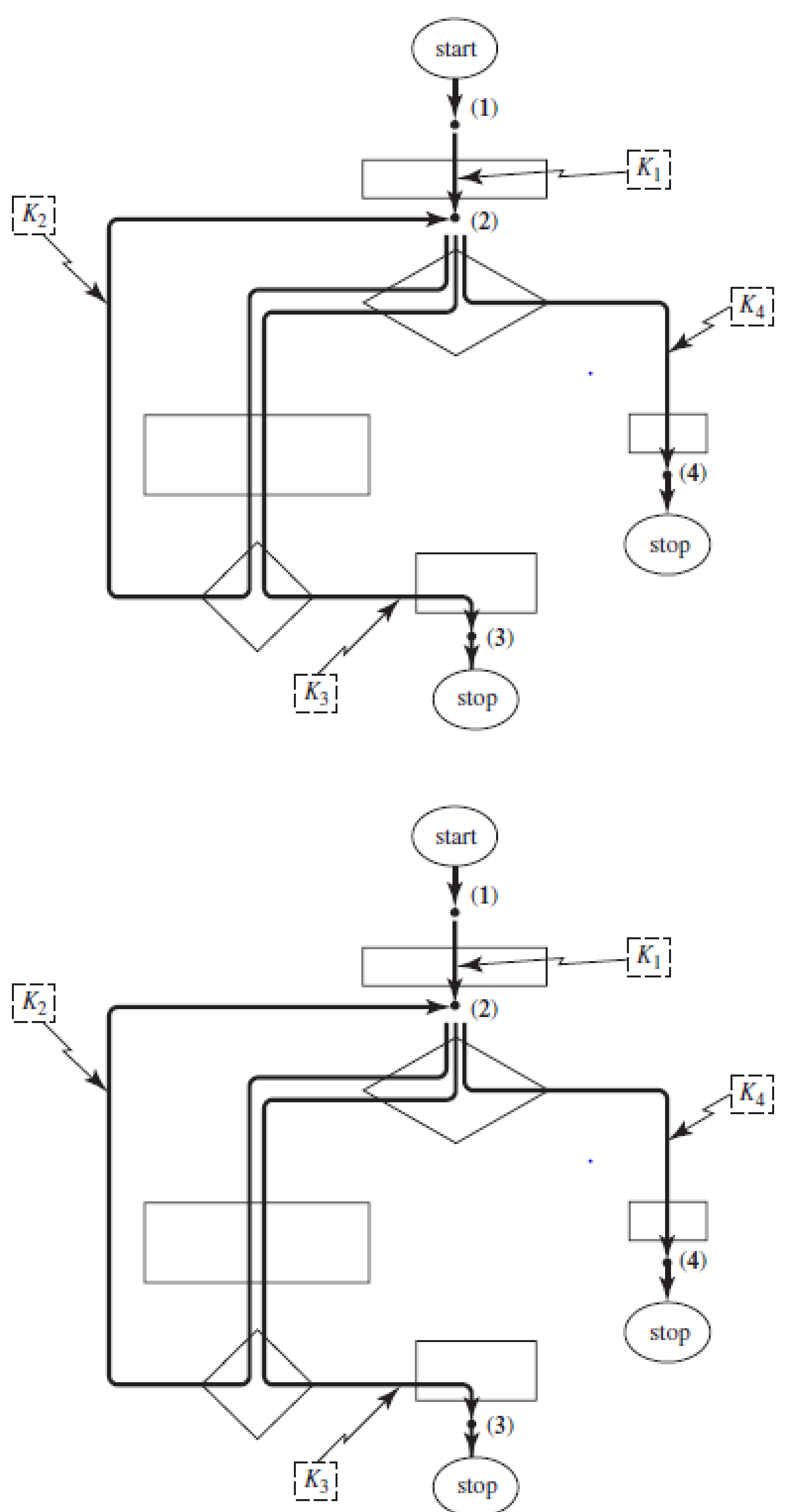
miliard : 30

miliard miliardów : 60

Warto zauważyć, że sami posługujemy się wariantem wyszukiwania binarnego, gdy szukamy numeru w książce telefonicznej. Różnica polega na tym, że niekoniecznie porównujemy dane nazwisko z tym, które pojawia się dokładnie w środku książki, a następnie z tym w 1/4 lub 3/4 pozycji i tak dalej. Raczej korzystamy z dodatkowej wiedzy, którą posiadamy, dotyczącej oczekiwanego rozmieszczenia nazwisk w księdze. Ta technika jest często nazywana wyszukiwaniem interpolacyjnym. Jeśli szukamy na przykład „Mary D. Ramsey", otworzymy książkę mniej więcej w punkcie dwóch trzecich. Oczywiście jest to dalekie od precyzji i pracujemy według ogólnych, intuicyjnych reguł, które zostaną omówione pod pojęciem heurystyki w Części 15. Niemniej jednak istnieją precyzyjne sformułowania algorytmów wyszukiwania, które działają w sposób krzywy, podyktowany naturą. i rozmieszczenie elementów. Ogólnie rzecz biorąc, chociaż są one średnio znacznie bardziej ekonomiczne, zmiany te wykazują również podobne zachowanie w najgorszym przypadku jak porównania $O(\log_2 N)$.

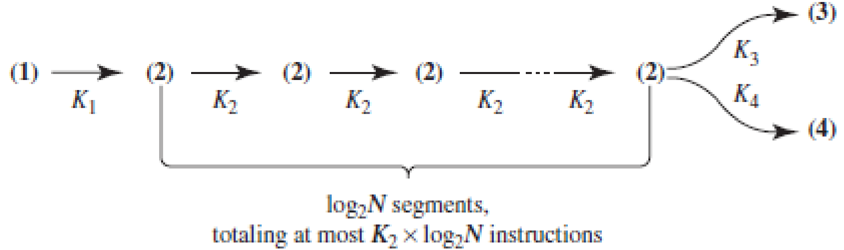
Dlaczego wystarczy liczyć porównania?

Aby zakończyć naszą analizę złożoności wyszukiwania binarnego, pokazujemy teraz, że jeśli jesteśmy zadowoleni z oszacowania dużego O, wystarczy zliczyć tylko porównania. Czemu? Oczywiście w wyszukiwaniu binarnym wykonywanych jest znacznie więcej instrukcji niż tylko porównania. Ponowne skupienie się na właściwej połowie listy i przygotowanie jej na następny raz wokół pętli prawdopodobnie wiązałoby się z pewnym testowaniem i zmianą indeksów. A potem jest test na pustą listę i tak dalej. Jak się okazuje, pomocne okazują się również punkty kontrolne użyte w poprzednim rozdziale do sprawdzania poprawności. Rysunek przedstawia schemat blokowy z uwzględnieniem punktów kontrolnych i ilustruje możliwe ścieżki lokalne lub „przeskoki" między nimi.



Zauważ, że również tutaj punkty kontrolne przecinają wszystkie pętle - w tym przypadku jest tylko jedna - tak, że cztery ścieżki lokalne są wolne od pętli, jak w przykładzie podanym w Części 5. Fakt ten był ważny dla dowodów poprawności, ponieważ dał początek łatwym do opanowania warunkom weryfikacji, tutaj jest to ważne, ponieważ segmenty bez pętli i podprogramów zajmują maksymalnie stałą ilość czasu. Prawdą jest, że obecność instrukcji warunkowych w takim segmencie może dać kilka różnych sposobów jego przechodzenia. Jednak brak pętli (i podprogramów rekurencyjnych)

gwarantuje, że istnieje ograniczenie tej liczby, a zatem całkowita liczba instrukcji wykonywanych w pojedynczym wykonaniu takiego segmentu nie jest większa niż pewna stała. Jak zobaczymy, często ułatwia to łatwe obliczenie czasu, jaki zajmuje algorytm. Na rysunku skojarzyliśmy stałe od K_1 do K_4 z czterema możliwymi ścieżkami lokalnymi. Na przykład oznacza to, że zakłada się, że każde pojedyncze wykonanie ścieżki lokalnej z punktu kontrolnego (2) z powrotem do samego siebie obejmuje nie więcej niż instrukcje K_2. (Umownie przyjmuje się, że obejmuje to porównanie wykonane na początku segmentu, ale nie na jego końcu). Rozważmy teraz typowe wykonanie wyszukiwania binarnego



Ponieważ punkt kontrolny (2) jest jedynym miejscem w tekście algorytmu dotyczącego porównania, jest on osiągany (w najgorszym przypadku) dokładnie $1 + \log_2 N$ razy, ponieważ, jak omówiono powyżej, jest to całkowita liczba dokonanych porównań. Teraz wszystkie z wyjątkiem ostatniego z tych porównań powodują, że procesor ponownie omija pętlę, a tym samym wykonuje co najwyżej kolejne K instrukcji. Oznacza to, że całkowity koszt czasu wszystkich tych przejść log2 N segmentu (2)→(2) wynosi $K_2 \times \log_2 N$. Aby zakończyć analizę, zauważ, że łączna liczba wszystkich wykonanych instrukcji, które nie są częścią (2) →(2) segmenty, to albo $K_1 + K_3$ albo $K_1 + K_4$, w zależności od tego, czy algorytm zatrzymuje się w punkcie kontrolnym (3) czy (4) ; w obu przypadkach mamy stałą niezależną od N. Jeśli przez K oznaczymy maksimum z tych dwóch sum, możemy wywnioskować, że całkowita liczba instrukcji wykonywanych przez algorytm przeszukiwania binarnego na liście o długości N jest ograniczona za pomocą:

$K + (K_2 \times \log_2 N)$

W związku z tym, ponieważ używamy notacji rzędu wielkości, stałe K i K_2 można „zakopać" pod big-O i możemy po prostu wywnioskować, że wyszukiwanie binarne przebiega w czasie $O(\log_2 N)$ w najgorszym przypadku . Nazywa się to algorytmem logarytmicznym.

Solidność notacji Big-O

Jak wyjaśniono, big-Os ukrywają stałe czynniki. W związku z tym w najgorszej analizie wyszukiwania binarnego nie było potrzeby, abyśmy byli bardziej precyzyjni w szczegółach poszczególnych instrukcji składających się na algorytm. Wystarczyło przeanalizować strukturę pętli algorytmu i przekonać się, że pomiędzy dowolnymi dwoma porównaniami jest tylko stała liczba instrukcji w najgorszym przypadku. Jakoś to nie brzmi całkiem dobrze. Z pewnością złożoność czasowa wyszukiwania binarnego zależy od dozwolonych podstawowych instrukcji. Gdybyśmy zezwolili na instrukcję:

search list L for item Y

algorytm składałby się po prostu z pojedynczej instrukcji, a zatem jego złożoność czasowa byłaby O(1), czyli stała liczba w ogóle nie zależna od N. Jak więc możemy powiedzieć, że szczegóły instrukcji są nieistotne? Cóż, złożoność czasowa algorytmu jest rzeczywiście pojęciem względnym i ma sens tylko w połączeniu z uzgodnionym zbiorem instrukcji elementarnych. Niemniej jednak, standardowe rodzaje problemów są zwykle rozważane w kontekście standardowych rodzajów instrukcji elementarnych. Na przykład, wyszukiwanie i sortowanie problemów zazwyczaj obejmuje porównania, aktualizacje

indeksów i testy końca listy, tak że analiza złożoności jest przeprowadzana w odniesieniu do nich. Co więcej, jeśli określony język programowania jest z góry ustalony, to określony zestaw instrukcji elementarnych również został poprawiony, na dobre lub na złe. To, co sprawia, że obserwacje te są istotne, to fakt, że w przypadku większości standardowych rodzajów instrukcji elementarnych możemy sobie pozwolić na dość niejasne, jeśli chodzi o złożoność rzędu wielkości. Napisanie algorytmu w określonym języku programowania lub użycie określonego kompilatora może oczywiście wpłynąć na ostateczny czas działania. Jeśli jednak algorytm używa konwencjonalnych instrukcji podstawowych, różnice te będą składać się tylko ze stałego współczynnika na instrukcję podstawową, a to oznacza, że złożoność big-O jest niezmienna przy takich fluktuacjach implementacyjnych. Innymi słowy, tak długo, jak uzgodniony jest podstawowy zestaw dozwolonych instrukcji elementarnych i tak długo, jak wszelkie skróty stosowane w opisach wysokiego poziomu nie ukrywają nieograniczonych iteracji takich instrukcji, a jedynie reprezentują ich skończone skupiska, duże-O szacunki czasu są solidne. Właściwie dla dobra czytelników, którzy z logarytmami czują się jak w domu, należy dodać, że podstawa logarytmiczna też nie ma znaczenia. Dla dowolnego ustalonego K liczby $\log_2 N$ i logKN różnią się tylko stałym współczynnikiem, a zatem tę różnicę można również ukryć pod notacją big-O. W konsekwencji będziemy odnosić się do wydajności w czasie logarytmicznym po prostu jako O(logN), a nie $O(\log_2 N)$. Solidność zapisu big-O stanowi zarówno jego siłę, jak i słabość. Kiedy ktoś wykazuje algorytm logarytmiczny czasu A, podczas gdy ktoś inny algorytm B działa w czasie liniowym, można bardzo dobrze stwierdzić, że B działa szybciej na przykładowych danych wejściowych niż A! Powód tkwi w ukrytych stałych. Powiedzmy, że skrupulatnie wzięliśmy pod uwagę stałą liczbę elementarnych instrukcji między punktami kontrolnymi, język programowania, w którym zaszyfrowany jest algorytm, kompilator, który tłumaczy go w dół, podstawowe instrukcje maszynowe używane przez komputer z kodem maszynowym oraz samą prędkość tego komputera. Po wykonaniu tej czynności możemy stwierdzić, że algorytm A działa w czasie ograniczonym przez $K \times \log_2 N$ nanosekund, a B działa w ciągu $J \times N$ nanosekund, ale przy K równym 1000, a J równym 10. Oznacza to, że jak możesz zweryfikować, że dla każdego wejścia o długości mniejszej niż tysiąc (w rzeczywistości dokładna liczba to 996), algorytm B jest lepszy od A. Tylko wtedy, gdy osiągnięto dane wejściowe o wielkości 1000 lub większej, różnica między N a $\log_2 N$ stają się widoczne i, jak już wspomniano, kiedy różnica zaczyna się opłacać, robi to bardzo ładnie: do czasu osiągnięcia wielkości wejściowych miliona, algorytm A staje się do 500 razy bardziej wydajny niż algorytm B, a dla nakłady o wielkości miliarda, poprawa jest ponad 330 000 razy! Tak więc użytkownik, który jest zainteresowany tylko danymi wejściowymi o długości poniżej tysiąca, powinien zdecydowanie zastosować algorytm B, pomimo wyższości rzędu wielkości A. Jednak w większości przypadków współczynniki stałe nie są tak daleko od siebie jak 10 i 1000, stąd szacunki dużego O są zwykle znacznie bardziej realistyczne niż w tym wymyślonym przykładzie. Morał tej historii polega na tym, aby najpierw poszukać dobrego i wydajnego algorytmu, podkreślającego wydajność dużego O, a następnie spróbować go ulepszyć za pomocą sztuczek stosowanych wcześniej w celu zmniejszenia zaangażowanych czynników stałych. W każdym razie, ponieważ wydajność big-O może być myląca, kandydujące algorytmy powinny być uruchamiane eksperymentalnie, a ich wyniki czasowe dla różnych typowo występujących rodzajów danych wejściowych powinny być zestawione. Odporność oszacowań dużego O, w połączeniu z faktem, że w większości przypadków algorytmy, które są lepsze w sensie dużego O, są również lepsze w praktyce, sprawia, że badanie złożoności czasowej rzędu wielkości jest najbardziej interesujące dla komputera. naukowcy. W związku z tym, w imię nauki i solidności, w dalszej części skoncentrujemy się głównie na szacunkach big-O i podobnych pojęciach, chociaż mogą one ukrywać kwestie o możliwym znaczeniu praktycznym, takie jak czynniki stałe.

Analiza czasu zagnieżdżonych pętli

Oczywiście skomplikowane algorytmy, które obejmują wiele powiązanych ze sobą struktur kontrolnych i zawierają potencjalnie rekurencyjne podprogramy, mogą być dość trudne do przeanalizowania.

Omówmy pokrótce złożoność czasową niektórych algorytmów pojawiających się w poprzednich częściach. Algorytm sortowania bąbelków z części 2 składał się z dwóch pętli zagnieżdżonych w następujący sposób:

(1) wykonaj następujące N-1 razy:

....

(1.2) wykonaj następujące N - 1 razy:

....

Wewnętrzna pętla jest wykonywana N-1 razy na każdy N-1 razy wykonywana jest pętla zewnętrzna. Jak poprzednio, wszystko inne jest stałe; stąd całkowita wydajność czasowa sortowania pęcherzyków jest rzędu (N - 1) × (N - 1), czyli N2 - 2N + 1. W tym przypadku N2 nazywamy wyrazem dominującym wyrażenia, co oznacza, że inne części, a mianowicie -2N i +1, zostają „połknięte" przez N2, gdy używa się notacji Big-O. W konsekwencji sortowanie bąbelkowe jest algorytmem O(N2), czyli czasu kwadratowego. Przypomnij sobie naszą dyskusję na temat ulepszonej wersji sortowania bąbelkowego, która umożliwiała przechodzenie przez coraz mniejsze fragmenty listy wejściowej. Pierwsze przejście składa się z N − 1 elementów, następne z N − 2 elementów i tak dalej. Dlatego analiza czasu wyniesie w sumie:

(N − 1) + (N − 2) + (N − 3) + ...·+3 + 2 + 1

które można wykazać, aby ocenić do (N2 − N)/2. To mniej niż N2 − 2N + 1 naiwnej wersji, ale nadal jest kwadratowa, ponieważ dominującym czynnikiem jest N2/2, a stały czynnik 12 również zostaje utracony. W ten sposób mamy 50% skrócenie czasu, ale nie dużą poprawę. Łatwo zauważyć, że prosty algorytm sumowania wynagrodzeń (rysunek 2.3) jest liniowy, to znaczy O(N), ale algorytm dla bardziej zaawansowanej wersji, która pociągała za sobą zagnieżdżoną pętlę do wyszukiwania bezpośrednich menedżerów , jest kwadratowy . (Dlaczego?) Algorytm wyszukiwania „pieniędzy", z zastosowaniem procedury lub bez niej może być postrzegany jako liniowy, co należy dokładnie sprawdzić; chociaż mogą istnieć dwa wskaźniki, które rozwijają się oddzielnie, tekst jest przeszukiwany tylko raz, w sposób liniowy. Mając wiedzę zdobytą w niniejszym rozdziale, powinno być całkiem proste ustalenie powodu, dla którego próbowaliśmy ulepszyć algorytm maksymalnej odległości wielokątnej w Części 4. Algorytm naiwny, który przebiega przez wszystkie pary wierzchołków, jest kwadratowy (gdzie N przyjmuje się, że jest liczbą wierzchołków), podczas gdy ulepszony algorytm wykonuje cykle wokół wielokąta tylko raz i można wykazać, że jest liniowy.

Analiza czasu rekurencji

Rozważmy teraz problem min&max z Części 4, który wymagał znalezienia elementów ekstremalnych na liście L. Algorytm naiwny przechodzi przez listę iteracyjnie, aktualizując dwie zmienne, które przechowują bieżące elementy ekstremalne. Jest wyraźnie liniowy. Oto procedura rekurencyjna, która w Rozdziale 4 została uznana za lepszą:

podprogram znajdź-min&maks-z L:

(1) jeśli L składa się z jednego elementu, to ustaw dla niego MIN i MAX; jeśli składa się z dwóch elementów, to ustaw MIN na mniejszy z nich i MAX na większy;

(2) w przeciwnym razie wykonaj następujące czynności:

(2.1) podzielić L na dwie połowy, Lleft i Lright;

(2.2) wywołaj find-min&max-of Lleft, umieszczając zwrócone wartości w MINleft i MAXleft;

(2.3) wywołaj find-min&max-of Lright, umieszczając zwrócone wartości w MINright i MAXright;

(2.4) ustaw MIN na mniejsze z MINleft i MINright;

(2.5) ustaw MAX na większy z MAXleft i MAXright;

(3) powrót z MIN i MAX.

Okaże się, że ta rekurencyjna procedura również działa w czasie liniowym. (W rzeczywistości, jak przekonujemy później, żaden algorytm dla problemu min&max nie może być podliniowy, powiedzmy logarytmiczny.) Jednak procedura rekurencyjna ma mniejszą stałą pod Big-O. Aby zobaczyć dlaczego, potrzebna jest bardziej wyrafinowana analiza złożoności. Algorytm iteracyjny działa, przeprowadzając dwa porównania dla każdego elementu na liście, jedno z bieżącym maksimum, a drugie z bieżącym minimum. Stąd daje całkowitą liczbę porównawczą 2BA. Interesującą częścią jest sposób, w jaki liczy się czas dla rutyny rekurencyjnej. Ponieważ nie wiemy od ręki, ile porównań rutyna wymaga, używamy specjalnie dostosowanej notacji abstrakcyjnej: niech C(N) oznacza (najgorszy przypadek) liczbę porównań wymaganych przez rekurencyjną procedurę min&max na listach o długości N. Teraz, chociaż nie znamy wyraźnej wartości C(N) jako funkcja N wiemy dwie rzeczy:

1. Jeśli N wynosi 2, to przeprowadzane jest dokładnie jedno porównanie - to, które wynika z wiersza (1) procedury; jeśli N wynosi 3, przeprowadzane są trzy porównania, co można zweryfikować.

2. Jeżeli N jest większe od 3, to przeprowadzone porównania składają się dokładnie z dwóch zestawów porównań dla list o długości N/2, ponieważ są dwa wywołania rekurencyjne i dwa dodatkowe porównania — te pojawiające się w wierszach (2.4) i (2.5). (Jeśli N jest nieparzyste, listy mają długość (N + 1)/2 i (N − 1)/2.) Możemy zatem zapisać następujące równania, czasami nazywane relacjami powtarzalności, które wychwytują odpowiednio dwie obserwacje właśnie zrobiłem:

i. $C(2) = 1$

ii. $C(N) = 2 \times C(N/2) + 2$

(Dotyczy to przypadku, w którym N jest potęgą liczby 2. Ogólny przypadek jest nieco bardziej skomplikowany.) Chcielibyśmy znaleźć funkcję C(N) spełniającą te ograniczenia. I rzeczywiście, istnieją metody rozwiązywania takich równań rekurencyjnych iw tym przypadku, jak łatwo sprawdzić, rozwiązanie okazuje się być:

$C(N) = 3N/2 - 2$

Oznacza to, że C(N) jest mniejsze niż 1,5N. Nawet w ogólnym przypadku, dla liczb, które nie są potęgami 2, procedura rekurencyjna wymaga mniej niż 1,7N porównań dla list o długości N, co jest lepsze niż 2N wymagane przez rozwiązanie iteracyjne. Jak wspomniano, jest to nadal O(N), niemniej jednak jest to poprawa, zwłaszcza jeśli porównania są bardzo czasochłonne w pożądanej aplikacji. Przedstawiono tutaj procedurę rekurencyjną min&max, aby dać przykład analizy czasowej algorytmu rekurencyjnego. Warto jednak zauważyć, że faktycznie możemy osiągnąć lepsze zachowanie, używając tylko 1,5N porównań dla ogólnego przypadku znajdowania minimum i maksimum na liście za pomocą (innego) algorytmu iteracyjnego w następujący sposób. Najpierw ułóż N elementów w pary, a następnie porównaj dwa elementy w każdej parze, zaznaczając większy z nich. Kosztuje to N/2 porównań. Następnie przejdź przez N/2 większe elementy, śledząc bieżące maksimum, i podobnie przez N/2 mniejsze elementy, śledząc minimum. Kosztuje to dwa razy więcej porównań N/2, co daje łącznie 1,5N.

W Częściach 2 i 4 zostały pokrótce opisane dwa dodatkowe algorytmy sortowania, sortowanie po drzewach i sortowanie przez scalanie. Nie będziemy tutaj zajmować się szczegółową analizą czasu, z wyjątkiem następujących uwag: Sortowanie drzew, jeśli jest zaimplementowane naiwnie, jest algorytmem kwadratowym. Powód wynika z możliwości, że pewne sekwencje elementów dają w wyniku bardzo długie i wąskie binarne drzewa poszukiwań. Chociaż przechodzenie przez takie drzewo w pierwszym przejściu w lewo jest procedurą w czasie liniowym, można wykazać, że konstruowanie drzewa z sekwencji wejściowej jest kwadratowe. Możliwe jest jednak zastosowanie schematu samodopasowania, omówionego w Części 2 i trzymaj drzewo szerokie i krótkie. Włączenie takiego schematu do fazy konstrukcyjnej sortowania drzew spowoduje poprawę o rząd wielkości, dając algorytm, który używa czasu w kolejności iloczynu N i logarytmu N (a nie N); w symbolach jest to algorytm O(N × log N). Poniższa tabela powinna dać wyobrażenie o oszczędnościach, jakie zapewnia ta poprawa, chociaż jej wpływ na czynniki stałe nie jest widoczny. Przechodząc do sortowania przez scalanie, pozostaje ci interesujące ćwiczenie pokazujące, że ten algorytm sam w sobie jest O(N × log N). Mergesort jest tak naprawdę jednym z kilku wydajnych czasowo algorytmów sortowania i jest zdecydowanie najłatwiejszy do opisania spośród algorytmów O(N × log N). Należy jednak zauważyć, że wykorzystuje nową listę do przechowywania wyników cząstkowych, a zatem wymaga dodatkowej liniowej ilości miejsca, co jest wadą w porównaniu z niektórymi innymi metodami sortowania. W ten sposób mergesort jest jednym z najbardziej efektywnych czasowo procedur sortowania, a także jednym z najłatwiejszych do opisania. Chcielibyśmy przypisać rekurencji obie zalety: ułatwia ona opisanie algorytmu, a także zapewnia przejrzysty mechanizm dzielenia problemu na dwa mniejsze problemy o połowę mniejsze, co stanowi pierwiastek O(N × log) algorytmu N) wydajność czasowa. Inne podejście wykorzystuje stertę zamiast binarnego drzewa wyszukiwania, jak w treesort. Algorytm heapsort najpierw wstawia wszystkie elementy listy wejściowej do sterty, a następnie wielokrotnie wyodrębnia minimalny element, aby utworzyć posortowaną listę. Reprezentacja sterty jako wektora (patrz rozdział 4) zapewnia, że sterta jest zawsze zrównoważona, a zatem każda operacja wstawiania i wydobywania z minimum zajmuje czas logarytmiczny, dając całkowity czas działania O(N × log N).

Złożoność średnich przypadków

Najgorszy przypadek naszych analiz czasowych można interpretować jako wadę. Prawdą jest, że nie można zagwarantować, powiedzmy, wydajności algorytmu w czasie liniowym, chyba że ograniczenie czasowe ma zastosowanie do wszystkich danych wejściowych zgodnych z prawem. Może się jednak zdarzyć, że algorytm jest bardzo szybki dla większości standardowych rodzajów danych wejściowych, a te, które powodują wzrost wydajności w najgorszym przypadku, są mniejszością, którą jesteśmy skłonni zignorować. W związku z tym istnieją inne przydatne szacunki wydajności czasowej algorytmu, takie jak jego zachowanie w średniej wielkości. Tutaj interesuje nas czas wymagany przez algorytm przeciętnie, biorąc pod uwagę cały zestaw danych wejściowych i ich prawdopodobieństwo wystąpienia. Nie będziemy tu wchodzić w szczegóły techniczne, z wyjątkiem uwagi, że analiza przeciętnego przypadku jest znacznie większa trudne do przeprowadzenia niż analiza najgorszego przypadku. Wymagana matematyka jest zwykle znacznie bardziej wyrafinowana i istnieje wiele algorytmów, dla których badacze w ogóle nie byli w stanie uzyskać średnich szacunków. Pomimo fundamentalnej różnicy, wiele algorytmów ma ten sam limit czasu Big-O zarówno dla zachowania najgorszego, jak i średniego przypadku. Na przykład proste sumowanie wynagrodzeń działa w ustalony sposób, zawsze biegnąc do samego końca listy, a zatem jest liniowe w najgorszych, najlepszych i przeciętnych przypadkach. Wersja, która szuka bezpośrednich menedżerów, która w najgorszym przypadku jest kwadratowa, może przeciętnie wymagać przejrzenia tylko połowy listy dla każdego menedżera pracownika, a nie całej. Jednak to tylko zmniejsza N^2 do około $N^2/2$, zachowując ograniczenie czasowe $O(N^2)$ nawet w przeciętnym przypadku, ponieważ stała połowa jest ukryta pod big-O. W przeciwieństwie do tego, w przypadku niektórych algorytmów analiza średnich przypadków

może ujawnić znacznie lepszą wydajność. Klasycznym tego przykładem jest jeszcze inny algorytm sortowania, zwany quicksort, którego nie będziemy tutaj opisywać. Quicksort, również naturalnie rekurencyjny algorytm, ma najgorszy przypadek kwadratowy czasu działania, a zatem wydaje się być gorszy zarówno od sortowania przez scalanie, jak i samodostosowującej się wersji sortowania po drzewach. Niemniej jednak można wykazać, że jego wydajność w przypadku średnich przypadków wynosi O (N × log N), co odpowiada wydajności lepszych algorytmów sortowania. Co sprawia, że quicksort jest szczególnie interesujące, to fakt, że jego wydajność w przypadku średniej wielkości obejmuje bardzo małą stałą. W rzeczywistości, tylko przy porównaniach zliczania, wydajność czasowa szybkiego sortowania wynosi średnio nieco ponad 1,4N × log N. Biorąc pod uwagę fakt, że wymaga tylko niewielkiej stałej ilości dodatkowej przestrzeni dyskowej i pomimo gorszej wydajności w najgorszym przypadku, quicksort jest w rzeczywistości jednym z najlepszych znanych algorytmów sortowania i zdecydowanie najlepszym spośród wymienionych tutaj. Aby uzupełnić naszą krótką dyskusję na temat przedstawionych metod sortowania, powinniśmy zauważyć, że sortowanie bąbelkowe jest najgorszym z wielu, ma nawet przeciętne zachowanie O(N2). (Ten fakt jest jednak dość trudny do udowodnienia.) Wiele osób sprzeciwia się opisaniu bąbelkowego sortowania na kursach informatyki, ponieważ jego elegancja jest wystarczająco myląca, a studenci mogą faktycznie zostać zwabieni do używania go w praktyce.

Górna i dolna granica

Pokazaliśmy wcześniej, jak naiwny algorytm czasu liniowego do przeszukiwania uporządkowanej listy można ulepszyć do czasu logarytmicznego za pomocą wyszukiwania binarnego. Dokładniej, pokazaliśmy, że istnieje algorytm, który przeprowadza takie wyszukiwanie, używając nie więcej niż log2 N porównań w najgorszym przypadku na liście o długości N. Czy możemy zrobić lepiej? Czy w najgorszym przypadku możliwe jest wyszukanie elementu w książce telefonicznej o milionach wpisów z mniej niż 20 porównaniami? Czy można znaleźć algorytm dla problemu przeszukiwania uporządkowanej listy, który wymaga tylko $\sqrt{\log_2 N}$), a może tylko $\log_2(\log_2 N)$, porównania w najgorszym przypadku? Aby spojrzeć na te pytania z odpowiedniej perspektywy, wyobraźmy sobie, że każdy problem algorytmiczny znajduje się w miejscu, wyposażony w nieodłączne optymalne rozwiązanie, do którego dążymy. Następnie pojawia się ktoś z algorytmem, powiedzmy $O(N^3)$ (określany jako czas sześcienny), a tym samym zbliża się do pożądanego rozwiązania „z góry”. Mając ten algorytm jako dowód, wiemy, że problem nie może wymagać czasu działania wyższego niż sześcienny; nie może być gorszy niż O(N3). Później ktoś inny odkrywa lepszy algorytm, powiedzmy taki, który działa w czasie kwadratowym, zbliżając się w ten sposób do pożądanego optymalnego rozwiązania, również z góry. Jesteśmy teraz przekonani, że problem nie może być z natury gorszy niż $O(N^2)$, a poprzedni algorytm staje się przestarzały. Pytanie brzmi, jak daleko w dół mogą zajść te ulepszenia? Mając na uwadze metaforę „podejścia z góry” , mówi się, że odkrycie algorytmu nakłada górną granicę na problem algorytmiczny. Lepsze algorytmy sprowadzają najbardziej znane ograniczenie czasowe problemu w dół, bliżej nieznanej, nieodłącznej złożoności samego problemu. Pytania, które zadajemy, dotyczą dolnej granicy problemu. Jeśli potrafimy rygorystycznie udowodnić, że problem algorytmiczny P nie może być rozwiązany przez żaden algorytm, który w najgorszym przypadku wymaga mniej niż, powiedzmy, czasu kwadratowego, to ludzie próbujący znaleźć wydajne algorytmy do rozwiązania P mogą zrezygnować, jeśli i kiedy znajdą algorytm czasowy, bo nie ma mowy, żeby mogli zrobić to lepiej. Taki dowód stanowi dolne ograniczenie problemu algorytmicznego, ponieważ pokazuje, że żaden algorytm nie może poprawić ograniczenia $O(N^2)$. W ten sposób odkrycie sprytnego algorytmu pokazuje, że nieodłączna wydajność czasowa problemu nie jest gorsza niż niektóre ograniczenia, a odkrycie dowodu dolnego ograniczenia pokazuje, że nie jest on lepszy niż niektóre ograniczenia. W obu przypadkach odkryto właściwość problemu algorytmicznego, a nie właściwość konkretnego algorytmu. Może to zabrzmieć mylące, zwłaszcza że dolna granica problemu wymaga rozważenia wszystkich

algorytmów dla niego, podczas gdy górną granicę osiąga się, konstruując jeden konkretny algorytm i analizując jego wydajność czasową . Osiągnięcie dolnej granicy wydaje się niemożliwe. Jak udowodnić coś na temat wszystkich algorytmów? Skąd możemy mieć pewność, że ktoś nie odkryje bardzo subtelnego, ale wydajnego algorytmu, którego nie przewidzieliśmy? Nie są to łatwe pytania, ale być może kontemplacja poniższego przykładu da nam częściowe odpowiedzi.

Dolna granica wyszukiwania w książce telefonicznej

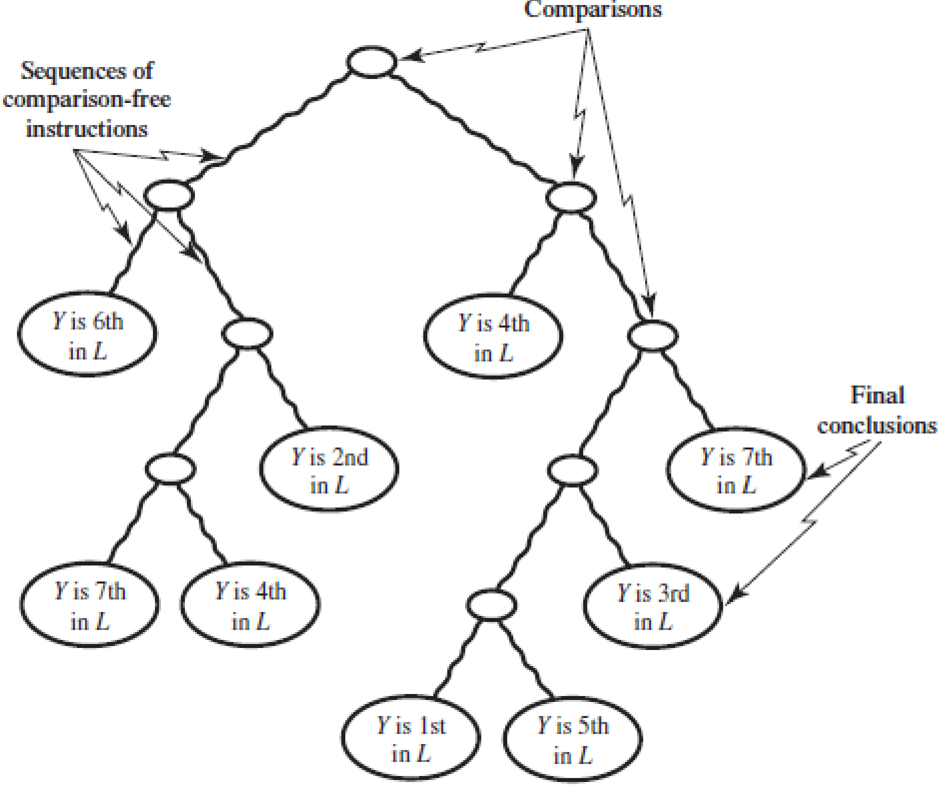
Zanim jednak przytoczymy przykład, powinniśmy ponownie podkreślić fakt, że wszelkie omówienie nieodłącznej złożoności czasowej, w tym dolnych granic, musi być przeprowadzone w odniesieniu do zestawu dozwolonych instrukcji podstawowych. Nikt nie może udowodnić, że problem wyszukiwania wymaga więcej niż jednej jednostki czasu, jeśli podstawowe instrukcje, takie jak:

lista wyszukiwania L dla pozycji Y

są dozwolone. Dlatego zasady gry muszą być dokładnie określone, zanim podejmiemy jakąkolwiek próbę udowodnienia niższych granic. Teraz chcemy pokazać, że wyszukiwanie binarne jest optymalne. Innymi słowy, chcielibyśmy udowodnić, że generalnie nie możemy znaleźć nazwiska w książce telefonicznej o długości N w porównaniach mniejszych niż log2 N w najgorszym przypadku; 20 porównań to minimum dla miliona nazwisk, 30 dla miliarda, 60 dla miliarda miliardów i tak dalej. Mówiąc bardziej ogólnie, nie ma algorytmu dla problemu przeszukiwania uporządkowanej listy, którego wydajność w najgorszym przypadku w czasie jest lepsza niż logarytmiczna.

Zasady gry są tutaj dość proste. Jedynym sposobem, w jaki proponowany algorytm może wydobyć jakiekolwiek informacje z danych wejściowych, jest przeprowadzenie dwukierunkowych porównań. Algorytm może porównać między sobą dwa elementy z listy wejściowej L lub element z L z elementem wejściowym Y, którego pozycja w L jest poszukiwana. Żadne inne zapytania nie mogą być kierowane do wejść L i Y . Jednak nie ma ograniczeń dotyczących instrukcji niezwiązanych z L lub Y ; każdy z nich może zostać uwzględniony, a każdy będzie kosztować tylko jedną jednostkę czasu. Aby pokazać, że dowolny algorytm dla tego problemu wymaga porównań log2N, będziemy argumentować następująco. Mając dany algorytm, dowolny algorytm, pokażemy, że nie może on w żaden sposób poprawić deklarowanej dolnej granicy log2N dla problemu. W związku z tym załóżmy, że wymyśliliśmy algorytm A dla problemu przeszukiwania uporządkowanej listy, który jest oparty na porównaniach dwukierunkowych. Udowodnimy, że istnieją listy wejściowe o długości N, na których algorytm A z konieczności przeprowadzi porównania $\log_2 N$.

W tym celu przyjrzyjmy się zachowaniu A na typowej liście L o długości N. Aby ułatwić sobie życie, pozostańmy przy szczególnym przypadku, w którym L składa się dokładnie z liczb 1, 2, 3, ... , N w kolejności i gdzie Y jest jedną z tych liczb. Możemy również bezpiecznie założyć, że wszystkie porównania mają tylko dwa wyniki, „mniejsze niż” lub „więcej niż”, ponieważ jeśli porównanie daje „równe”, Y zostało znalezione (chyba że przeprowadzono jakieś bezużyteczne porównanie elementu z samym sobą, w którym to przypadku algorytm może się bez niego obejść). Algorytm A, pracujący na naszej specjalnej liście L, może wykonać kilka innych akcji, ale w końcu osiąga swoje pierwsze porównanie. Następnie przypuszczalnie rozgałęzia się w jednym z dwóch kierunków, w zależności od (dwukierunkowego) wyniku. W każdym z nich A może jeszcze trochę popracować przed kolejnym porównaniem, które ponownie dzieli zachowanie A na dwie dalsze możliwości. Zachowanie A można zatem schematycznie opisać jako drzewo binarne w którym krawędzie odpowiadają możliwym sekwencjom działań nieporównywania, a węzły odpowiadają porównaniom i ich wynikom rozgałęziania.



To drzewo przechwytuje zachowanie A dla wszystkich możliwych list składających się z liczb 1, 2, 3, ..., N. Teraz drzewo jest zdecydowanie skończone, ponieważ A musi zakończyć się na wszystkich dozwolonych danych wejściowych, a jego liście odpowiadają osiągnięciu ostatecznej decyzji dotyczącej pozycji Y w L, czyli innymi słowy, dotyczącej wartości Y z wśród 1, 2, ... , N. Oczywiście nie ma dwóch różnych wartości Y, które mogą być reprezentowane przez ten sam liść, ponieważ oznaczałoby to, że A daje takie same dane wyjściowe (czyli pozycję w L) dla dwóch różnych wartości Y - oczywiście absurd. Ponieważ Y ma N możliwych wartości, drzewo musi mieć co najmniej N możliwych liści. (Może mieć więcej, ponieważ może istnieć więcej niż jeden sposób wyciągnięcia tego samego wniosku na temat pozycji Y.) Używamy teraz następującego standardowego faktu o drzewach binarnych, których prawdziwość powinieneś spróbować ustalić:

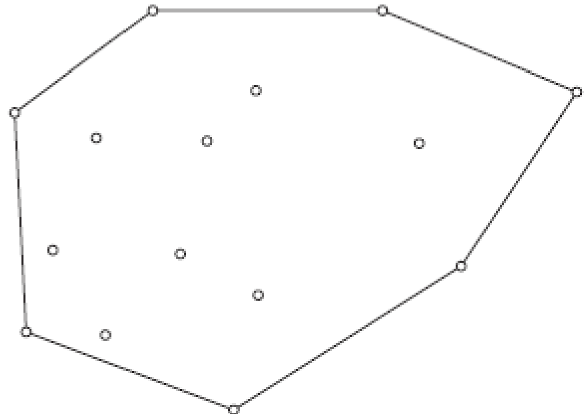
Każde drzewo binarne mające co najmniej N liści ma głębokość co najmniej $\log_2 N$; oznacza to, że drzewo zawiera co najmniej jedną ścieżkę od korzenia, której długość wynosi $\log_2 N$ lub więcej. Wynika z tego, że nasze drzewo porównawcze zawiera ścieżkę o długości co najmniej $\log_2 N$. Ale ścieżka w drzewie odpowiada dokładnie wykonaniu proponowanego algorytmu A na jakiejś liście wejściowej o długości N, przy czym węzły oznaczają porównania dokonane w tym właśnie wykonanie . Stąd, jeśli w drzewie istnieje ścieżka o długości co najmniej $\log_2 N$, wynika z tego, że istnieje pewna lista wejściowa, która wymaga od A wykonania co najmniej log2N porównań. To kończy dowód, że log2N jest dolnym ograniczeniem liczby porównań, a zatem, że algorytmiczny problem wyszukiwania w uporządkowanej liście ma dolne ograniczenie w czasie logarytmicznym. Zwróć uwagę, że argument został sformułowany na tyle ogólnie, aby można go było zastosować do dowolnego proponowanego algorytmu, o ile używa porównań jako jedynego środka do wyodrębnienia informacji z listy wejściowej.

Zamknięte problemy i luki algorytmiczne

Dolne granice można ustalić dla wielu innych problemów algorytmicznych. Na przykład wyszukiwanie na nieuporządkowanej liście jest kwintesencją problemu czasu liniowego i można łatwo wykazać, że w najgorszym przypadku wymaga N porównań. Dlatego ma dolną granicę O(N) dla algorytmów opartych na porównaniach. Możesz chcieć wypracować argument na to. Kilka problemów, o których mówiliśmy, może być interpretowanych jako odmiany nieuporządkowanego wyszukiwania i w konsekwencji są one również ograniczone od dołu przez O(N). Należą do nich problem min&max, proste sumowanie wynagrodzeń, maksymalna odległość wielokąta i tak dalej. Zauważ, że dla każdego z nich rzeczywiście dostarczyliśmy algorytmy czasu liniowego. Oznacza to, że górna i dolna granica faktycznie się spotykają (poza możliwymi różnymi czynnikami stałymi). Innymi słowy, te problemy algorytmiczne są zamknięte, jeśli chodzi o oszacowania czasu Big-O. Mamy algorytm liniowy i wiemy, że nie możemy zrobić nic lepszego. Wyszukiwanie w uporządkowanej liście, jak pokazaliśmy, jest również zamkniętym problemem; ma górną i dolną granicę czasu logarytmicznego. Sortowanie też jest zamknięte: z jednej strony mamy algorytmy, takie jak mergesort, heapsort lub samodostosowująca się wersja treesort, które są O(N × log N), a z drugiej możemy udowodnić O(N × log N). N) dolna granica sortowania listy o długości N. W obu tych przypadkach granice są oparte na modelu porównawczym, w którym informacje o danych wejściowych uzyskuje się tylko przez porównania dwukierunkowe. Wiele problemów algorytmicznych nie ma jednak jeszcze właściwości zamknięcia. Ich górna i dolna granica się nie spotykają. W takich przypadkach mówimy, że powodują one luki algorytmiczne, przy czym najlepiej znana górna granica różni się od (a zatem jest wyższa) od najlepiej znanej dolnej granicy. W Części 4 przedstawiliśmy kwadratowy algorytm znajdowania minimalnych linii kolejowych (problem minimalnego drzewa rozpinającego), ale najbardziej znane dolne ograniczenie jest liniowe. To znaczy, chociaż możemy udowodnić, że problem wymaga czasu O(N) (tu N jest liczbą krawędzi w linii kolejowej wykresu, a nie liczby węzłów), nikt nie zna algorytmu czasu liniowego, a zatem problem nie jest zamknięty. W Części 7 zobaczymy kilka uderzających przykładów luk algorytmicznych, które są niedopuszczalnie duże. Na razie wystarczy zdać sobie sprawę, że jeśli z jakiegoś problemu powstaje luka algorytmiczna, to niedoskonałości nie tkwi w samym problemie, ale w naszej wiedzy na jego temat. Nie udało nam się albo znaleźć najlepszego algorytmu do tego, albo udowodnić, że lepszy algorytm nie istnieje, albo w obu przypadkach.

Zabarykadowanie śpiących tygrysów: przykład

Poniższy algorytm wykorzystuje sortowanie w dość nieoczekiwany sposób i dziedziczy również granice złożoności. Załóżmy, że mamy do czynienia z N śpiącymi tygrysami i aby uniknąć zjedzenia, gdy jeden lub więcej z nich się obudzi, jesteśmy zainteresowani zbudowaniem wokół nich ogrodzenia. Algorytmicznie otrzymujemy dokładną lokalizację tygrysów i chcemy znaleźć najmniejszy wielokąt, który je wszystkie otacza. Oczywiście problem sprowadza się do znalezienia minimalnej sekwencji lokalizacji niektórych tygrysów, które połączone liniowymi fragmentami ogrodzenia (które nazwiemy segmentami linii) obejmą wszystkie pozostałe (patrz rysunek 1). Ta obudowa nazywana jest wypukłą kadłubem zbioru punktów. Zauważ, że ten problem został przedstawiony w zoologicznej postaci tylko po to, by brzmiał trochę bardziej zabawnie. Problem wypukłego kadłuba jest właściwie jednym z podstawowych problemów pojawiających się w grafice komputerowej. Znalezienie szybkich algorytmów dla tego problemu i wielu innych problemów w geometrii obliczeniowej może mieć dużą różnicę w szybkości i możliwości zastosowania wielu aplikacji graficznych. Przedstawiony teraz algorytm opiera się na następującej prostej obserwacji. Aby jakiś odcinek łączący dwa punkty był częścią wypukłego kadłuba, konieczne i wystarczające jest, aby wszystkie pozostałe punkty znajdowały się po tej samej stronie (a raczej jego przedłużenia do pełnej linii).



Ta obserwacja daje początek prostemu algorytmowi: rozważ po kolei każdy potencjalny odcinek linii i sprawdź, czy wszystkie N − 2 punkty, które nie znajdują się na nim, są po jednej jego stronie. Test decydujący, do której strony linii należy dany punkt, można łatwo przeprowadzić w stałym czasie przy użyciu elementarnej geometrii analitycznej. Ponieważ jest N punktów, to jest N2 odcinków potencjału, z których jeden może łączyć każdą parę punktów i każdy z nich musi być porównywany z N − 2 punktami. Daje to sześcienny (czyli algorytm O(N3)), jak łatwo zauważyć. Jest jednak sposób na zrobienie tego znacznie lepiej. Oto główne kroki w proponowanym algorytmie:

(1) znajdź „najniższy" punkt P_1;

(2) posortuj pozostałe punkty według wielkości kąta, jaki tworzą z osią poziomą w połączeniu z P_1 i niech otrzymaną listą będzie P2, ... , PN ;

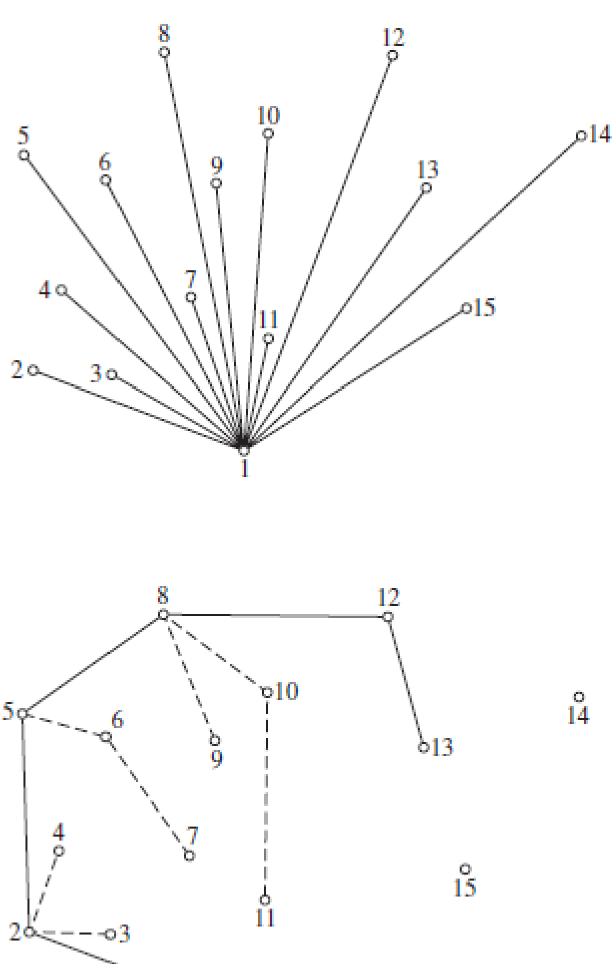
(3) wystartować z P_1 i P_2 w obecnym kadłubie;

(4) dla I od 3 do N wykonaj następujące czynności:

(4.1) wstępnie dodać P_I do aktualnego kadłuba;

(4.2) działa wstecz przez bieżący kadłub, eliminując punkt PJ, jeśli dwa punkty P1</> i PI leżą po różnych stronach linii między P_{J-1} i P_J , i kończąc skanowanie wstecz, gdy P_J, którego nie trzeba eliminować napotkany.

Może trochę trudno zobaczyć, co robi algorytm, ale rzut oka na rysunki 2 i 3 powinien pomóc.



Rysunek 2 pokazuje punkty posortowane po kroku (2) w kolejności, w jakiej będą brane pod uwagę w kroku (4), a rysunek 3 pokazuje kilka pierwszych dodanych i wyeliminowanych w kroku (4). Na rysunku 3 właśnie dodano punkt P_{13}, a następny krok będzie uwzględniał P_{14}. Jednakże, ponieważ linia między P_{12} i P_{13} przebiega między P_1 i P_{14} punkt P_{13} zostanie wyeliminowany, sprowadzając kadłub wokół punktów powyżej P13. Choć udowodnienie poprawności tego algorytmu jest ćwiczeniem pouczającym, bardziej interesuje nas jego efektywność czasowa. Przyjmiemy zatem, że algorytm jest poprawny i skoncentrujemy się na jego analizie czasowej. Łatwo zauważyć, że krok (1), który polega na prostym wyszukaniu punktu, który leży najniżej na obrazie, czyli tego, który ma najmniejszą współrzędną pionową, zajmuje czas liniowy. Teraz krok (2), sortowanie, można przeprowadzić przez dowolne wydajne sortowanie. W związku z tym, ponieważ obliczenie kąta lub porównanie dwóch kątów można przeprowadzić w stałym czasie, krok (2) zajmuje całkowity czas $O(N \times \log N)$. Interesująca część dotyczy kroku (4). Wygląda na to, że zagnieżdżona struktura pętli kroku (4) zapewnia wydajność w czasie kwadratowym. Jednak właściwym sposobem analizy tej części algorytmu jest zlekceważenie jego struktury i rozliczenie czasu punkt po punkcie. Zauważ, że krok (4.2) eliminuje tylko punkty i zatrzymuje się, gdy napotkany zostanie punkt, którego nie należy eliminować. Teraz, ponieważ żaden punkt nie zostanie wyeliminowany więcej niż raz,łączna liczba punktów branych pod uwagę w zagnieżdżonej pętli kroku (4.2) nie może być większa niż $O(N)$. Ponieważ wszystkie pozostałe części kroku (4) również zabierają nie więcej niż czas liniowy, krok (4) w całości zajmuje czas liniowy. Całkowity czas działania algorytmu wynosi zatem:

Krok (1) O(N)

Krok (2) O(N × log N)

Krok (3) O(1)

Krok (4) O(N)

Razem O(N × log N)

To trochę zaskakujące, że sortowanie ma w ogóle coś wspólnego ze znalezieniem wypukłego kadłuba. Bardziej zaskakujące jest to, że sortująca część algorytmu jest w rzeczywistości dominującą częścią, jeśli chodzi o złożoność obliczeniową.

Badania efektywności algorytmów

Znalezienie wydajnych algorytmów do rozwiązywania problemów algorytmicznych jest jednym z najczęstszych tematów badawczych w informatyce. Rzeczywiście, prawie wszystkie zagadnienia omawiane w tym rozdziale są przedmiotem znacznych wysiłków badawczych, a większość z nich została zebrana pod ogólnym terminem teoria konkretnej złożoności. W tej dziedzinie ludzie są zainteresowani opracowywaniem i wykorzystywaniem struktur danych i metod algorytmicznych w celu ulepszania istniejących algorytmów lub opracowywania nowych. Wiele genialnych pomysłów trafia do wyrafinowanych algorytmów, zapewniając czasami zaskakujące skrócenie czasu działania. Najczęściej odbywa się to z praktycznym celem szybszego rozwiązania problemu za pomocą komputera. Jednak czasami siłą napędową jest po prostu chęć przygwożdżenia nieodłącznej złożoności problemu o rząd wielkości, nawet jeśli wynika to z praktycznego punktu widzenia, nie są lepsze niż te znane wcześniej. Dobrym tego przykładem jest problem układu linii kolejowych (drzewo opinające) z Części 4. Przedstawiliśmy dla tego algorytm w czasie kwadratowym, który z pewnym wysiłkiem można ulepszyć do O(N × log N). Ze wszystkich praktycznych celów ten ulepszony algorytm lub dowolny z wielu innych algorytmów o podobnej złożoności działa dobrze. Jednak teoretycy złożoności nie są z tego zadowoleni, ponieważ najlepiej znana dolna granica jest liniowa. Chcą wiedzieć, czy rzeczywiście można znaleźć algorytm czasu liniowego dla tego problemu. Ostatnio, przy użyciu dość sprytnych i skomplikowanych technik, górna granica została zbliżona do O(N). W szczególności istnieje algorytm, który działa w czasie ograniczonym przez O(f (N) × N), gdzie f (N) jest funkcją, która rośnie niewiarygodnie wolno - dużo, dużo wolniej niż, powiedzmy, log N. Poniższa tabela pokazuje najmniejsze Ns dla kilku pierwszych wartości tej funkcji:

Najmniejsze N : takie, że f (N) to

4 : 2

16 : 3

64 000 : 4

znacznie więcej niż całkowita liczba cząstek w znanym wszechświecie; 5

absolutnie niewyobrażalne : 6

Oczywiście, dla wszystkich praktycznych celów algorytm O(f (N) × N) jest naprawdę liniowy; wartość f (N) wynosi 5 lub mniej dla dowolnej liczby, którą kiedykolwiek będziemy zainteresowani. Musimy jednak pamiętać, że bez względu na to, jak wolno rośnie f (N), w końcu stanie się ono większe niż jakakolwiek stała. Stąd od pewnego momentu, to znaczy dla wszystkich wystarczająco dużych N, każdy dany algorytm czasu liniowego przewyższa algorytm O(f (N) × N) i mówimy, że pierwszy jest

asymptotycznie lepszy od drugiego. Tym samym problem drzewa opinającego jest nadal otwarty, ponieważ wciąż tworzy lukę algorytmiczną, z której badacze nie zrezygnowali. Mamy nadzieję, że w dającej się przewidzieć przyszłości problem zostanie w taki czy inny sposób zamknięty. Albo zostanie odkryty algorytm czasu liniowego, albo zostanie znaleziony dowód nieliniowości dolnej granicy, pasujący do znanej górnej granicy. Jak wspomniano wcześniej, dolne granice są bardzo trudne do znalezienia, a dla wielu problemów nikt nie zna żadnych nieliniowych granic dolnych. Innymi słowy, pomimo tego, że najlepsze algorytmy dla niektórych problemów są kwadratowe, sześcienne lub gorzej, nikt nie jest w stanie udowodnić, że nie istnieje dla nich jakiś algorytm czasu liniowego, który czekałby na odkrycie. Metody udowadniania nieliniowych dolnych granic są niezwykle rzadkie i badacze dokładają wszelkich starań, aby je znaleźć. Wydajność dla przypadków średnich prowadzi do kolejnego trudnego kierunku badań, a wciąż istnieje wiele znanych algorytmów, dla których nie przeprowadzono zadowalających analiz przypadków średnich. W ogóle nie omawialiśmy tutaj złożoności przestrzennej, ale warto stwierdzić, że osiągnięcie dobrych górnych i dolnych granic wymagań dotyczących przestrzeni pamięciowej problemów algorytmicznych jest również przedmiotem wielu badań nad konkretną złożonością. Ludzi interesują nie tylko oddzielne granice w czasie i przestrzeni, ale także wspólna czasowo-przestrzenna złożoność problemu. Może być możliwe osiągnięcie, powiedzmy, górnego ograniczenia w przestrzeni liniowej na problem za pomocą jakiegoś algorytmu i, powiedzmy, górnego ograniczenia w czasie kwadratowym za pomocą innego algorytmu, ale to nie implikuje istnienia algorytmu, który osiąga oba jednocześnie. Możliwe, że problem wiąże się z kompromisem między czasem a przestrzenią, co oznacza, że płacimy za oszczędność czasu większą przestrzenią i za gospodarkę przestrzeni większą ilością czasu. W takich przypadkach badacze starają się udowodnić, że istnieje taki kompromis. Zwykle przybiera to formę dowodu, że działanie dowolnego algorytmu rozwiązującego problem spełnia pewne równanie, które występuje jako ograniczenie dolne lub ograniczenie górne (lub oba). Równanie zazwyczaj ujmuje trójczynnikową zależność między wejściową długością N a (najgorszym przypadkiem) czasem działania i przestrzenią pamięci dowolnego algorytmu rozwiązania. Na przykład załóżmy, że następujące równanie zostało ustalone jako górna i dolna granica złożoności czasowo-przestrzennej problemu P:

$S^2 \times T = O(N^3 \times (\log N)2)$

Oznacza to, że jeśli chcemy spędzić $O(N^3)$ czasu, możemy rozwiązać problem używając tylko przestrzeni O(log N), podczas gdy jeśli nalegamy na spędzanie nie więcej niż O(N2) czasu, potrzebowalibyśmy O(√ N × log N) przestrzeń. Kolejne dwa rozdziały skoncentrują się na teorii złożoności złych wiadomości dla nas. Omówią również pojęcia, które są jeszcze bardziej niezawodne niż notacja duże-O.

Ograniczenia i wytrzymałość

Nieefektywność i nieusuwalność

W poprzedniej części widzieliśmy, że niektóre problemy algorytmiczne dopuszczają rozwiązania, które są znacznie bardziej efektywne czasowo niż ich naiwne odpowiedniki. Widzieliśmy na przykład, że możliwe jest przeszukanie posortowanej listy według czasu logarytmicznego, co po doprecyzowaniu oznacza, że w najgorszym przypadku możemy wyszukać nazwisko w książce telefonicznej o milionach wpisów, z zaledwie 20 porównaniami, nie milion. W podobnym duchu sortowanie nieposortowanej książki telefonicznej o milionach wpisów można osiągnąć za pomocą zaledwie kilku milionów porównań, a nie wielu miliardów, ponieważ istnieją algorytmy sortowania O(N × log N), które przewyższają naiwne algorytmy kwadratowe. W tym momencie możesz nie być pod wrażeniem. Możesz twierdzić, że jesteś wystarczająco „bogaty", aby pozwolić sobie na milion porównań przy przeszukiwaniu listy. Albo że kilka dodatkowych sekund czasu komputera nie ma znaczenia, a zatem wyszukiwanie liniowe jest tak samo dobre jak wyszukiwanie binarne. Argument ten zyskuje na wiarygodności, gdy zdamy sobie sprawę, że to nie człowiek, ale komputer wykonuje tę nudną i pozbawioną wyobraźni robotę polegającą na przerzucaniu wszystkich nazwisk w książce. Podobny argument można również przedstawić za sortowaniem, zwłaszcza jeśli aplikacja jest taka, że sortowanie ma być wykonywane rzadko, a sortowane listy nigdy nie zawierają więcej niż, powiedzmy, miliona pozycji. Biorąc pod uwagę takie nastawienie, pytania o luki algorytmiczne również stają się nieciekawe. Po znalezieniu dość dobrego algorytmu dla palącego problemu algorytmicznego, możemy nie być zainteresowani lepszymi algorytmami lub dowodami na to, że one nie istnieją. Celem tej części jest pokazanie, że nie zawsze można przyjąć podejście „pogódźmy się z tym, co mamy". Zobaczymy, że w wielu przypadkach rozsądne algorytmy, powiedzmy liniowe lub kwadratowe, w ogóle nie istnieją. Wykazano, że najlepsze algorytmy dla wielu ważnych problemów algorytmicznych wymagają ogromnej ilości czasu lub miejsca w pamięci, co czyni je zupełnie bezużytecznymi.

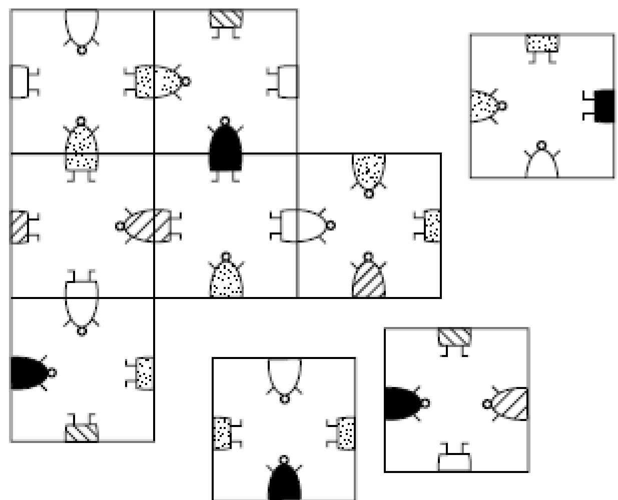
Ponowna wizyta w wieżach Hanoi

Przypomnijmy sobie problem Wież Hanoi z części 2, w którym poproszono nas o stworzenie sekwencji ruchów jednopierścieniowych w celu przeniesienia N pierścieni z jednego z trzech kołków na drugi zgodnie z pewnymi zasadami. Zachęcamy do przeprowadzenia prostej analizy czasowej opisanego tam rekurencyjnego ruchu rozwiązania, podobnej do analizy przeprowadzonej w Części 6 dla procedury min&max. Pokaże, że liczba ruchów jednopierścieniowych generowanych przez algorytm dla przypadku N-pierścieniowego wynosi dokładnie 2^N − 1, czyli o jeden mniej niż 2 × 2 × 2 × . . . × 2, przy czym 2 pojawia się N razy. Ponieważ N pojawia się w wykładniku, taką funkcję nazywamy wykładniczą. Można wykazać, że 2^N − 1 jest również dolnym ograniczeniem wymaganej liczby ruchów do rozwiązania problemu, tak że nasze rozwiązanie jest naprawdę optymalne i nie możemy zrobić nic lepszego. Czy ta wiadomość jest dobra czy zła? Cóż, odpowiadając na pytanie w sposób pośredni, gdyby hinduscy księża, skonfrontowani ze sprawą 64 pierścieni, mieli odświeżyć swój akt i przesunąć milion pierścieni na sekundę, to i tak zajęłoby im to ponad pół miliona lat proces! Jeśli, bardziej realistycznie, mieliby przesunąć jeden pierścień co 10 sekund, zajęłoby im to znacznie ponad pięć bilionów lat, aby wykonać to zadanie. Nic dziwnego, że wierzyli, że świat się skończy przedtem! Wydaje się zatem, że problem Wież Hanoi, przynajmniej dla 64 lub więcej pierścieni, jest beznadziejnie czasochłonny. Chociaż to stwierdzenie wydaje się trudne do zakwestionowania, może powodować wrażenie, że trudność wynika z chęci wydrukowania całej sekwencji ruchów, a ponieważ wymaganych jest beznadziejnie wiele ruchów, oczywiście znalezienie ich i wydrukowanie zajmie beznadziejnie dużo czasu. Moglibyśmy zatem ulec pokusie, by oczekiwać, że tak niszczycielska wydajność czasowa wystąpi tylko w przypadku problemów, których wyniki są nadmiernie długie. Aby przekonać samych siebie, że tak nie jest, pouczające jest rozważenie problemów typu tak/nie; to znaczy problemy algorytmiczne,

które nie dają żadnych „rzeczywistych” wyników poza „tak” lub „nie”. Są one czasami nazywane problemami decyzyjnymi, ponieważ ich celem jest jedynie rozstrzygnięcie, czy dana właściwość obowiązuje dla ich danych wejściowych. Większość tego rozdziału (i następnego) będzie poświęcona problemom decyzyjnym.

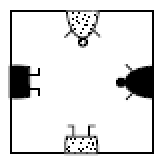
Problem z łamigłówką małp: przykład

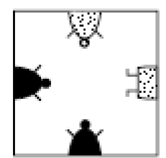
W pewnym momencie życia mogłeś natknąć się na jedną lub więcej wersji bardzo frustrującej małpiej układanki



Składa się z dziewięciu kwadratowych kart, których boki są nadrukowane górną i dolną połówką kolorowych małp. Celem jest ułożenie kart w formie kwadratu 3 na 3 tak, aby połówki pasowały do siebie, a kolory były identyczne wszędzie tam, gdzie stykają się krawędzie. W ogólnym zagadnieniu algorytmicznym związanym z tą zagadką mamy dane (opisy) N kart, gdzie N jest pewną liczbą kwadratową, powiedzmy, N to M^2, a problem wymaga wykazania, jeśli to możliwe, układu kwadratów M na M karty N, aby kolory i połówki zachowywały się zgodnie z opisem. Założymy, że karty są zorientowane, co oznacza, że krawędzie mają ustalone kierunki: „w górę”, „w dół”, „w prawo” i „w lewo”, tak że nie można ich obracać. Skoncentrujemy się na pozornie prostszej wersji tak/nie, która po prostu pyta, czy istnieje taki układ M na M, bez pytania o to, by rzeczywiście go wystawić. Nie jest trudno znaleźć naiwne rozwiązanie tego problemu. Musimy tylko zauważyć, że każde dane wejściowe obejmują tylko skończoną liczbę kart i że istnieje tylko skończenie wiele miejsc do wypełnienia. W związku z tym istnieje tylko skończenie wiele różnych sposobów ułożenia kart wejściowych w kwadrat M na M. Co więcej, dany układ można łatwo przetestować pod kątem legalności (to znaczy, że wszystkie karty wejściowe są rzeczywiście używane i że połówki i kolory pasują), po prostu biorąc pod uwagę każdą kartę i każdą ze stykających się krawędzi po kolei. W związku z tym można zaprojektować algorytm, który będzie działał na jego drodze wszystkie możliwe ustalenia, zatrzymanie się i powiedzenie „tak”, jeśli dane rozwiązanie jest zgodne z prawem, oraz zatrzymanie się i powiedzenie „nie”, jeśli wszystkie ustalenia zostały rozważone i wszystkie zostały uznane za nielegalne. Oczywiście możliwe jest uczynienie tego podejścia mniej brutalnym, unikając konieczności wyraźnego sprawdzania rozszerzeń częściowego porozumienia, które zostało już udowodnione jako niezgodne z prawem. Wymagane jest prowadzenie ksiąg rachunkowych, aby upewnić się, że wszystkie możliwe ustalenia są rzeczywiście brane pod uwagę i że żaden nie jest rozpatrywany dwukrotnie, ale nie będziemy tutaj rozwodzić się nad szczegółami algorytmu -bardziej interesuje nas jego czas. Załóżmy, że N wynosi 25, co oznacza, że końcowy kwadrat ma mieć rozmiar 5 na 5. Załóżmy również, że mamy komputer zdolny do skonstruowania i oceny miliarda układów na sekundę (tj. jednego układu na

nanosekundę) , w tym wszystkie zaangażowane księgi rachunkowe. To całkiem rozsądne założenie, biorąc pod uwagę dzisiejsze komputery. Pytanie brzmi: ile czasu zajmie algorytmowi w najgorszym przypadku (czyli wtedy, gdy nie ma porozumienia prawnego, aby sprawdzić wszystkie możliwe ustalenia)? Jeśli arbitralnie ponumerujemy lokalizacje w siatce 5 na 5, istnieje oczywiście 25 możliwości wyboru karty, która ma zostać umieszczona w pierwszej lokalizacji. Po umieszczeniu jakiejś karty w tej lokacji masz do wyboru 24 karty dla drugiej lokacji, 23 dla trzeciej i tak dalej. Łączna liczba aranżacji może zatem wynieść: $25 \times 24 \times 23 \times \ldots \times 3 \times 2 \times 1$ liczba oznaczona przez 25! i nazwano 25 silnią. To, co jest zdumiewające, to wielkość tej niewinnie wyglądającej liczby; zawiera 26 cyfr, co nie wydaje się niepokojące, dopóki nie zdamy sobie sprawy, że nasz komputer z miliardem aranżacji na sekundę zajmie znacznie ponad 490 milionów lat, aby przebić się przez wszystkie 25! ustalenia. Jeśli po prostu zwiększymy rozmiar kwadratu o jeden, przechodząc od kwadratu 5 na 5 do kwadratu 6 na 6, tak aby N wynosiło 36, sytuacja stanie się znacznie gorsza. Czas potrzebny na przejście wszystkich 36! wartości byłyby niewyobrażalnie długie, daleko, daleko, DUŻO dłuższe niż czas, który upłynął od Wielkiego Wybuchu. Oczywiście, poszczególne łamigłówki z małpami można przygotować w sposób ułatwiający życie. (Aby podać kilka skrajnych przykładów, jeśli wszystkie karty są albo identyczne z kartą z rysunku (a) lub identyczne z kartą z rysunku (b), pytanie ma trywialne odpowiedzi.)


Arrangements always exist
(a)


Arrangements never exist
(b)

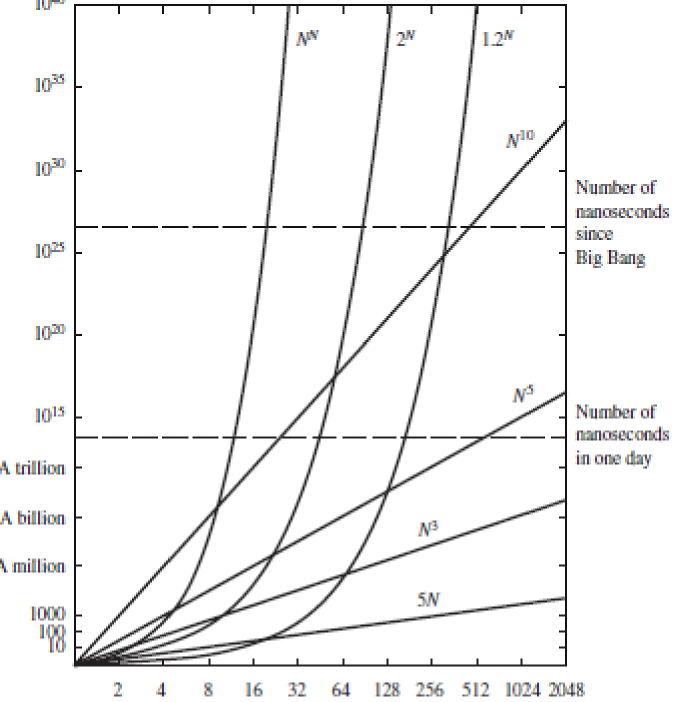
Co więcej, inteligentniejsza wersja algorytmu , który nie uwzględnia rozszerzeń nielegalnych układów częściowych, będzie działał znacznie lepiej w przypadku wielu łamigłówek wejściowych. Jednak nasza gra to analiza najgorszego przypadku. Projektant puzzli faktycznie dąży do zestawów kart, które dopuszczają wiele rozwiązań częściowych (w których części kwadratu są legalnie pokryte kartami), ale tylko bardzo niewiele rozwiązań kompletnych - może tylko jedno. To właśnie powstrzymuje problem przed dopuszczeniem do szybkich i łatwych rozwiązań. Dlatego nawet mniej naiwna wersja algorytmu będzie wykazywać podobnie katastrofalne zachowanie w najgorszym przypadku. Dlatego rozwiązanie brute-force jest zupełnie bezużyteczne, nawet w przypadku bardzo małej wersji 5 na 5 (lub powiedzmy 10 na 10, jeśli używana jest wersja mniej naiwna, a my jesteśmy skłonni zadowolić się przeciętnym przypadkiem wydajność). Jeśli teraz oczekujesz, że zostanie zaprezentowane naprawdę sprytne rozwiązanie, które pokaże, jak naprawdę rozwiązać problem w użyteczny sposób, czeka Cię rozczarowanie. Jedyne znane rozwiązania, które są lepsze od omówionych, nie są na tyle lepsze, aby były rozsądne; jeśli N wynosi 25, w najgorszym przypadku nadal wymagałyby wielu, wielu lat obliczeń dla pojedynczego przypadku, a jeśli N wynosi 36, cóż, . . . zapomnij o tym . . . Czy istnieje jakieś ukryte rozwiązanie problemu z małpią układanką, które byłoby praktyczne dla rozsądnej liczby kart, powiedzmy do 225? Rozumiemy przez to pytanie, czy istnieje prosty sposób rozwiązania problemu, którego jeszcze nie odkryliśmy. Być może istnieje układ dokładnie wtedy, gdy liczba różnych kart jest wielokrotnością 17, z jakiegoś dziwnego powodu. Odpowiedź na to pytanie brzmi „prawdopodobnie nie, ale nie jesteśmy do końca pewni”. Omówimy tę kwestię dalej po zbadaniu ogólnego zachowania takich niepraktycznych algorytmów, jak ten właśnie opisany.

Rozsądny kontra nierozsądny czas

Funkcja silni N! rośnie w tempie, które jest o rzędy wielkości większe niż tempo wzrostu którejkolwiek z funkcji wymienionych w poprzednich rozdziałach. Rośnie znacznie szybciej niż na przykład funkcje

liniowe lub kwadratowe iw rzeczywistości z łatwością przewyższa wszystkie funkcje postaci N^K , dla dowolnego ustalonego K. Prawdą jest, że na przykład N¹⁰⁰⁰ jest większe niż N! dla wielu wartości N (dokładnie dla wszystkich N do 1165). Jednak dla dowolnego K istnieje pewna wartość N (1165, jeśli K wynosi 1000), powyżej której funkcja N! pozostawia N^K daleko w tyle, bardzo, bardzo szybko. Inne funkcje wykazują podobnie niedopuszczalne tempo wzrostu. Na przykład funkcja N^N oznaczająca N × N × N × . . . × N z N wystąpień N, rośnie nawet szybciej niż N!. Funkcja 2^N , czyli 2 × 2 × 2 × . . . × 2, z N wystąpień 2, rośnie wolniej niż N!, ale jest również uważany za „złą" funkcję; wciąż rośnie znacznie szybciej niż funkcje N^K. Jeśli N wynosi 20, wartość 2^N wynosi około miliona, a jeśli N wynosi 30, to około miliarda. (Dzieje się tak, ponieważ 2^N odnosi się do N dokładnie tak, jak N do $\log_2 N$.) Jeśli N wynosi 300, liczba 2^N jest miliardy razy większa niż liczba protonów w całym znanym wszechświecie. Rysunki poniższe ilustrują względne tempo wzrostu niektórych z tych funkcji.

	Function \ N	20	60	100	300	1000
Polynomial	$5N$	100	300	500	1500	5000
	$N \times \log_2 N$	86	354	665	2469	9966
	N^2	400	3600	10,000	90,000	1 million (7 digits)
	N^3	8000	216,000	1 million (7 digits)	27 million (8 digits)	1 billion (10 digits)
Exponential	2^N	1,048,576	a 19-digit number	a 31-digit number	a 91-digit number	a 302-digit number
	$N!$	a 19-digit number	an 82-digit number	a 161-digit number	a 623-digit number	unimaginably large
	N^N	a 27-digit number	a 107-digit number	a 201-digit number	a 744-digit number	unimaginably large

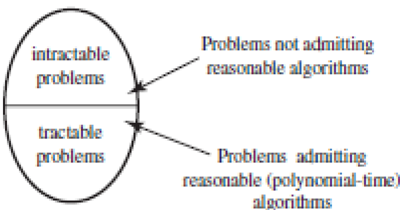


Uderzającą ilustracją różnic między tymi funkcjami są czasy działania algorytmów. Rysunek przedstawia rzeczywiste czasy działania kilku hipotetycznych algorytmów rozwiązywania problemu łamigłówki małp dla różnych wartości N.

	Function \ N	20	40	60	100	300
Polynomial	N^2	1/2500 millisecond	1/625 millisecond	1/278 millisecond	1/100 millisecond	1/11 millisecond
	N^5	1/300 second	1/10 second	78/100 second	10 seconds	40.5 minutes
Exponential	2^N	1/1000 second	18.3 minutes	36.5 years	400 billion centuries	a 72-digit number of centuries
	N^N	3.3 billion years	a 46-digit number of centuries	an 89-digit number of centuries	a 182-digit number of centuries	a 725-digit number of centuries

Zakłada się, że algorytmy są uruchamiane na komputerze zdolnym do wykonania miliarda instrukcji na sekundę (tj. jednej instrukcji na nanosekundę). Jak widać, nawet najmniejsza z pojawiających się w niej „złych” funkcji, 2^N , może wymagać 400 miliardów stuleci dla pojedynczego wystąpienia problemu 10 na 10! Dla funkcji takich jak N! lub N^N , czas jest niewyobrażalnie gorszy. Fakty te prowadzą do zasadniczej klasyfikacji funkcji na „dobre” i „złe”. Należy dokonać rozróżnienia między funkcjami wielomianowymi i superwielomianowymi. Dla naszych celów funkcja wielomianowa N to taka, która jest ograniczona od góry przez NK dla pewnego ustalonego K (co oznacza, że nie ma większej wartości niż N^K dla wszystkich wartości N od pewnego momentu). Wszystkie inne są superwielomianami. Na przykład funkcje logarytmiczne, liniowe i kwadratowe są wielomianowe, podczas gdy funkcje takie jak $1.001^N + N^6$, 5^N , N^N i N! są wykładnicze lub gorsze. Chociaż istnieją funkcje, takie jak na przykład $N\log_2N$, które są superwielomianowe, ale nie do końca wykładnicze, a inne, takie jak N^N , które są superwykładnicze, aktualna praktyka będzie polegać na lekkim nadużywaniu terminologii poprzez użycie „wykładniczego” jako synonim „super-wielomianu” w sequelu. Algorytm, którego działanie w czasie rzędu wielkości jest ograniczone od góry przez funkcję wielomianową N, gdzie N jest wielkością jego danych wejściowych, nazywa się algorytmem wielomianowym i będzie określany tutaj jako algorytm rozsądny. Podobnie algorytm, który w najgorszym przypadku wymaga czasu superwielomianowego lub wykładniczego, zostanie nazwany nieracjonalnym. Jeśli chodzi o problem algorytmiczny, problem, który dopuszcza rozwiązanie rozsądne lub wielomianowe, jest uważany za wykonalny, podczas gdy problem, który dopuszcza tylko rozwiązania nierozsądne lub w czasie wykładniczym, jest określany jako niewykonalny. Dyskusja i przykłady towarzyszące powyższym rysunkom mają na celu wsparcie tego rozróżnienia. Ogólnie rzecz biorąc, niewykonalne problemy wymagają niepraktycznie dużej ilości czasu, nawet przy stosunkowo niewielkich danych wejściowych, podczas gdy wykonalne problemy dopuszczają algorytmy, które są praktyczne dla danych wejściowych o rozsądnych rozmiarach. Możemy być usprawiedliwieni kwestionując mądrość wyznaczania granicy między dobrem a złem dokładnie tam, gdzie to zrobiliśmy. Jak już wspomniano, algorytm N1000 (który jest rozsądny z naszej definicji, ponieważ N^{1000} jest funkcją wielomianową) jest gorszy niż bardzo nieuzasadnione N! algorytm dla danych wejściowych o wielkości 1165, a punkt zwrotny jest znacznie większy, jeśli N^{1000} porównamy, powiedzmy, z wolniej rosnącą funkcją wykładniczą 1.001^N. Niemniej jednak większość nierozsądnych algorytmów jest naprawdę bezużyteczna, a większość rozsądnych jest wystarczająco przydatna, aby uzasadnić dokonane rozróżnienie. W rzeczywistości ogromna większość algorytmów wielomianowych dla praktycznych problemów ma wykładnik N, który jest nie większy niż 5 lub 6. Później zostaną przedstawione dowody na to, że wprowadzona tutaj dychotomia jest w rzeczywistości niezwykle solidna, a nawet bardziej tak niż ignorowanie stałych, które przychodzi przy

użyciu notacji Big-O. Sferę wszystkich problemów algorytmicznych można zatem podzielić na dwie główne klasy, jak pokazano na rysunku



Linia podziału zawiera jedną z najważniejszych klasyfikacji w teorii złożoności algorytmicznej.

Więcej o problemie z małpią łamigłówką

Problem małpich łamigłówek jest naprawdę gorszy niż wszystko, co do tej pory widzieliśmy. Prosi tylko o prostą odpowiedź tak/nie, ale nawet przy użyciu najbardziej znanych algorytmów moglibyśmy spędzić całe życie na jednej, bardzo małej instancji problemu i nigdy nie znaleźć prawidłowej odpowiedzi. Problem, dla którego nie ma znanego algorytmu wielomianowego, jest więc niewiele lepszy od problemu, dla którego w ogóle nie ma znanego algorytmu. Kluczowe pytanie brzmi, czy naprawdę nie ma rozsądnego rozwiązania; innymi słowy, czy problem z małpią łamigłówką jest naprawdę trudny do rozwiązania? Aby lepiej zrozumieć sytuację, skupmy się na kilku możliwych kłopotliwych kwestii.

1. Komputery z tygodnia na tydzień stają się szybsze. W ciągu ostatnich 10 lat szybkość komputera wzrosła mniej więcej 50-krotnie. Być może uzyskanie praktycznego rozwiązania problemu to tylko kwestia oczekiwania na dodatkową poprawę szybkości komputera.

2. Czy fakt, że nie znaleźliśmy lepszego algorytmu dla tego problemu, nie świadczy o naszej niekompetencji w konstruowaniu wydajnych algorytmów? Czy informatycy nie powinni pracować nad poprawą sytuacji, zamiast spędzać czas na pisaniu o tym książek?

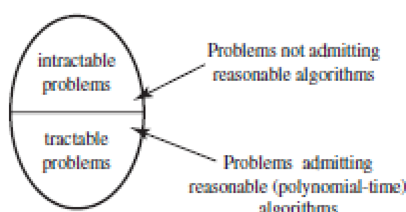
3. Czy ludzie nie próbowali szukać dolnego ograniczenia dla problemu w czasie wykładniczym, abyśmy mogli mieć dowód na to, że nie istnieje żaden rozsądny algorytm?

4. Może cały problem nie jest wart zachodu, ponieważ problem z małpią łamigłówką to tylko jeden konkretny problem. Może i jest kolorowa, ale na pewno nie wygląda na bardzo ważną.

Te kwestie są dobrze przyjęte, ale poniższa niezwykła sytuacja dostarcza odpowiedzi na wszystkie. Przede wszystkim pozbądźmy się zarzutu nr (1). Rysunek pokazuje, że nawet gdyby najszybszy komputer miał być zrobiony 1000 razy szybciej, algorytm 2^N dla problemu z małpią łamigłówką byłby w stanie, w danym przedziale czasowym (powiedzmy, godzinie), poradzić sobie tylko z około 10 kartami więcej niż może dzisiaj.

Function	Maximal number of cards solvable in one hour:		
	with today's computer	with computer 100 times faster	with computer 1000 times faster
N	A	$100 \times A$	$1000 \times A$
N^2	B	$10 \times B$	$31.6 \times B$
2^N	C	$C + 6.64$	$C + 9.97$

W przeciwieństwie do tego, gdyby algorytm zajął czas N, poradziłby sobie z 1000 razy większą liczbą kart niż obecnie. Stąd poprawa szybkości komputera o stały czynnik, nawet duży, poprawi sytuację, ale jeśli algorytm jest wykładniczy, to zrobi to tylko w bardzo nieznacznym stopniu. Odnieśmy się teraz do punktu (4) - pozostałe dwa punkty są traktowane w sposób dorozumiany później. Tak się składa, że problem małpich łamigłówek nie jest sam. Na tej samej łodzi są inne problemy. Ponadto łódź jest duża, efektowna i wieloboczna. Problem małpiej łamigłówki to tylko jeden z blisko 1000 różnych problemów algorytmicznych, z których wszystkie wykazują dokładnie te same zjawiska. Wszystkie dopuszczają nierozsądne rozwiązania w czasie wykładniczym, ale żadne z nich nie dopuszcza rozwiązań rozsądnych. Co więcej, nikt nie był w stanie udowodnić, że którykolwiek z nich wymaga czasu superwielomianowego. W rzeczywistości najbardziej znanymi dolnymi ograniczeniami większości problemów w tej klasie są O(N), co oznacza, że można sobie wyobrazić (choć mało prawdopodobne), że dopuszczają one bardzo wydajne algorytmy czasu liniowego. Oznaczymy tę klasę problemów NPC, co oznacza problemy NP-zupełne, jak wyjaśniono później. Luka algorytmiczna związana z problemami w NPC jest więc ogromna. Ich dolne granice są liniowe, a ich górne granice wykładnicze! Problem nie polega na tym, czy na ich rozwiązanie poświęcamy czas liniowy czy kwadratowy, czy też potrzebujemy 20 porównań do wyszukiwania, czy milion. Sprowadza się to do ostatecznego pytania, czy możemy rozwiązać te problemy za pomocą nawet rozsądnie małych nakładów na nawet największych i najpotężniejszych komputerach. To takie proste. Lokalizacja tych problemów algorytmicznych w sferze rysunku



jest więc nieznana, ponieważ ich górna i dolna granica leżą po obu stronach linii podziału. Istnieją dwie dodatkowe właściwości, które charakteryzują NPC specjalnej klasy i czynią go jeszcze bardziej niezwykłym. Jednak przed ich omówieniem należy podkreślić, że klasa NPC zawiera coraz większą różnorodność problemów algorytmicznych, pojawiających się w takich obszarach jak kombinatoryka, badania operacyjne, ekonomia, teoria grafów, teoria gier i logika. Warto przyjrzeć się niektórym innym problemom tam znalezionym.

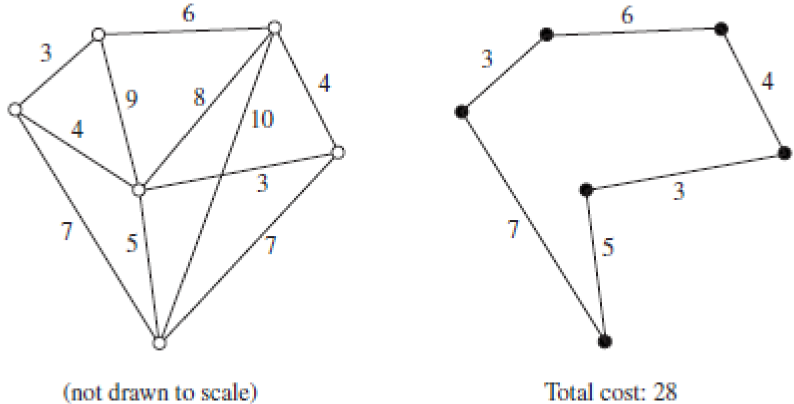
Problemy z układem dwuwymiarowym

Niektóre z najbardziej atrakcyjnych problemów w NPC to problemy z układaniem wywodzące się z dwuwymiarowych łamigłówek, takich jak łamigłówka z małpami. Innymi dobrymi przykładami są te łamigłówki, czasami rozdawane przez linie lotnicze, które zawierają szereg nieregularnych kształtów,

które mają być ułożone w prostokąt. Ogólny problem decyzyjny wymaga rozstrzygnięcia, czy N danych kształtów może być ułożonych w prostokąt i jest w NPC. Jedną z przyczyn oczywistego braku szybkiego rozwiązania jest istnienie wielu różnych rozwiązań częściowych, których nie można rozszerzyć na kompletne. Czasami układanka dopuszcza tylko jedno kompletne rozwiązanie. Rozważ zwykłe układanki. Nie dopuszczają praktycznie żadnych rozwiązań częściowych, z wyjątkiem części unikalnego rozwiązania końcowego. Dodanie elementu do częściowo rozwiązanej układanki polega zwykle na przejrzeniu wszystkich nieużywanych elementów i znalezieniu jednego, który będzie pasował. Wynika to albo z niepowtarzalnych kształtów przyrządów i wyszczerbików na poszczególnych kawałkach, albo z niejednorodnego charakteru tworzonego obrazu, albo z obu. Zatem, jak możemy zweryfikować, zwykłą, „grzeczną" układankę z N elementów można ułożyć w czasie kwadratowym. Jednak każdy, kto kiedykolwiek pracował nad układanką zawierającą dużo nieba, morza lub pustyni, wie, że nie wszystkie układanki są takie proste. (W takich łamigłówkach może się wydawać, że kilka części pasuje idealnie w danym miejscu, a pewne rozwiązanie częściowe będzie konieczne w celu uzyskania pełnego rozwiązania). te, które zawierają mniej niejednorodne obrazy i których elementy zawierają wiele identycznych przyrządów i wyszczerbików. Okazuje się, że ogólny problem z układanką N-częściową występuje również w klasie NPC. W gruncie rzeczy jest to po prostu problem z małpią układanką lub problem z nieregularnymi kształtami w przebraniu.

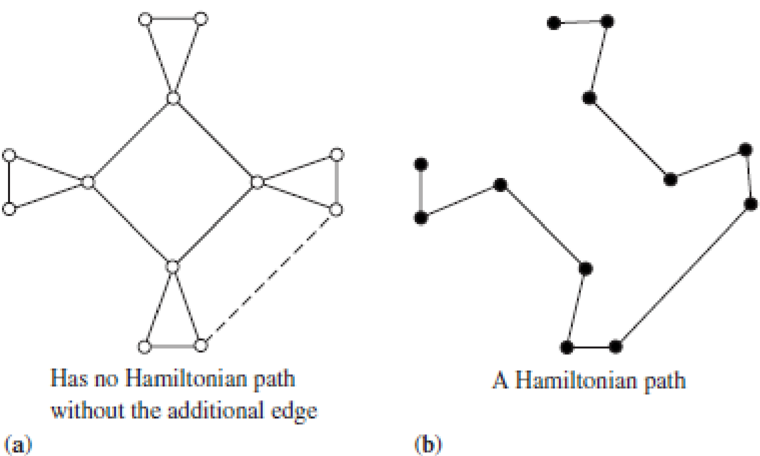
Problemy ze znajdowaniem ścieżki

W Części 4 opisano dwa problemy, oba związane ze znalezieniem pewnych struktur o minimalnych kosztach w sieciach miejskich. Obejmowały one odpowiednio leniwych wykonawców kolei (znajdowanie drzew o minimalnej rozpiętości) i zmęczonych podróżnych (znajdowanie najkrótszych tras). Sieć miejska to graf składający się z N punktów (miast) i krawędzi z powiązanymi kosztami (są to odległości między miastami). Zachowanie czasowe obu algorytmów przedstawionych w Części 4 jest kwadratowe, a zatem oba problemy są wykonalne. Oto kolejny problem, który na pierwszy rzut oka wygląda bardzo podobnie do dwóch pozostałych. Chodzi o komiwojażera, który musi odwiedzić każde z miast w danej sieci, zanim wróci do punktu startowego, z którego podróż dobiegnie końca. Problem algorytmiczny wymaga podania najtańszej trasy, a mianowicie trasy zamkniętej, która przechodzi przez każdy z węzłów na grafie i której całkowity koszt (czyli suma odległości oznaczających krawędzie) jest minimalny. Rysunek przedstawia sieć sześciu miast, w której pokazano taką optymalną trasę.

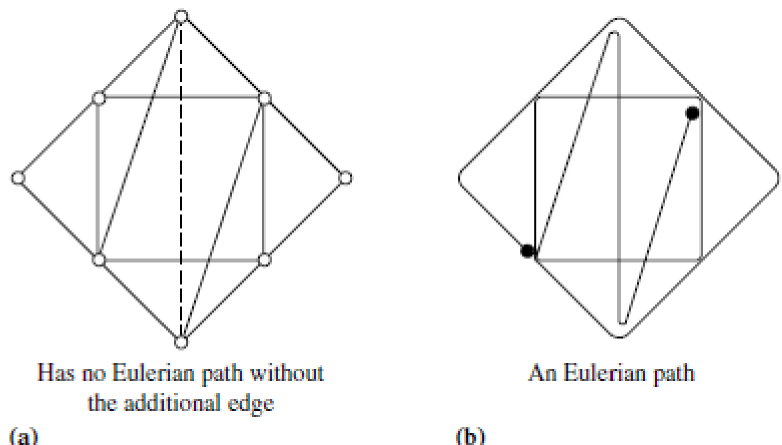


W wersji tak/nie dane wejściowe zawierają liczbę K poza wykresem miasta G, a problem decyzyjny pyta po prostu, czy istnieje wycieczka po G o całkowitym koszcie nie większym niż K. Tak więc dla wykresu z rysunku, odpowiedź brzmiałaby „tak", jeśli K wynosi 30, ale „nie", jeśli K wynosi 27. Pomimo nieco żartobliwego opisu, problem komiwojażera, podobnie jak problemy z minimalnym drzewem opinającym i najkrótszą drogą, nie jest przykładem zabawkowym. Jego warianty pojawiają się w

projektowaniu sieci telefonicznych i układów scalonych, w planowaniu linii konstrukcyjnych, w programowaniu robotów przemysłowych, by wymienić tylko kilka zastosowań. We wszystkich tych przypadkach możliwość znalezienia niedrogich wycieczek po danych wykresach może być bardzo istotna. Ponownie, łatwo jest znaleźć naiwne rozwiązanie w czasie wykładniczym. Po prostu rozważ wszystkie możliwe wycieczki, zatrzymując się i mówiąc „tak", jeśli koszt danej wycieczki nie przekracza K, i „nie", jeśli wszystkie wycieczki zostały uwzględnione, ale stwierdzono, że wszystkie kosztują więcej niż K. algorytm O(N!). (Dlaczego?) Znowu, nawet przypadek, w którym N wynosi 25 jest po prostu beznadziejny i musimy zdać sobie sprawę, że o ile dla komiwojażera z walizką pełną drobiazgów 25 miast może brzmieć dużo, to jest to prawie śmieszna liczba dla rodzaj rzeczywistych aplikacji wspomnianych powyżej. Faktem jest, że problem komiwojażera jest również w NPC, co sprawia, że problem jest w praktyce nierozwiązywalny, o ile wiemy. Warto pokrótce rozważyć kilka innych problemów ze znajdowaniem ścieżki. Zobaczmy, co się stanie, gdy całkowicie pominiemy długości krawędzi. Mając graf składający się z punktów i krawędzi, możemy po prostu zapytać, czy istnieje jakakolwiek ścieżka, która przechodzi przez wszystkie punkty dokładnie raz. Takie ścieżki nazywamy hamiltonianem. Rysunek (a) pokazuje wykres, który nie ma ścieżki Hamiltona, a Rysunek (b) pokazuje, jak dodanie pojedynczej krawędzi może zmienić sytuację.



Choć pozornie o wiele łatwiej, ten problem dotyczy również NPC. Istnieje prosty algorytm czasu wykładniczego, który sprawdza wszystkie N! ścieżki, szukając takiej, która dociera do każdego punktu raz, ale nikt nie zna rozwiązania wielomianowego. Co ciekawe, jeśli szukamy ścieżki, która ma przejść przez wszystkie krawędzie dokładnie raz, a nie przez wszystkie punkty, historia jest zupełnie inna. Takie ścieżki są określane jako Eulerowskie, a rysunek jest odpowiednikiem Eulera z rysunku powyżej.



Na pierwszy rzut oka wydaje się, że nie ma lepszego sposobu na znalezienie ścieżki Eulera niż podążanie każdą możliwą ścieżką. (W najgorszym przypadku jest ich około (N2)!. Dlaczego?) Jednak proste, ale raczej sprytne rozwiązanie problemu ścieżki Eulera w czasie wielomianowym zostało odnalezione w 1736 roku przez wielkiego szwajcarskiego matematyka Leonharda Eulera. Rozwiązanie uzyskuje się pokazując, że graf zawiera ścieżkę Eulera dokładnie wtedy, gdy spełnia następujące dwie właściwości: (1) jest połączony (to znaczy, że dowolny punkt jest osiągalny z dowolnego innego) oraz (2) liczba krawędzi emanujących z dowolnego punktu (może z wyjątkiem dwóch punktów) jest parzysty. W związku z tym algorytm musi tylko sprawdzić, czy wykres wejściowy spełnia te właściwości. Sprawdzenie drugiego jest trywialne, a pierwszy można łatwo wykazać, że przyjmuje szybki algorytm, taki, który jest w rzeczywistości liniowy pod względem liczby krawędzi.

Problemy z planowaniem i dopasowywaniem

Wiele problemów z NPC dotyczy w taki czy inny sposób harmonogramowania lub dopasowywania. Załóżmy na przykład, że podano nam konkretne godziny, w których każdy z N nauczycieli jest dostępny, oraz konkretne godziny, w których można zaplanować każdą z M klas. Dodatkowo podawana jest nam ilość godzin, jaką każdy z nauczycieli ma przerobić na poszczególnych zajęciach. Problem z rozkładem jazdy pyta, czy możliwe jest takie dopasowanie nauczycieli, klas i godzin, aby wszystkie założone ograniczenia były spełnione, aby nie było dwóch nauczycieli w tej samej klasie w tym samym czasie i aby nie było dwóch klas w tym samym nauczyciel w tym samym czasie. Problem z rozkładem jazdy należy również do specjalnej klasy NPC. Inne problemy z dopasowaniem w NPC obejmują dopasowywanie ładunków różnej wielkości do ciężarówek o różnej pojemności (czasami nazywane problemem pakowania w bin-packing) lub przydzielanie uczniów do akademików w sposób, który spełnia pewne ograniczenia. Należy podkreślić, że wiele innych problemów związanych z planowaniem i dopasowywaniem jest całkiem wykonalnych, tak jak w przypadku aranżacji, problemów ze znajdowaniem ścieżek i tak dalej.

Ustalenie logicznej prawdy

Jednym z najbardziej znanych problemów w NPC jest ustalenie prawdziwości lub fałszu zdań w prostym formalizmie logicznym zwanym rachunkiem zdań. W tym języku można łączyć symbole oznaczające twierdzenia elementarne w twierdzenia bardziej złożone, używając spójników logicznych & (oznaczające „i”), ∨ (oznaczające „lub”), ~ („nie”) oraz →(„implikuje ”). Na przykład zdanie:

~ (E →F) & (F ∨ (D → ~ E))

stwierdza, że (1) nie jest tak, że prawda E implikuje prawdziwość F i (2) albo F jest prawdziwe, albo prawda D implikuje fałszywość E. Problem algorytmiczny wymaga określenia spełnialności takich zdań. Innymi słowy, mając dane zdanie jako dane wejściowe, chcemy wiedzieć, czy podstawowym stwierdzeniom w nim występującym można przypisać „prawdę" lub „fałsz", aby całe zdanie okazało się prawdziwe. Przypisanie wartości logicznych do podstawowych stwierdzeń nazywa się przypisaniem prawdy. W tym przykładzie możemy ustalić, że E jest prawdziwe, a D i F fałszywe. Spowoduje to, że (1) będzie prawdziwe, ponieważ E jest prawdziwe, a F nie, tak że E nie może implikować F. Ponadto (2) jest prawdziwe, pomimo fałszywości F, ze względu na fałszywość D3. zadowalający. Możesz sprawdzić, czy podobne zdanie:

~ ((D & E) →F) & (F ∨ (D → ~E))

jest niezadowalająca; nie ma sposobu na przypisanie wartości prawdy do D, E i F, które sprawią, że to zdanie będzie prawdziwe. Nie jest zbyt trudne wymyślenie algorytmu wykładniczego dla problemu spełnialności, gdzie N jest liczbą odrębnych elementarnych stwierdzeń w zdaniu wejściowym. Po prostu wypróbowujemy wszystkie możliwe przypisania prawdziwości. Jest ich dokładnie 2N (dlaczego?) i łatwo jest sprawdzić prawdziwość wzoru względem każdego z tych przypisań prawdziwości w czasie, który jest wielomianem w N. W konsekwencji cały algorytm działa w czasie wykładniczym. Niestety, problem spełnialności dla rachunku zdań występuje również w NPC, więc naiwny algorytm czasu wykładniczego jest w istocie najbardziej znany. O ile obecnie wiadomo, nie da się algorytmicznie sprawdzić, czy nawet dość krótkie zdania mogą być prawdziwe.

Kolorowanie map i wykresów

W Części 5 opisaliśmy twierdzenie o czterech kolorach, które dowodzi, że każdą mapę krajów można pokolorować tylko czterema kolorami w taki sposób, że żadne dwa sąsiadujące kraje nie są pokolorowane tak samo. Wynika z tego, że algorytmiczny problem określenia, czy dana mapa może być czterokolorowa, jak określa się ten rodzaj procesu, jest trywialny: na dowolnej mapie wejściowej odpowiedź brzmi po prostu „tak". Ten sam problem, ale tam, gdzie dozwolone są tylko dwa kolory, też nie jest trudny. Mapa może być dwukolorowa dokładnie wtedy, gdy nie zawiera punktu będącego skrzyżowaniem nieparzystej liczby krajów (dlaczego?), a ta właściwość jest łatwa do sprawdzenia. Zauważ, że liczba kolorów nie jest tutaj brana jako część danych wejściowych. Omawiamy problemy algorytmiczne uzyskane przez ustalenie dozwolonej liczby kolorów i pytanie, czy wejściową mapę kraju można pokolorować za pomocą tej stałej liczby. Jak wykazaliśmy, przypadki dwóch i czterech kolorów są trywialne. Ciekawym przypadkiem są trzy kolory. Określanie, czy mapa może być trójkolorowa, odbywa się w NPC, co oznacza, że znamy tylko nieracjonalne rozwiązania, przez co problem jest w praktyce nierozwiązywalny, z wyjątkiem bardzo małej liczby krajów. Powiązany problem polega na kolorowaniu wykresów. Zasada kolorowania jest tutaj podobna do kolorowania mapy, z tą różnicą, że węzły (punkty na wykresie) pełnią rolę krajów. Żadne dwa sąsiednie węzły (czyli węzły połączone krawędzią) nie mogą być monochromatyczne. W przeciwieństwie do map krajów, których płaskość ogranicza możliwe konfiguracje krajów granicząych, każdy węzeł grafu może być połączony krawędziami z dowolnym innym. Tworzenie wykresów, które wymagają dużej liczby kolorów, jest łatwe. Wykres zawierający K węzłów, z których każdy jest połączony ze wszystkimi innymi, wymaga oczywiście K kolorów. Taki wykres nazywa się kliką. Problem algorytmiczny wymaga podania minimalnej liczby kolorów wymaganych do pokolorowania danego wykresu. Wersja tak/nie, która pyta, czy wykres można pokolorować za pomocą K kolorów, gdzie K jest częścią danych wejściowych, jest również w NPC, a zatem nie wiadomo, czy ma rozsądne rozwiązanie. Ponieważ oryginalna wersja jest co najmniej tak trudna, jak wersja tak/nie (dlaczego?), nie wiadomo, czy można ją rozwiązać w rozsądnym czasie.

Krótkie Certyfikaty i Magiczne Monety

Wszystkie te problemy NPC zdają się wymagać, jako część ich wrodzonej natury, abyśmy wypróbowali częściowe dopasowania, częściowe przypisania prawdy, częściowe aranżacje lub częściowe kolory i stale je rozszerzać w nadziei na osiągnięcie ostatecznego, pełnego rozwiązania. Kiedy częściowe rozwiązanie nie może zostać rozszerzone, najwyraźniej musimy się cofnąć; to znaczy cofnąć rzeczy już zrobione, przygotowując się do wypróbowania alternatywy. Jeśli ten proces jest przeprowadzany ostrożnie, żadne możliwe rozwiązanie nie zostanie pominięte, ale w najgorszym przypadku wymagany jest wykładniczy czas. W związku z tym, mając dane wejściowe do problemu NPC, niezwykle trudno jest stwierdzić, czy odpowiedź na pytanie zawarte w problemie brzmi „tak" czy „nie". Interesujące jest jednak to, że we wszystkich tych problemach, jeśli odpowiedź brzmi „tak", istnieje łatwy sposób, aby kogoś o tym przekonać. Istnieje tak zwany certyfikat, który zawiera rozstrzygający dowód na to, że odpowiedź rzeczywiście brzmi „tak". Co więcej, certyfikat ten może być zawsze skrócony. Jego rozmiar zawsze może być ograniczony wielomianem w N. (W rzeczywistości najczęściej jest on liniowy w N). Na przykład, jak omówiono wcześniej, wydaje się, że notorycznie trudno jest stwierdzić, czy graf zawiera ścieżkę Hamiltona, czy też zawiera taką, której długość nie jest większa niż pewna podana liczba K. Z drugiej strony, jeśli taka ścieżka rzeczywiście istnieje, można ją pokazać i łatwo sprawdzić, czy jest pożądanego rodzaju, służąc w ten sposób jako doskonały dowód, że odpowiedź brzmi „tak". Podobnie, chociaż trudno jest znaleźć przypisanie prawdziwości, które spełnia zdanie w rachunku zdań, łatwo jest poświadczyć jego spełnialność, po prostu wykazując przypisanie, które je spełnia. Co więcej, łatwo jest zweryfikować w czasie wielomianowym, czy to konkretne przypisanie prawdziwości spełnia swoje zadanie. W podobnym duchu pokazanie prawnego ułożenia kart z małpią łamigłówką służy jako rozstrzygający dowód, że odpowiedni problem algorytmiczny mówi „tak", gdy zostanie zastosowany do łamigłówki jako danych wejściowych. Również tutaj legalność układu (dopasowanie kolorów i połówek) można łatwo zweryfikować w czasie wielomianowym. Ustalenie, czy problem NPC mówi „tak" na dane wejściowe, jest zatem trudne, ale stwierdzenie, że rzeczywiście tak jest, gdy tak jest, jest łatwe. Jest inny sposób opisania tego zjawiska. Załóżmy, że mamy bardzo specjalną magiczną monetę, której można użyć w opisanej właśnie procedurze cofania się. Gdy tylko możliwe jest rozszerzenie częściowego rozwiązania na dwa sposoby (na przykład dwie karty z małpami mogą być legalnie umieszczone w aktualnie pustym miejscu lub następny symbol asercji może być przypisany „prawda" lub „fałsz"), moneta jest odwracana i wybór jest dokonany zgodnie z wynikiem. Moneta nie wypada jednak losowo; posiada magiczny wgląd, zawsze wskazujący najlepszą możliwość. Moneta zawsze wybierze możliwość, która prowadzi do kompletnego rozwiązania, jeśli istnieje kompletne rozwiązanie. (Jeśli obie możliwości prowadzą do kompletnych rozwiązań, a jeśli żadne nie, moneta zachowuje się jak zwykła losowa moneta.) Technicznie mówimy, że algorytmy używające takich magicznych monet są niedeterministyczne, ponieważ zawsze „zgadują", która z dostępnych opcji jest lepsze, niż konieczność zastosowania jakiejś deterministycznej procedury, aby przejść przez wszystkie. Jakoś zawsze udaje im się dokonać właściwego wyboru. Oczywiście, gdyby pozwolić algorytmom na wykorzystanie takiego magicznego niedeterminizmu, moglibyśmy ulepszyć rozwiązania pewnych problemów algorytmicznych, ponieważ unika się pracy związanej z wypróbowywaniem możliwości. W przypadku problemów NPC poprawa ta nie jest wcale marginalna: każdy problem NPC ma niedeterministyczny algorytm wielomianowy. Fakt ten można udowodnić, pokazując, że omówione powyżej „krótkie" certyfikaty odpowiadają bezpośrednio wielomianowym „magicznym" egzekucjom; wszystko, co musimy zrobić, to postępować zgodnie z instrukcjami magicznej monety, a gdy mamy gotowe rozwiązanie kandydata, po prostu sprawdzić, czy jest to legalne. Ponieważ moneta zawsze wskazuje najlepszą możliwość, możemy śmiało powiedzieć „nie", jeśli proponowane rozwiązanie narusza zasady. Moneta znalazłaby rozwiązanie prawne, gdyby takie istniało. Tak więc problemy NPC są pozornie nierozwiązywalne, ale stają się „wykonalne" przez użycie magicznego niedeterminizmu. To

wyjaśnia część akronimu NPC: N i P oznaczają niedeterministyczny czas wielomianowy, więc mówi się, że problem występuje w NP, jeśli dopuszcza krótki certyfikat. Przejdźmy teraz do C, co oznacza „CompletOprócz dopuszczania deterministycznych rozwiązań, które wymagają nierozsądnego czasu i „magicznych", niedeterministycznych, które wymagają rozsądnego czasu, problemy z NPC mają dodatkową, najbardziej niezwykłą właściwość. Los każdego jest ściśle związany z losem wszystkich pozostałych. Albo wszystkie problemy z NPC można rozwiązać, albo żaden z nich nie! Termin „kompletny" jest używany do oznaczenia tej dodatkowej właściwości, tak że, jak wspomniano, problemy w NPC są znane jako problemy NP-zupełne. Wyostrzmy to stwierdzenie. Jeśli ktoś miałby znaleźć algorytm wielomianowy dla dowolnego pojedynczego problemu NP-zupełnego, natychmiast powstałyby algorytmy wielomianowe dla nich wszystkich. To implikuje podwójny fakt: jeśli ktoś miałby udowodnić dolne ograniczenie w czasie wykładniczym dla dowolnego problemu NP-zupełnego, ustalając, że nie można go rozwiązać w czasie wielomianowym, natychmiast wynikałoby z tego, że żaden taki problem nie może być rozwiązany w czasie wielomianowym. To jest szczyt solidarności i nie jest to przypuszczenie - zostało to udowodnione: wszystkie problemy NP-zupełne stoją lub upadają razem. Po prostu nie wiemy, który to jest. Parafrazując dzielnego starego księcia Yorku, moglibyśmy powiedzieć:

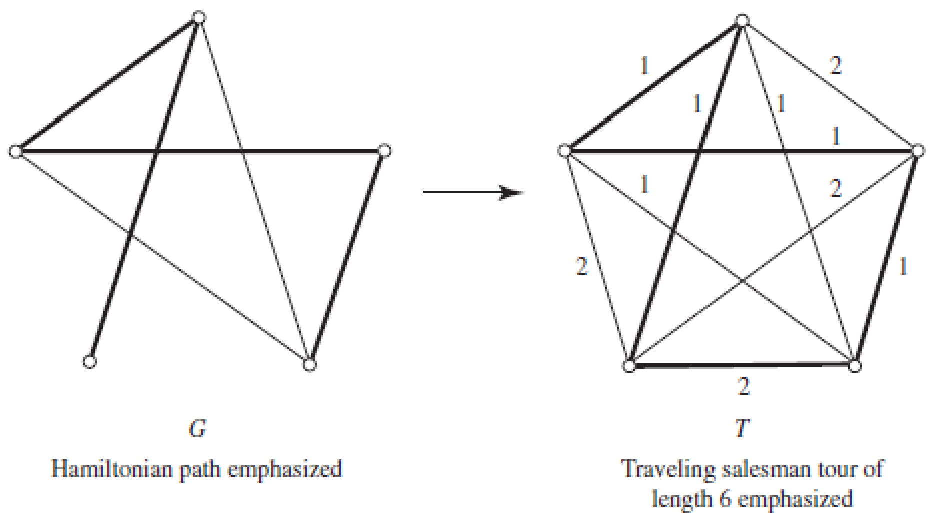
A kiedy wstają, wstają

A kiedy są na dole, są na dole

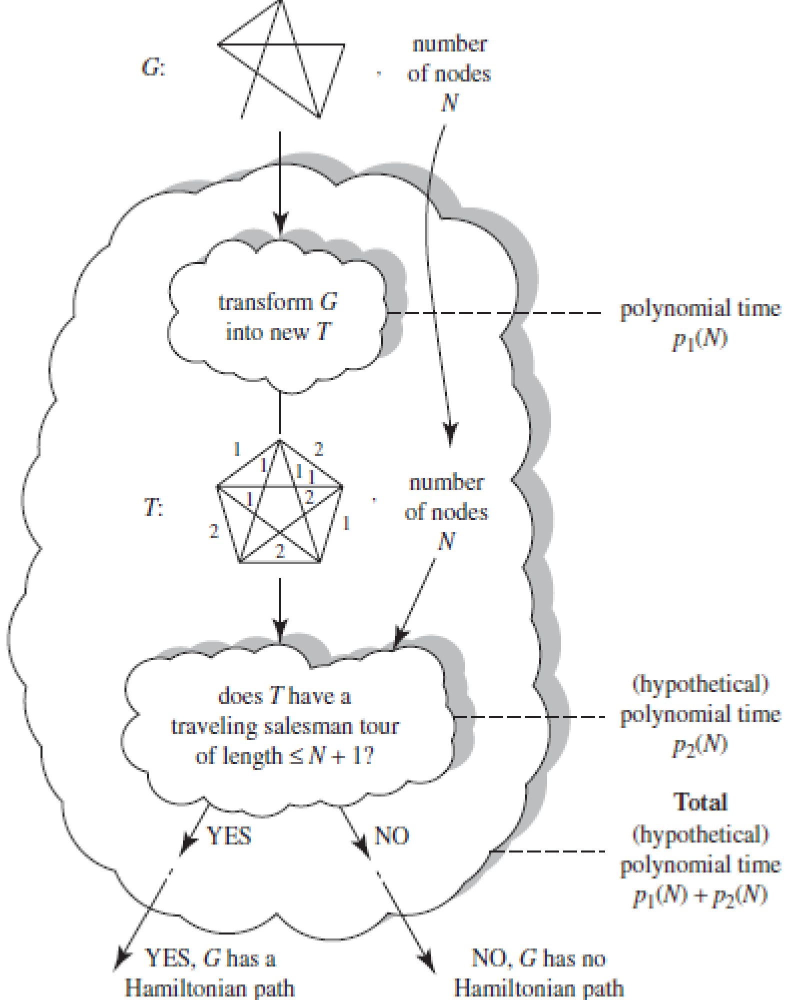
A ponieważ nie mogą być w połowie drogi

Są albo w górę, albo w dół

Jak możemy udowodnić tak szerokie twierdzenie? Przypomnijmy, że blisko 1000 różnych problemów w różnych obszarach jest znanych jako NP-zupełne! Koncepcja, która jest używana do ustalenia tego zjawiska typu stand-or-fall-together, to redukcja wielomianu w czasie. Biorąc pod uwagę dwa problemy NP-zupełne, redukcja wielomianowa jest algorytmem, który działa w czasie wielomianowym i redukuje jeden problem do drugiego, w następującym sensie. Jeśli ktoś przychodzi z wejściem X do pierwszego problemu i chce odpowiedzi „tak" lub „nie", używamy algorytmu do przekształcenia X w dane wejściowe Y do drugiego problemu w taki sposób, że odpowiedź drugiego problemu na Y jest właśnie odpowiedzią pierwszego problemu na X. Na przykład, dość łatwo jest zredukować problem ścieżki Hamiltona do problemu komiwojażera. Mając graf G z N węzłami, skonstruuj sieć komiwojażera T w następujący sposób. Węzły T są dokładnie węzłami G, ale krawędzie są rysowane między każdymi dwoma węzłami, przypisując koszt 1 krawędzi, jeśli była obecna w oryginalnym grafie G, i 2, jeśli jej nie było. Rysunek ilustruje transformację.



Nietrudno zauważyć, że T ma podróż komiwojażera o długości N + 1 lub mniej (przechodząc raz przez każdy punkt), dokładnie jeśli G zawiera ścieżkę Hamiltona. Zatem, aby odpowiedzieć na pytania o istnienie ścieżek hamiltonowskich, weźmy graf wejściowy G i przeprowadźmy transformację do T . Następnie zapytaj, czy T ma wycieczkę komiwojażera, która nie jest dłuższa niż N + 1, gdzie N jest liczbą węzłów w G. Odpowiedź na pierwsze pytanie na G brzmi „tak" dokładnie wtedy, gdy odpowiedź na drugie pytanie na T jest tak." Zauważ też, że transformacja zajmuje tylko wielomianową ilość czasu. Dlaczego ten fakt jest interesujący? Ponieważ pokazuje, że pod względem wykonalności problem ścieżki Hamiltona nie jest gorszy niż problem komiwojażera; jeśli ta ostatnia ma rozsądne rozwiązanie, to czyni ją również ta pierwsza. Rysunek ilustruje sposób wykorzystania redukcji do uzyskania rozsądnego algorytmu ścieżki hamiltonowskiej z (hipotetycznego) rozsądnego algorytmu komiwojażera.



Teraz pojawia się fakt, który ustala wspólne zjawisko losu problemów NP-zupełnych: każdy problem NP-zupełny jest wielomianowo redukowalny do każdego innego! W konsekwencji podatność jednego implikuje podatność wszystkich, a podatność jednego implikuje podatność wszystkich. Interesujące jest to, że aby ustalić nowo rozpatrywany problem R jako NP-zupełny, nie musimy konstruować wielomianowych redukcji między R a wszystkimi innymi problemami NP-zupełnymi. W rzeczywistości wystarczy zredukować wielomianowo R do pojedynczego problemu, o którym już wiadomo, że jest NP-zupełny, nazwać go Q, i zredukować inny taki problem (prawdopodobnie ten sam), nazwać go S, do R. Pierwszy z nich redukcja pokazuje, że pod względem wykonalności R nie może być gorsza niż Q, a druga pokazuje, że nie może być lepsza niż S. A zatem, jeśli Q jest wykonalne, to także R, a jeśli R jest wykonalne, to także S. Ale ponieważ Q i S są oba NP-zupełne, stoją i upadają razem, stąd R również stoi i upada wraz z nimi, co implikuje własną NP-zupełność. Wynika z tego, że znając jeden problem NP-zupełny, możemy pokazać, że inne są również NP-zupełne, stosując mechanizm redukcji dwukrotnie dla każdego nowego problemu kandydującego. Właściwie w praktyce przeprowadza się tylko drugą z tych redukcji. Aby ustalić, że R nie może być gorsze od problemów NP-zupełnych, czyli jest w NP, zwykle łatwiej jest przedstawić krótki certyfikat lub magiczny niedeterministyczny algorytm czasu wielomianowego niż jawnie zredukować R do jakiegoś znanego Problem NP-zupełny. Ponadto, aby pokazać, że problem R jest zupełny w NP, niekoniecznie potrzebujemy problemu, który wcześniej okazał się NP-zupełny. Zamiast tego możemy użyć ogólnego argumentu, aby pokazać, że każdy problem w NP można sprowadzić wielomianowo do R. Teraz wyraźnie wszystko to musiało się gdzieś zacząć; musiał istnieć pierwszy problem, który okazał się NP-zupełny. Rzeczywiście, w 1971 r.

wykazano, że problem spełnialności dla rachunku zdań jest NP-zupełny, stanowiąc w ten sposób kotwicę dla dowodów NPzupełności. Wynik, znany jako twierdzenie Cooka, jest uważany za jeden z najważniejszych wyników w teorii złożoności algorytmicznej.

Redukcja pomarańczy do jabłek

Redukcje w czasie wielomianowym między problemami NP-zupełnymi mogą być znacznie subtelniejsze niż właśnie podane. Chociaż wcale nie jest jasne, co mają ze sobą wspólnego rozkłady jazdy i małpie łamigłówki, wiemy, że musi być między nimi redukcja, ponieważ oba są NP-zupełne. Ponieważ nowy problem wymaga tylko jednej redukcji w każdym kierunku, często najlepiej znana redukcja między dwoma problemami składa się z łańcucha redukcji prowadzącego przez kilka innych problemów NP-zupełnych. Byłoby bezowocne męczyć cię jednym z naprawdę trudnych przypadków, więc oto przykład redukcji, która nie jest zbyt trudna, ale też nie całkiem trywialna. Pokazujemy, jak zredukować problem trójkolorowania mapy do spełnialności w rachunku zdań. To dowodzi, że pierwszy nie może być gorszy pod względem wykonalności niż drugi. W szczególności musimy opisać algorytm, który wprowadza opis jakiejś arbitralnej mapy M i wyprowadza zdanie zdaniowe F, takie, że M może być trójkolorowe wtedy i tylko wtedy, gdy F jest spełnialne. Ponadto algorytm musi działać w czasie wielomianowym, co oznacza między innymi, że liczba symboli we wzorze F może być co najwyżej wielomianowo większa niż liczba krajów w M. Niech M będzie daną mapą, zakładając zaangażowanie krajów $C_1 \dots C_N$. Opiszemy zdanie F i argumentujemy, że jego rozmiar jest wielomianem w N. Powinieneś być w stanie dość łatwo zobaczyć, jak można to przekształcić w ogólny algorytm wielomianowy, który działa dla każdego M.

Zakładając, że trzy kolory to R, B i Y (czerwony, niebieski i żółty), zdanie F zawiera 3N elementarne twierdzenia, po jednym dla każdej kombinacji koloru i kraju. Na przykład stwierdzenie „C_I-is-Y ” ma oznaczać, że kraj C_I ma kolor żółty. Konstruujemy F przez „i”-składając razem dwie części. Pierwsza część twierdzi, że każdy kraj jest pokolorowany dokładnie jednym z trzech kolorów, ni mniej, ni więcej. Składa się z „i” razem następujących zdań dla każdego kraju C_I

$$
\begin{aligned}
&((C_I\text{-}is\text{-}R \ \& \ {\sim}C_I\text{-}is\text{-}B \ \& \ {\sim}C_I\text{-}is\text{-}Y) \\
&\vee (C_I\text{-}is\text{-}B \ \& \ {\sim}C_I\text{-}is\text{-}R \ \& \ {\sim}C_I\text{-}is\text{-}Y) \\
&\vee (C_I\text{-}is\text{-}Y \ \& \ {\sim}C_I\text{-}is\text{-}B \ \& \ {\sim}C_I\text{-}is\text{-}R))
\end{aligned}
$$

co oznacza, że C_I jest koloru czerwonego, a nie niebieskiego ani żółtego, lub niebieskiego, a nie czerwonego ani żółtego, lub żółtego, a nie niebieskiego ani czerwonego. Drugą część uzyskuje się identyfikując wszystkie pary krajów C_I i C_J które sąsiadują na mapie M, i łącząc ze sobą następujące zdania dla każdej takiej pary, stwierdzając, że oba kraje nie są pokolorowane tym samym kolorem:

$$
\begin{aligned}
&{\sim}((C_I\text{-}is\text{-}R \ \& \ C_J\text{-}is\text{-}R) \\
&\quad \vee (C_I\text{-}is\text{-}B \ \& \ C_J\text{-}is\text{-}B) \\
&\quad \vee (C_I\text{-}is\text{-}Y \ \& \ C_J\text{-}is\text{-}Y))
\end{aligned}
$$

co oznacza, że nie jest tak, że zarówno C_I jak i C_J są w kolorze czerwonym lub oba w kolorze niebieskim lub oba w kolorze żółtym. Jak długie jest zdanie F? Pierwsza część jest liniowa w N, ponieważ zawiera jedno zdanie o stałej długości dla każdego kraju C_I . Drugi jest nie gorszy niż kwadratowy w N, ponieważ zawiera jedno podzdanie o stałej długości dla każdej pary sąsiednich krajów I i J , a par może być nie więcej niż N^2. Stąd wyraźnie F jest wielomianem w N. Pozostaje wykazać, że F jest spełnialne wtedy i tylko wtedy, gdy M jest trójkolorowe. Aby to udowodnić, postępujemy w obu kierunkach. Jeśli M jest trójkolorowe według jakiegoś schematu kolorowania S (który, jak można założyć, obejmuje kolory

czerwony, niebieski i żółty), możemy spełnić F po prostu przypisując „prawda" elementarnemu stwierdzeniu C_I-is-X, jeśli schemat S wymaga pokolorowania kraju C_I kolorem X, w przeciwnym razie „fałsz". Łatwo zauważyć, że wszystkie części F są w ten sposób spełnione. I odwrotnie, jeśli F jest spełnione przez jakieś przypisanie prawdy S, to M można pokolorować trzema kolorami, przypisując kolor X krajowi C_I dokładnie wtedy, gdy przypisanie S przypisuje „prawda" twierdzeniu C_I -is-X. Konstrukcja F gwarantuje, że każdy kraj jest pokolorowany dokładnie jednym kolorem i nie ma konfliktów. Na tym kończy się redukcja.

Czy P jest równe NP?

Tak jak NP oznacza klasę problemów, które dopuszczają niedeterministyczne algorytmy wielomianowe, tak P oznacza to, co nazywaliśmy problemami wykonalnymi; mianowicie te, które dopuszczają algorytmy wielomianowe. Duża klasa problemów, które obszernie omawialiśmy, problemy NP-zupełne, są „najtrudniejszymi" problemami w NP w tym sensie, że istnieją wielomianowe redukcje z każdego problemu w NP do każdego z nich. Jeśli któryś z nich okaże się łatwy, to znaczy w P, to wszystkie problemy w NP również są w P. Skoro oczywiście P jest częścią NP (dlaczego?), pytanie tak naprawdę sprowadza się do tego, czy P jest równe do NP czy nie. P = NP? problem, jak się go nazywa, jest otwarty od momentu postawienia go w 1971 roku i jest jednym z najtrudniejszych nierozwiązanych problemów w informatyce. To zdecydowanie najbardziej intrygujące. Albo wszystkie te interesujące i ważne problemy mogą być rozsądnie rozwiązane przez komputer, albo żaden z nich nie jest w stanie tego zrobić. Wielu najbardziej utalentowanych informatyków teoretycznych pracowało nad tym problemem, ale bezskutecznie. Większość z nich uważa, że P = NP, co oznacza, że problemy NP-zupełne są z natury nierozwiązywalne, ale nikt nie wie tego na pewno. W każdym razie wykazanie, że problem algorytmiczny jest NP-zupełny, jest uważane za ważny dowód jego prawdopodobnej nierozwiązywalności. Niektóre problemy, co do których wykazano, że występują w NP, nie są ani NPkompletne, ani w P. Przez wiele lat najbardziej znanym tego przykładem był problem testowania liczby pod kątem pierwszości; to znaczy, pytając, czy ma jakieś czynniki (liczby, które go dokładnie dzielą) inne niż 1 i siebie. Jeśli problem jest sformułowany w formie, która pyta, czy liczba nie jest liczbą pierwszą, istnieje oczywisty krótki certyfikat na wypadek, gdyby odpowiedź brzmi „tak" (to znaczy, że liczba nie jest liczbą pierwszą). Certyfikat to po prostu czynnik, który nie jest ani 1, ani samą liczbą. Sprawdzamy, czy rzeczywiście jest to czynnik, za pomocą prostego podziału. A zatem łatwo zauważyć, że problem niepierwotności dotyczy NP. Z drugiej strony, jeśli problem dotyczy tego, czy liczba jest liczbą pierwszą, wcale nie jest oczywiste, że istnieje krótki certyfikat. Niemniej jednak prawie 30 lat temu wykazano, że problem pierwszości występuje również w NP. Mimo to, podobnie jak w przypadku wszystkich problemów, które są w NP, ale nie wiadomo, że występują w P, zawsze istniała dokuczliwa możliwość, że problem pierwszości okaże się nierozwiązywalny. Wielką niespodzianką jest to, że pierwszorzędność jest w rzeczywistości w P. Tuż przed ukończeniem tej edycji książki odkryto niezwykły algorytm wielomianowy dla pierwszości (nazywany algorytmem AKS, od inicjałów jego autorów), dzięki czemu reszta jednego z najciekawszych otwartych problemów algorytmiki.

Niedoskonałe rozwiązania problemów NP-zupełnych

Wiele problemów decyzyjnych NP-zupełnych to wersje tak/nie tego, co czasami nazywa się problemami optymalizacji kombinatorycznej. Dobrym przykładem jest problem komiwojażera. Oczywiście problem ze znalezieniem optymalnej trasy nie może być wykonalny, jeśli wersja tak/nie jest również wykonalna, ponieważ gdy już znajdziemy optymalną trasę, możemy łatwo sprawdzić, czy jej całkowita długość nie jest większa niż podana liczba K. W tym celu Z tego powodu mówi się, że pierwotny problem jest NP-zupełny, a zatem, jeśli chodzi o obecną wiedzę, jest nierozwiązywalny. Jednak w niektórych przypadkach możemy rozwiązać problemy optymalizacyjne w sposób, który nie jest doskonały, ale ma znaczną wartość praktyczną. Algorytmy zaprojektowane do tego celu są ogólnie

nazywane algorytmami aproksymacyjnymi i opierają się na założeniu, że w wielu przypadkach mniej niż optymalna trasa jest lepsza niż brak trasy w ogóle, a rozkład jazdy z kilkoma naruszeniami ograniczeń jest lepszy niż całkowity brak rozkładu jazdy . Jeden typ algorytmu aproksymacji daje wyniki, które są gwarantowane „blisko" optymalnego rozwiązania. Na przykład, istnieje dość sprytny algorytm dla pewnej wersji problemu komiwojażera (gdzie przyjmuje się, że wykres reprezentuje realistyczną dwuwymiarową mapę), który działa w czasie sześciennym i tworzy trasę, która na pewno nie będzie dłuższa niż 1,5 razy większa (nieznana) optymalna trasa. Gwarancja opiera się oczywiście na rygorystycznym dowodzie matematycznym. W rzeczywistości istnieje znacznie mniej wyrafinowany algorytm, który gwarantuje wycieczkę nie dłuższą niż dwukrotność optymalnej trasy, i który warto spróbować skonstruować. Polega ona na znalezieniu minimalnego drzewa opinającego i dwukrotnym przejściu każdej jego krawędzi. (Dlaczego trasa nie jest dłuższa niż dwukrotna optymalna?). Inne podejście do aproksymacji prowadzi do rozwiązań, które nie gwarantują, że zawsze będą znajdować się w pewnym ustalonym zakresie optimum, ale prawie zawsze będą bardzo bliskie optimum. W tym przypadku wymagana analiza jest podobna do tej przeprowadzanej dla wydajności algorytmów dla średnich przypadków i zwykle obejmuje nieco zaawansowaną teorię prawdopodobieństwa. Na przykład, istnieje szybki algorytm dla problemu komiwojażera, który dla niektórych wykresów wejściowych może dawać objazdy znacznie dłuższe niż optymalne. Jednak w zdecydowanej większości przypadków algorytm zapewnia prawie optymalne trasy. Ten konkretny algorytm oparty jest na heurystyce, czyli regule kciuka , dzięki czemu wykres jest najpierw podzielony na wiele lokalnych klastrów zawierających bardzo niewiele punktów. Następnie znajduje się optymalne objazdy w każdym z nich, a następnie łączy je w obchód globalny metodą podobną do algorytmu zachłannego dla problemu drzewa opinającego. Czy problemy NP-zupełne zawsze dopuszczają algorytmy szybkiego przybliżania? Jeśli chcemy być nieco elastyczni w naszych wymaganiach dotyczących optymalności, czy możemy być pewni, że odniesiemy sukces? Cóż, to trudne pytanie. Ludzie mieli nadzieję, że dla większości problemów NP-zupełnych można znaleźć potężne algorytmy aproksymacyjne, nawet nie znając odpowiedzi na prawdziwe pytanie P vs. NP. Mieliśmy nadzieję, że być może uda nam się zbliżyć do optymalnego wyniku, nawet jeśli znalezienie prawdziwego optimum nadal będzie poza naszym zasięgiem. Jednak w ostatnich latach ta nadzieja zadała miażdżący cios odkryciem kolejnych złych wiadomości: dla wielu problemów NP-zupełnych (nie wszystkich) przybliżenia okazują się nie łatwiejsze niż pełne rozwiązania! Wykazano, że znalezienie dobrego algorytmu aproksymacji dla dowolnego z tych problemów jest równoznaczne ze znalezieniem dobrego rozwiązania nieprzybliżonego. Ma to następującą uderzającą konsekwencję. Znalezienie dobrego algorytmu aproksymacji dla jednego z tych specjalnych problemów NP-zupełnych wystarczy, aby wszystkie problemy NP-zupełne były wykonalne; to znaczy ustaliłoby, że P = NP. Odwrotnie, jeśli P ≠ NP, to nie tylko problemy NP-zupełne nie mają dobrych pełnych rozwiązań, ale wielu z nich nie można nawet przybliżyć! Jako przykład rozważmy problem dotyczący minimalnej liczby kolorów wymaganej do pokolorowania dowolnego wykresu. Ponieważ jest to NP-zupełne, badacze szukali algorytmu aproksymacji, który zbliżyłby się do liczby optymalnej w czasie wielomianu. Być może więc istnieje metoda, która na podstawie wykresu wejściowego znajduje liczbę, która nigdy nie jest większa niż 10% lub 20% od minimalnej liczby kolorów potrzebnej do pokolorowania sieci. Okazuje się, że to jest tak trudne, jak w rzeczywistości. Wykazano, że jeśli dowolny algorytm wielomianowy może znaleźć kolorowanie mieszczące się w ustalonym stałym współczynniku minimalnej liczby kolorów potrzebnej do pokolorowania grafu, to istnieje algorytm wielomianowy dla pierwotnego problemu znalezienia optymalnego numer sam. Ma to dalekosiężną konsekwencję, którą właśnie opisaliśmy: odkrycie dobrego algorytmu aproksymacji do kolorowania wykresów jest tak samo trudne, jak wykazanie, że P = NP.

Problemy prawdopodobnie nierozwiązywalne

Pomimo naszej niezdolności do znalezienia rozsądnych rozwiązań licznych problemów NP-zupełnych, nie jesteśmy pewni, czy takie rozwiązania nie istnieją; z tego co wiemy, problemy NP.-zupełne mogą mieć bardzo wydajne rozwiązania wielomianowe. Należy jednak zdać sobie sprawę, że wiele problemów (choć nie NP-zupełnych) okazało się nierozwiązywalnych i nie ograniczają się one tylko do tych (takich jak Wieże Hanoi), których wyniki są nadmiernie długie. Oto kilka przykładów. Pod koniec Części 1 omówiliśmy problem rozstrzygnięcia, czy przy danej szachowej konfiguracji szachów białe mają gwarantowaną strategię wygrywającą. Jak się zorientowałeś, drzewo gry w szachy rośnie wykładniczo. Innymi słowy, jeśli ustalimy jakąś początkową konfigurację u korzenia drzewa, a każdy węzeł zostanie rozszerzony w dół przez potomków odpowiadających wszystkim możliwym kolejnym konfiguracjom, rozmiar drzewa ogólnie staje się wykładniczy w swojej głębokości. Jeśli chcemy spojrzeć na N ruchów do przodu, być może będziemy musieli rozważyć konfiguracje KN, dla pewnej ustalonej liczby K, która jest większa niż 1. Ten fakt nie oznacza, że szachy jako gra są trudne do wykonania. W rzeczywistości, ponieważ w całej grze jest tylko skończenie wiele konfiguracji (choć jest ich bardzo dużo), problem strategii wygrywającej nie jest tak naprawdę problemem algorytmicznym, dla którego możemy mówić o wydajności rzędu wielkości. Problem algorytmiczny powszechnie kojarzony z szachami obejmuje uogólnioną wersję, w której dla każdego N istnieje inna gra, rozgrywana na planszy N na N, zestaw bierek i dozwolone ruchy są odpowiednio przedłużane. Wykazano, że szachy uogólnione, jak również uogólnione warcaby, mają dolną granicę czasu wykładniczego. Jest zatem nie do udowodnienia. Oprócz tych nieco wymyślonych uogólnień gier o stałym rozmiarze, kilka bardzo prostych gier, których początkowe konfiguracje różnią się rozmiarem, również mają dolne ograniczenia w czasie wykładniczym. Jedna z nich nazywa się blokadą dróg i jest rozgrywana przez dwóch graczy, Alicję i Boba, na sieci przecinających się dróg, z których każda ma jeden z trzech kolorów. (Drogi mogą przechodzić pod innymi.) Niektóre skrzyżowania są oznaczone jako „Alice wygrywa" lub „Bob wygrywa", a każdy gracz ma flotę samochodów, które zajmują określone skrzyżowania. W swojej (lub jej) turze gracz przesuwa jeden ze swoich samochodów wzdłuż odcinka drogi, którego wszystkie segmenty muszą być tego samego koloru, do nowego skrzyżowania, o ile po drodze nie spotkają się żadne samochody. Zwycięzcą zostaje pierwszy gracz, który dotrze do jednego ze swoich „wygranych" skrzyżowań.

Problem algorytmiczny wprowadza opis sieci, z samochodami umieszczonymi na określonych skrzyżowaniach, i pyta, czy Alicja (czyj jest zakręt) ma zwycięską strategię. Problem blokady na wyłączność ma dolną granicę czasu wykładniczego, co oznacza, że istnieje stała liczba K (większa niż 1) taka, że każdy algorytm rozwiązujący problem wymaga czasu, który rośnie co najmniej tak szybko, jak KN , gdzie N jest liczbą skrzyżowań w sieć. Innymi słowy, o ile pewne konfiguracje mogą być łatwe do przeanalizowania, to nie ma i nigdy nie będzie praktycznej metody algorytmicznej na ustalenie, czy dany gracz ma gwarantowaną strategię wygrania gry z blokadą. Dla najlepszego algorytmu, jaki możemy zaprojektować, zawsze będą stosunkowo małe konfiguracje, które spowodują, że będzie działał przez nierozsądny czas. Powinniśmy zauważyć, że te z natury problemy z czasem wykładniczym nie dopuszczają tego rodzaju krótkich certyfikatów, jakie mają problemy NP-zupełne. Nie tylko trudno jest udowodnić, czy istnieje zwycięska strategia dla gracza z blokadą dróg z danej konfiguracji, ale także niemożliwie czasochłonne jest przekonanie kogoś, że taka istnieje, jeśli takowa jest.

Udowodniony nierozwiązywalny problem z satysfakcją

Inny przykład problemu, którego nie da się udowodnić, dotyczy spełnialności logicznej. Wcześniej spotkaliśmy się z rachunkiem zdań, czyli formalizmem, który umożliwia pisanie zdań składających się z logicznych kombinacji twierdzeń podstawowych. Zbiór wartości logicznych dla twierdzeń podstawowych określa wartość logiczną samego zdania. Zdanie w rachunku zdań ma zatem charakter

„statyczny” — jego prawdziwość zależy tylko od obecnych wartości prawdziwości jego składników. W rozdziale 5 krótko omówiliśmy logikę dynamiczną, w której wolno nam używać konstrukcji:

after (A, F)

gdzie A jest algorytmem, a F jest stwierdzeniem lub stwierdzeniem. Twierdzi, że F jest prawdziwe po wykonaniu A. Tu oczywiście prawdziwość zdania nie zależy już tylko od stanu rzeczy w chwili obecnej, ale także od stanu po tym, jak algorytm jest wykonany. Jedna z wersji dynamicznej logiki, zwana dynamiczną logiką zdań (lub w skrócie PDL), ogranicza algorytmy lub programy dozwolone wewnątrz konstrukcji po jako kombinacje nieokreślonych programów elementarnych. Tak jak możemy budować kompleks twierdzenia z podstawowych symboli asercji, używając &, ∨ ˜ i →, możemy teraz budować złożone programy z podstawowych symboli programu, używając konstrukcji programistycznych, takich jak sekwencjonowanie, rozgałęzienia warunkowe i iteracja. Programy i asercje są następnie łączone za pomocą konstrukcji after. Poniżej znajduje się zdanie PDL, które stwierdza, że E jest fałszywe po tym, jak dwa programy wewnątrz po są wykonywane w kolejności:

after(while E do A end; if E then do B end, ∼E)

To zdanie jest zawsze prawdziwe, bez względu na to, co oznacza twierdzenie E, i bez względu na to, co oznaczają programy A i B, nawet jeśli B może mieć możliwość zmiany wartości logicznej E. Zachęcamy do przekonania się o tym fakcie. Problem spełnialności dla rachunku zdań jest interesujący, ponieważ dotyczy wykonalności wnioskowania o zdaniach logicznych o charakterze statycznym. W tym samym duchu problem spełnialności dla PDL jest interesujący, ponieważ dotyczy wykonalności wnioskowania o zdaniach logicznych o charakterze dynamicznym, obejmujących programy i niektóre z ich najbardziej elementarnych właściwości. Problem spełnialności dla rachunku zdań jest NP-zupełny, a zatem podejrzewa się, że jest nierozwiązywalny. Z drugiej strony problem spełnialności dla PDL ma dolną granicę czasu wykładniczego i dlatego wiadomo, że jest nierozwiązywalny. Nie ma i nigdy nie będzie metody algorytmicznej, za pomocą której można by w praktyce decydować, czy zdania PDL mogą być prawdziwe. Każdy taki algorytm z konieczności będzie działał przez niesamowitą ilość czasu na pewnych zdaniach o bardzo rozsądnym rozmiarze. Powinniśmy zauważyć, że wszystkie opisane właśnie problemy (szachy, warcaby, blokada dróg i PDL) dopuszczają algorytmy czasu wykładniczego, tak że mamy pasujące granice górne i dolne, a zatem dokładnie znamy ich status wykonalności; wszystkie są z natury problemami czasu wykładniczego.

Problemy, które są jeszcze trudniejsze!

Ta część została oparta na założeniu, że problemy algorytmiczne, których najlepsze algorytmy są wykładnicze w czasie, są niewykonalne. Są jednak problemy, które są jeszcze gorsze. Wśród nich jedne z najciekawszych dotyczą spełnialności i określania prawdy w różnych bogatych formalizmach logicznych. Spotkaliśmy się już z rachunkiem zdań i PDL, w których podstawowe twierdzenia były po prostu symbolami typu E i F, które mogły przybierać wartości prawdziwościowe. Jednak gdy formułuje się stwierdzenia dotyczące rzeczywistych obiektów matematycznych, takich jak liczby, chcemy, aby podstawowe twierdzenia były bardziej wyraziste. Chcielibyśmy móc napisać X = 15 lub Y + 8 > Z; chcielibyśmy móc rozważać zbiory liczb lub innych obiektów i mówić o wszystkich elementach zbioru; chcielibyśmy porozmawiać o istnieniu pierwiastków o określonych właściwościach i tak dalej. Istnieje wiele logicznych formalizmów, które zaspokajają takie pragnienia, i można w nich zapisać najciekawsze matematyczne twierdzenia, przypuszczenia i twierdzenia. Jest zatem naturalne, że informatycy poszukują skutecznych metod określania, czy zdania w takich formalizmach są prawdziwe; jest to jeden ze sposobów ustalenia absolutnej prawdy matematycznej. Teraz wiemy, że spełnialności w rachunku zdań najprawdopodobniej nie da się określić w czasie krótszym niż wykładniczy, ponieważ problem jest NP-zupełny, a prawda w PDL zdecydowanie nie może, ponieważ problem ten jest dowodem

wykładniczym. Kilka bardziej skomplikowanych formalizmów jest znacznie gorszych. Rozważmy funkcję 2^{2N}, która wynosi 2 × 2 × . . . × 2, przy czym 2 pojawia się 2^N razy. Jeśli N wynosi 5, wartość znacznie przekracza miliard, natomiast jeśli N wynosi 9, wartość jest znacznie większa niż liczba protonów w znanym nam wszechświecie. Funkcja 22N odnosi się oczywiście do nieuzasadnionej 2^N, tak jak 2N do bardzo rozsądnej funkcji N. Jest zatem podwójnie nieuzasadniona i faktycznie jest określana jako funkcja podwójnie wykładnicza. Funkcja potrójnie wykładnicza 2^{22N} jest zdefiniowana podobnie, jak wszystkie K-krotne funkcje wykładnicze w czasie $2^{2\cdots 2N}$ z K występami równymi 2. Wykazano, że kilka formalizmów ma dolne granice podwójnego wykładniczego czasu. Wśród nich jest logika znana jako arytmetyka Presburgera, która pozwala nam mówić o dodatnich liczbach całkowitych i zmiennych, których wartości są dodatnimi liczbami całkowitymi. Pozwala również na operację „+" i symbol „=". Asercje łączymy za pomocą operacji logicznych rachunku zdań, a także kwantyfikatorów existsX i forallX. Na przykład poniższa formuła stwierdza, że istnieje nieskończenie wiele liczb parzystych, stwierdzając, że dla każdego X istnieje parzyste Y, które jest co najmniej tak duże jak X:

forallX existsY existsZ (X + Z = Y & existsW (W + W = Y))

Podczas gdy prawda w arytmetyce Presburgera jest prawdopodobnie podwójnie wykładnicza, inny formalizm, zwany WS1S, jest znacznie gorszy. W WS1S możemy mówić nie tylko o (dodatnich) liczbach całkowitych, ale także o zbiorach liczb całkowitych. Możemy stwierdzić, że zbiór S zawiera element X, pisząc X &∈; S.

Poniżej znajduje się prawdziwy wzór WS1S, stwierdzający, że każda liczba parzysta jest uzyskiwana przez dodanie 2 do 0 pewną liczbę razy.

forallB ((0 ∈ B & forallX (X ∈ B → X + 2 ∈ B)) → forallY (existW (Y = W + W) →Y ∈ B))

Dokonuje tego poprzez stwierdzenie, że każdy zbiór B, który zawiera 0 i zawiera X + 2 za każdym razem, gdy zawiera X, musi zawierać wszystkie liczby parzyste. WS1S jest niewyobrażalnie trudny do zanalizowania. Wykazano, że nie dopuszcza żadnego algorytmu wykładniczego K-krotnego dla żadnego K! (tu wykrzyknik, nie silnia...) Oznacza to, że dla dowolnego algorytmu A określającego prawdziwość formuł WS1S (a takie algorytmy są), oraz dla dowolnej stałej liczby K będą formuły o długości N, dla większej i większe N, które będzie wymagało działania A przez czas dłuższy niż $2^{2\cdots 2N}$ jednostek czasu, przy czym K pojawia się jako 2. W takich niszczących przypadkach mówimy, że problem decyzyjny jest, do udowodnienia, nieelementarny. Nie tylko jest nieuleczalna, ale nie jest nawet podwójnie lub potrójnie nieuleczalna. Jego wydajność czasowa jest gorsza niż jakikolwiek wykładniczy k-krotny i możemy słusznie powiedzieć, że ma nieograniczoną trudność.

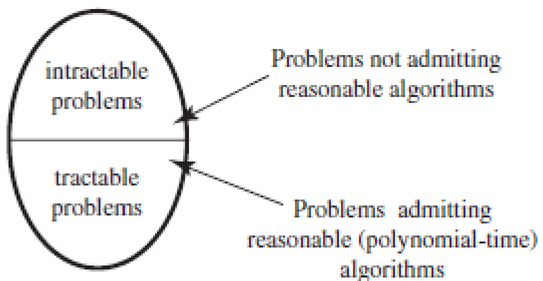
Nieuzasadnione ilości miejsca w pamięci

Chociaż zobowiązaliśmy się skoncentrować na wydajności czasu, musimy poświęcić chwilę na kontemplację nieuzasadnionych wymagań dotyczących miejsca w pamięci. Istnieją problemy algorytmiczne, które, jak udowodniono, mają dolne granice przestrzeni wykładniczej. Oznacza to, że każdy algorytm rozwiązujący je będzie wymagał, powiedzmy, 2<sup>N<sup> komórek pamięci na pewnych wejściach o rozmiarze N. W rzeczywistości można wykazać, że jest to konsekwencja podwójnego wykładniczego czasu dolnego ograniczenia (jak w przypadku problemu prawdy dla Presburgera arytmetyka) jest dolną granicą w przestrzeni wykładniczej. Podobnie, nieelementarne dolne ograniczenie czasu (jak w przypadku prawdy w WS1S) implikuje również nieelementarne dolne ograniczenie dotyczące przestrzeni. Te fakty mają uderzające konsekwencje. Jeśli problem ma dolną granicę 2<sup>N<sup> dla przestrzeni pamięci, to dla dowolnego algorytmu będą dane wejściowe o całkiem rozsądnej wielkości (mniej niż 270, konkretnie), które wymagałyby tak dużo miejsca na dane pośrednie, że nawet gdyby każdy bit miał być wielkości protonu, cały znany wszechświat nie

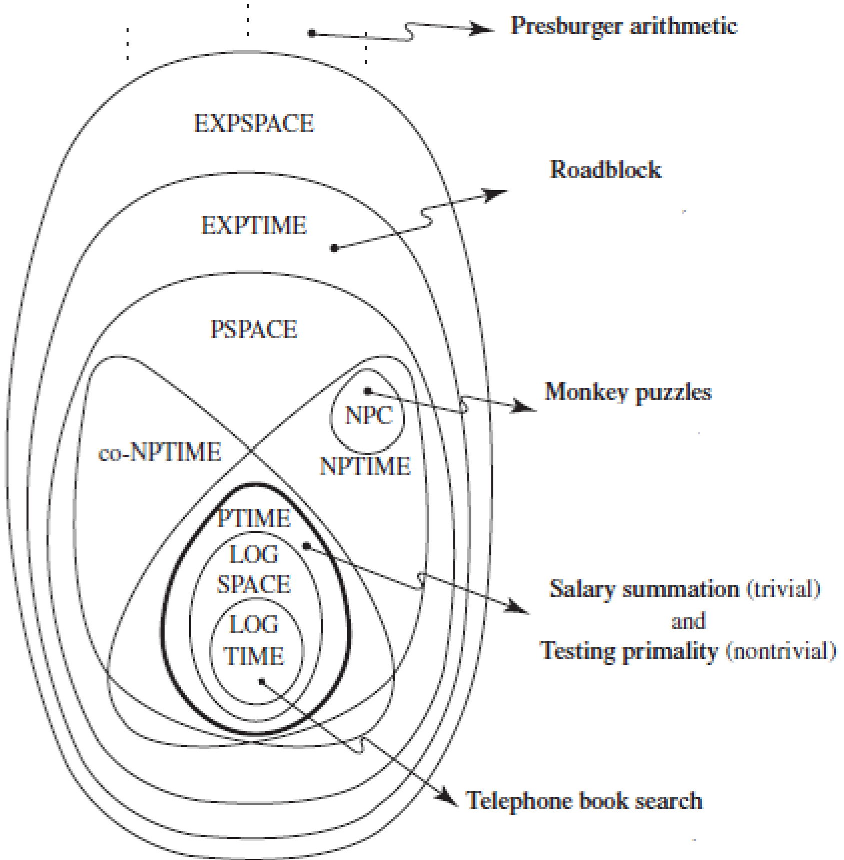
wystarczyłby, aby to wszystko zapisać! Sytuacja jest wyraźnie niewyobrażalnie gorsza w przypadku nieelementarnych ograniczeń przestrzeni.

Badania nad klasami złożoności i nieusuwalności

W połowie lat 60. ludzie zaczęli zdawać sobie sprawę ze znaczenia uzyskiwania algorytmów wielomianowych dla problemów algorytmicznych, a znaczenie linii podziału na rysunku stało się oczywiste.



Od tego czasu zagadnienia i koncepcje omawiane w tej części są przedmiotem intensywnych i szeroko zakrojonych badań wielu informatyków teoretycznych. Co jakiś czas dla problemu, którego status wykonalny/niewykonalny był nieznany, znajduje się algorytm wielomianu lub dolna granica czasu wykładniczego. Najbardziej uderzającym przykładem z ostatnich lat jest wspomniane wcześniej testowanie pierwszości. Innym jest planowanie liniowe, lepiej znane jako programowanie liniowe. Planowanie liniowe to ogólne ramy, w ramach których możemy sformułować wiele rodzajów problemów związanych z planowaniem pojawiających się w organizacjach, w których należy sprostać ograniczeniom czasu, zasobów i personelu w sposób efektywny kosztowo. Należy podkreślić, że problem planowania liniowego nie jest NP-zupełny, ale najlepszym algorytmem, jaki ktokolwiek był w stanie znaleźć, była procedura czasu wykładniczego, znana jako metoda simpleks. Pomimo faktu, że niektóre dane wejściowe zmuszały metodę simpleks do działania przez wykładniczy okres czasu, były one raczej wymyślone i raczej nie pojawiały się w praktyce; gdy metoda ta była wykorzystywana do prawdziwych problemów, nawet o niebanalnych rozmiarach, zwykle działała bardzo dobrze. Niemniej jednak oficjalnie nie było wiadomo, że problem występuje w P, ani nie było dolnej granicy wskazującej, że tak nie jest. W 1979 r. znaleziono genialny algorytm wielomianowy dla tego problemu, ale było to trochę rozczarowanie. Metoda simplex z czasem wykładniczym przewyższała ją w wielu przypadkach pojawiających się w praktyce. Niemniej jednak pokazał, że programowanie liniowe jest w P. Co więcej, ostatnie prace oparte na tym algorytmie zaowocowały wydajniejszymi wersjami, a ludzie obecnie wierzą, że niedługo będzie szybkim algorytmem wielomianowym do planowania liniowego, który będzie przydatny w praktyce dla wszystkich danych wejściowych o rozsądnej wielkości. Ten rodzaj pracy ma na celu poszerzenie naszej wiedzy o konkretnych problemach i jest analogiczny do poszukiwania wydajnych algorytmów w samym P, jak omówiono w rozdziale 6. Praca o bardziej ogólnym charakterze obejmuje klasy złożoności, takie jak same P i NP. Tutaj interesuje nas identyfikacja dużych i znaczących klas problemów, z których wszystkie mają wspólne cechy wydajności. Używając przedrostka LOG dla logarytmicznego, P dla wielomianu, EXP dla wykładniczego i 2EXP dla podwójnie wykładniczego, możemy napisać LOGTIME dla klasy problemów rozwiązywalnych w czasie logarytmicznym, PTIME dla klasy o nazwie P powyżej, PSPACE dla problemów rozwiązywalnych za pomocą wielomianowa ilość miejsca w pamięci i tak dalej. Możemy wtedy ustalić następujące relacje włączenia (gdzie ⊆ oznacza „jest podzbiorem”)



LOGTIME ⊆ LOGSPACE ⊆ PTIME ⊆ PSPACE ⊆ EXPTIME ⊆ EXPSPACE ⊆ 2EXPTIME . . .

Jeśli dodamy przedrostek N dla „niedeterministycznego", pisząc na przykład NPTIME dla NP, otrzymamy znacznie więcej klas i pojawi się wiele interesujących pytań o współzależności. Na przykład wiadomo, że NP mieści się między PTIME i PSPACE, ale w wielu przypadkach nikt nie wie, czy symbole ⊆ reprezentują ścisłe wtrącenia, czy nie. Czy jest problem w PSPACE, którego nie ma w PTIME? Jeśli tak, chcielibyśmy wiedzieć, który z dwóch wtrąceń w następującej kolejności jest ścisły:

PTIME ⊆ NPTIME ⊆ PSPACE

Ścisłość pierwszego to oczywiście tylko problem P vs. NP.

Możemy również rozważyć podwójne klasy złożoności, takie jak np. co-NP, która jest klasą problemów, których dopełnienie lub wersja dualna (w której odpowiedzi „tak" i „nie" są zamienione) występuje w NP. Nie wiadomo na przykład, czy NP = co-NP. Z drugiej strony wiadomo, że jeśli NP = co-NP, to także P = NP. Odwrotność jednak nie jest prawdą; może być tak, że NP i co-NP są równe, podczas gdy P i NP nie są. Pojawia się wiele innych pytań, na niektóre z których odpowiedzi są znane, a na inne nie. Drugim ważnym obszarem badań są rozwiązania przybliżone, a rozwiązania gwarantowane przeciętnie są dobre. Są one poszukiwane, nawet jeśli wiadomo lub podejrzewa się, że problem jest trudny do rozwiązania. Naukowcy wciąż próbują zrozumieć powiązania między nieodłączną złożonością czasową najgorszego przypadku problemu a dostępnością szybkich przybliżonych rozwiązań.

Pomimo dość przygnębiającego charakteru faktów omawianych w tym rozdziale, wydaje się, że najczęstsze problemy pojawiające się w codziennych zastosowaniach komputerów można skutecznie rozwiązać. To stwierdzenie jest jednak nieco mylące, ponieważ mamy tendencję do utożsamiania

„zwykłych" i „codziennych" problemów z tymi, które umiemy rozwiązać. W rzeczywistości coraz więcej problemów pojawiających się w nietrywialnych zastosowaniach komputerów okazuje się NP-zupełne lub gorsze. W takich przypadkach musimy uciekać się do algorytmów aproksymacyjnych lub probabilizmu i heurystyki. Zanim jednak wrócimy do bardziej wesołego materiału, nadejdą gorsze wieści. Niektóre problemy algorytmiczne nie dają żadnych rozwiązań, nawet nierozsądnych.

Nieobliczalność i nierozstrzygalność

W kwietniu 1984 roku Time Magazine opublikował artykuł z okładki na temat oprogramowania komputerowego. W skądinąd znakomitym artykule był akapit, w którym cytowano wydawcę magazynu programistycznego: „Włóż do komputera odpowiedni rodzaj oprogramowania, a zrobi to, co zechcesz”. Mogą istnieć ograniczenia co do tego, co możesz zrobić z samymi maszynami, ale nie ma ograniczeń co do tego, co możesz zrobić z oprogramowaniem. W pewnym sensie wyniki Części 7 już zaprzeczają temu twierdzeniu, pokazując, że pewne problemy są niemożliwe do rozwiązania. Możemy jednak argumentować, że niewykonalność jest w rzeczywistości konsekwencją niewystarczających zasobów. Mając wystarczająco dużo czasu i miejsca w pamięci (choć nieracjonalnie duże ilości), być może każdy problem algorytmiczny można w zasadzie rozwiązać za pomocą odpowiedniego oprogramowania. Rzeczywiście, powody, dla których ludziom często nie udaje się nakłonić swoich komputerów do robienia tego, czego chcą, można z grubsza podzielić na trzy kategorie: niewystarczająca ilość pieniędzy, niewystarczająca ilość czasu i niewystarczająca ilość mózgów. Za więcej pieniędzy można było kupić większy i bardziej zaawansowany komputer, wspomagany lepszym oprogramowaniem, a potem być może wykonać zadanie. Mając więcej czasu, można by dłużej czekać na zakończenie czasochłonnych algorytmów, a mając więcej mózgów, można by być może wynaleźć algorytmy dla problemów, które wydają się nie do rozwiązania. Problemy algorytmiczne, które chcemy omówić w tym rozdziale, są takie, że żadna ilość pieniędzy, czasu ani rozumu nie wystarczy do znalezienia rozwiązania. Oczywiście nadal wymagamy, aby algorytmy kończyły się dla każdego legalnego wprowadzenia w określonym czasie, ale teraz pozwalamy, aby ten czas był nieograniczony. Algorytm może trwać tak długo, jak chce na każdym wejściu, ale w końcu musi się zatrzymać i wytworzyć pożądany wynik. Podobnie podczas pracy z danymi wejściowymi algorytm otrzyma dowolną ilość pamięci, o jaką poprosi. Mimo to zobaczymy ciekawe i ważne problemy, dla których po prostu nie ma algorytmów i nie ma znaczenia, jak sprytni jesteśmy, jak wyrafinowane i potężne są nasze komputery. Takie fakty mają głębokie implikacje filozoficzne, nie tylko dotyczące granic maszyn stworzonych przez człowieka, ale także naszych własnych granic jako śmiertelników o skończonej masie. Nawet gdybyśmy otrzymali nieograniczoną ilość ołówka i papieru oraz nieograniczoną długość życia, mielibyśmy ściśle określone problemy, których nie bylibyśmy w stanie rozwiązać. Są ludzie, którzy z różnych powodów sprzeciwiają się wyciąganiu tak rozbudowanych wniosków na podstawie samych wyników algorytmicznych. Biorąc pod uwagę fakt, że zagadnienie w tak rozbudowanej formie zdecydowanie zasługuje na znacznie szersze potraktowanie, pozostaniemy tutaj przy czystej algorytmice, a głębsze implikacje pozostawimy filozofom i neurobiologom. Należy jednak pamiętać o istnieniu takich implikacji.

Zasady gry

Żeby to wyjaśnić, należy podkreślić, że pytania dotyczące zdolności komputera do prowadzenia firm, podejmowania dobrych decyzji czy miłości nie mają znaczenia w naszych obecnych dyskusjach, ponieważ nie dotyczą precyzyjnie zdefiniowanych problemów algorytmicznych. Innym faktem wartym przypomnienia jest wymóg, aby problem algorytmiczny był powiązany ze zbiorem legalnych danych wejściowych, a proponowane rozwiązanie dotyczyło wszystkich danych wejściowych w zbiorze. W konsekwencji, jeśli zbiór danych wejściowych jest skończony, problem zawsze dopuszcza rozwiązanie. Jako prosty przykład, dla problemu decyzyjnego, którego jedynymi prawnymi danymi wejściowymi są pozycje I_1, I_2, … , I_K , istnieje algorytm, który „zawiera” tabelę z odpowiedziami K. Algorytm może brzmieć:

(1) jeśli wejście to I_1, to wyjście „tak” i zatrzymaj się;

(2) jeśli wejście to I_2> to wyjście „tak” i zatrzymanie;

(3) jeśli wejście to I_3, to wyjście „nie” i zatrzymaj się;

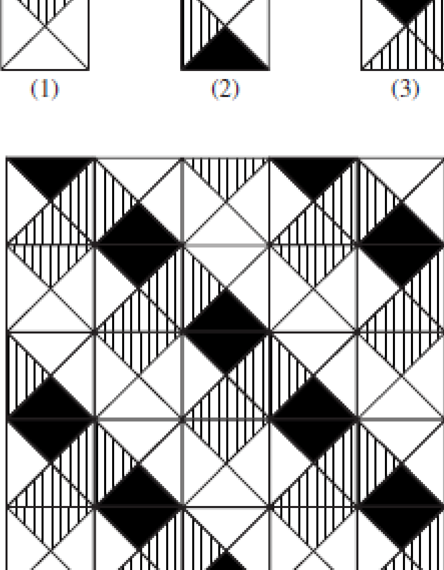
…

(K) jeśli wejście to I_K, to wypisz „tak” i zatrzymaj się.

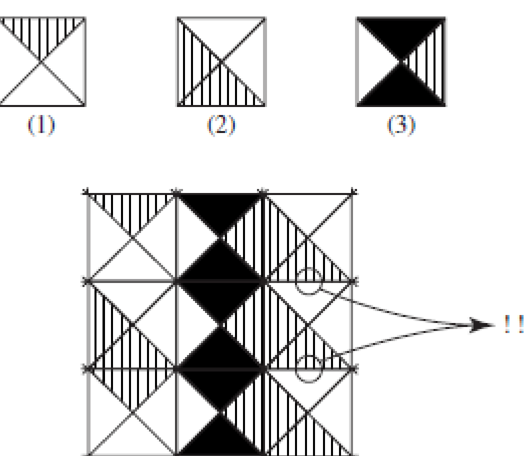
To oczywiście działa, ponieważ skończoność zestawu wejść umożliwia zestawienie wszystkich par wejścia/wyjścia i „okablowanie” ich do algorytmu. Wykonanie tabulacji (czyli skonstruowanie takiego algorytmu opartego na tabeli) może być trudne, ale nie interesuje nas tutaj ta „metatrudność”. Dla naszych obecnych celów wystarczy, że skończone problemy zawsze mają rozwiązania. To właśnie problemy z nieskończenie wieloma wejściami są naprawdę interesujące. W takich przypadkach skończony algorytm musi być w stanie poradzić sobie z nieskończenie wieloma przypadkami, co skłania do zakwestionowania samego istnienia takich algorytmów dla wszystkich problemów. Aby móc sformułować nasze twierdzenia jak najdokładniej, ale tak ogólnie, jak to tylko możliwe, będziesz musiał pogodzić się z pewnym rodzajem luzu terminologicznego w tym rozdziale, aby w następnym być w pełni uzasadnionym. W szczególności zakłada się, że arbitralny, ale ustalony język programowania wysokiego poziomu L jest środkiem do wyrażania algorytmów, a słowo „algorytm” będzie używane jako synonim słowa „program w języku L”. W szczególności, kiedy mówimy „żaden algorytm nie istnieje”, naprawdę mamy na myśli, że żaden program nie może być napisany w języku L. Ta konwencja może tutaj wyglądać nieco pretensjonalnie, najwyraźniej osłabiając nasze twierdzenia. Bynajmniej. W następnym rozdziale zobaczymy, że przy założeniu nieograniczonych zasobów wszystkie języki programowania są równoważne. Tak więc, jeśli żaden program nie może być napisany w L, żaden program nie może być napisany w jakimkolwiek efektywnie wdrażalnym języku, uruchomionym na dowolnym komputerze o dowolnym rozmiarze i kształcie, teraz lub w dowolnym momencie w przyszłości.

Problem kafelków: przykład

Poniższy przykład przypomina problem małpiej łamigłówki z Części 7. Problem polega na pokryciu dużych obszarów kwadratowymi płytkami lub kartami z kolorowymi krawędziami, tak że sąsiednie krawędzie są monochromatyczne. Płytka to kwadrat 1 na 1, podzielony na cztery przez dwie przekątne, każda ćwiartka pokolorowana jakimś kolorem. Podobnie jak w przypadku kart z małpami, zakładamy, że kafelki mają ustaloną orientację i nie można ich obracać. (W tym przypadku założenie jest w rzeczywistości konieczne. Czy widzisz dlaczego?) Problem algorytmiczny wprowadza pewien skończony zbiór T opisów płytek i pyta, czy jakikolwiek skończony obszar o dowolnej wielkości może być pokryty tylko przy użyciu płytek o rodzaje opisane w T , takie, że kolory na dowolnych dwóch stykających się krawędziach są takie same. Zakłada się, że dostępna jest nieograniczona liczba płytek każdego rodzaju, ale liczba rodzajów płytek jest skończona. Pomyśl o układaniu płytek w domu. Dane wejściowe T to opis różnych dostępnych rodzajów płytek, a ograniczenie dopasowania kolorów odzwierciedla zasadę narzuconą przez projektanta wnętrz ze względów estetycznych. Pytanie, które chcielibyśmy zadać z wyprzedzeniem, brzmi: czy pomieszczenie o dowolnej wielkości może być wyłożone płytkami tylko z dostępnych rodzajów płytek, przy zachowaniu ograniczenia? Ten problem algorytmiczny i jego warianty są powszechnie znane jako problemy kafelkowania, ale czasami są nazywane problemami domina, z powodu podobnego do domina ograniczenia dotykania krawędzi.



Rysunek 1 pokazuje trzy typy płytek i płytki 5 na 5. Nie powinieneś mieć trudności ze sprawdzeniem, czy wzór w dolnej części rysunku może być rozciągany w nieskończoność we wszystkich kierunkach, aby uzyskać płytki o dowolnym obszarze. Natomiast jeśli zamienimy dolne kolory kafelków (2) i (3), to dość łatwo można wykazać, że nawet bardzo małych pomieszczeń w ogóle nie da się wykafelkować. Rysunek 8.2 ma na celu zilustrowanie tego faktu. Algorytm rozwiązywania problemu kafelkowania powinien zatem odpowiedzieć „tak” na wejścia z rysunku 1 i „nie” na dane z rysunku 2.



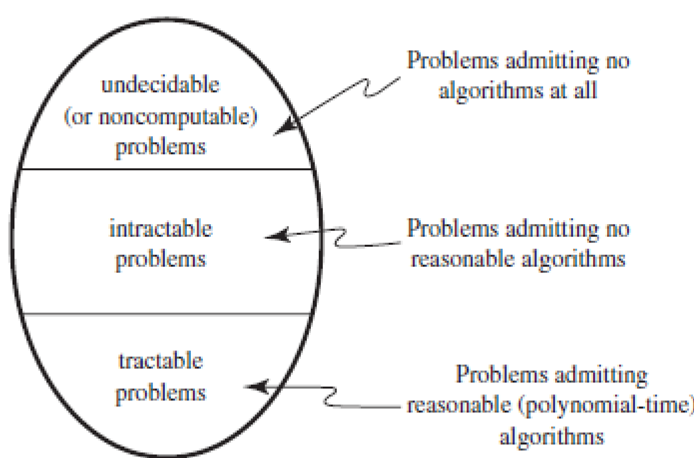
Problem polega na tym, aby w jakiś sposób zmechanizować lub „algorytmizować” rozumowanie stosowane do generowania tych odpowiedzi. I tu pojawia się ciekawostka: tego rozumowania nie da się zmechanizować. Nie ma i nigdy nie będzie algorytmu rozwiązywania problemu kafelkowania! Dokładniej, dla każdego algorytmu, który możemy zaprojektować dla tego problemu, zawsze będą istnieć zbiory danych wejściowych T (w rzeczywistości będzie nieskończenie wiele takich zbiorów), na których algorytm albo będzie działał w nieskończoność i nigdy się nie zakończy, albo zakończy się z błędną odpowiedzią.

Jak możemy sformułować tak ogólne twierdzenie, nie ograniczając podstawowych operacji dozwolonych w naszych algorytmach? Z pewnością, jeśli coś jest dozwolone, następująca dwuetapowa procedura rozwiązuje problem:

(1) jeśli typy w T mogą kafelkować dowolny obszar, wypisz „tak" i zatrzymaj się;

(2) w przeciwnym razie wypisz „nie" i zatrzymaj się.

Odpowiedź tkwi w tym, że używamy tutaj „algorytmu" jako programu w konwencjonalnym języku programowania L. Żaden program w jakimkolwiek efektywnie wykonywalnym języku nie może poprawnie zaimplementować testu w linii (1) procedury, a zatem dla naszych celów , taka „procedura" w ogóle nie będzie uważana za algorytm. Problem algorytmiczny, który nie dopuszcza żadnego algorytmu, jest określany jako nieobliczalny; jeśli jest to problem decyzyjny, jak ma to miejsce w tym przypadku i w przypadku większości poniższych przykładów, jest określany jako nierozstrzygalny. Problem kafelków lub domina jest zatem nierozstrzygnięty. Nie ma możliwości skonstruowania algorytmu uruchamianego na komputerze, dowolnym komputerze, niezależnie od ilości czasu i wymaganej przestrzeni pamięci, który będzie miał możliwość decydowania, czy dowolne skończone zestawy typów kafelków mogą kafelkować obszary dowolnego rozmiar. Możemy teraz doprecyzować sferę problemów algorytmicznych pojawiających się w Części 7, biorąc pod uwagę problemy nieobliczalne. Rysunek 3 to aktualna wersja.

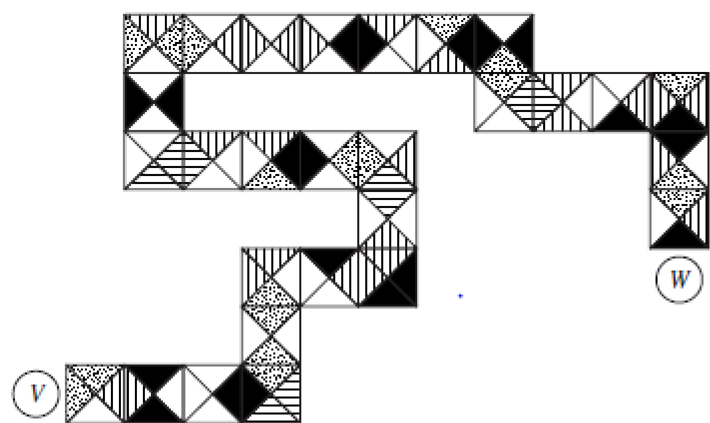


Interesujące jest to, że poniższy, nieco inaczej wyglądający problem, jest w rzeczywistości równoważny z tym, który właśnie opisałem. W tej wersji, zamiast wymagać, aby T był w stanie ułożyć sąsiadujące obszary o dowolnej wielkości, wymagamy, aby T był w stanie ułożyć sąsiadująco całą siatkę liczb całkowitych; czyli całą nieskończoną płaszczyznę. Jeden kierunek równoważności (jeśli możemy pokryć całą płaszczyznę, to możemy pokryć dowolnym skończonym obszarem) jest trywialny, ale argument, który ustala drugi kierunek, jest dość delikatny i zachęcamy do próby znalezienia go dla siebie. Co ciekawe, nierozstrzygalność tej wersji oznacza, że muszą istnieć zestawy kafelków T, które można wykorzystać do kafelkowania całej siatki, ale nie okresowo. Oznacza to, że chociaż taka litera T dopuszcza całkowite kafelkowanie siatki, kafelkowanie, w przeciwieństwie do tego z rysunku 1, nie składa się ze skończonej części, która powtarza się w nieskończoność we wszystkich kierunkach. Powodem tego jest to, że w przeciwnym razie moglibyśmy rozstrzygnąć problem za pomocą algorytmu, który przystąpiłby do wyczerpującego sprawdzania wszystkich skończonych obszarów, szukając albo skończonego obszaru, którego w ogóle nie można rozmieścić, albo takiego, który dopuszcza powtórzenie we wszystkich kierunkach. Przez „inaczej" rozumiemy, że gdyby każdy zestaw kafelków,

który dopuszcza pełne kafelkowanie siatki, dopuszczał również pełne okresowe kafelkowanie, algorytm ten miałby gwarancję, że zakończy się dla każdego wejścia z poprawnym wynikiem.

Bezgraniczność może wprowadzać w błąd

Niektórzy reagują na takie wyniki, mówiąc: „Cóż, oczywiście problem jest nierozstrzygnięty, ponieważ pojedyncze dane wejściowe mogą prowadzić do potencjalnie nieskończonej liczby spraw do sprawdzenia i nie ma mowy, abyś mógł wykonać nieskończoną pracę przez algorytm, który musi zakończyć się po skończonej liczbie kroków.” I rzeczywiście, tutaj jedno wejście T najwyraźniej wymaga sprawdzenia wszystkich obszarów o wszystkich rozmiarach (lub, równoważnie, jak wspomniano powyżej, pojedynczego obszaru o nieskończonym rozmiarze) i wydaje się, że nie ma sposobu na ograniczenie liczby przypadków, które mają być w kratkę. Ta zasada nieograniczoności – implikuje – nierozstrzygalności jest całkiem błędna i może być bardzo myląca. To tak, jakby powiedzieć, że każdy problem, który wydaje się wymagać wykładniczo wielu sprawdzeń, jest z konieczności nie do rozwiązania. W Części 7 widzieliśmy dwa problemy ścieżek hamiltonowskich i ścieżek Eulera, z których oba wydawały się wymagać przeszukania wszystkich wykładniczo wielu ścieżek na grafie wejściowym. Wykazano jednak, że drugi problem dopuszcza łatwy algorytm wielomianowy. Z nierozstrzygalnością można też wykazać dwa bardzo podobne warianty problemu, oba o pozornie nieograniczonym charakterze, które w dość zaskakujący sposób kontrastują z naruszeniem zasady. Dane wejściowe w obu przypadkach zawierają skończony zbiór T typów płytek i dwie lokalizacje V i W na nieskończonej siatce liczb całkowitych. Oba problemy pytają, czy możliwe jest połączenie V z W za pomocą „węża domina” składającego się z płytek z T , przy czym co dwie sąsiednie płytki mają monochromatyczne stykające się krawędzie .



Zauważ, że wąż pochodzący z V może skręcać się i obracać chaotycznie, docierając do nieskończenie odległych punktów, zanim zbiegnie się z W. Stąd, na pierwszy rzut oka, problem wymaga potencjalnie nieskończonych poszukiwań, co skłania nas do przypuszczenia, że również jest nierozstrzygnięty. Interesujące jest zatem, że rozstrzygalność problemu węża domina zależy od części płaszczyzny dostępnej do ułożenia płytek łączących. Oczywiście, jeśli ta część jest skończona, problem jest trywialnie rozstrzygalny, ponieważ istnieje tylko skończenie wiele możliwych węży, które można umieścić w danym skończonym obszarze. Rozróżnienie, którego chcemy dokonać, dotyczy dwóch nieskończonych porcji i jest całkiem sprzeczne z intuicją. Jeśli węże mogą płynąć gdziekolwiek (to znaczy, jeśli dozwoloną częścią jest cała płaszczyzna), problem jest rozstrzygalny, ale jeśli dozwolony obszar to tylko połowa płaszczyzny (powiedzmy, górna połowa), problem staje się nierozstrzygalny ! Ten ostatni przypadek wydaje się „bardziej ograniczony” niż pierwszy, a zatem być może „bardziej rozstrzygalny”. Fakty są jednak zupełnie inne. W rzeczywistości udowodniono, że problem węża

domina jest nierozstrzygnięty dla prawie każdego wyobrażalnego ograniczenia płaszczyzny nieskończonej, o ile rozważana część jest nieograniczona w dwóch prostopadłych kierunkach. Tak więc jest to nierozstrzygalne nie tylko w półpłaszczyźnie, ale w ćwiartce, w jednej ósmej itp. Najbardziej uderzający wynik można opisać jako ustalenie, że tylko jeden punkt stoi między rozstrzygalnością a nierozstrzygalnością: podczas gdy problem, jak widzieliśmy, jest rozstrzygalny na całej płaszczyźnie, staje się nierozstrzygalny, jeśli z płaszczyzny zostanie usunięty choćby jeden punkt! Mówiąc dokładniej, dane wejściowe to skończony zbiór T typów płytek, dwa punkty V i W na nieskończonej siatce liczb całkowitych oraz trzeci punkt U. Problem pyta, czy możliwe jest połączenie V z W przez legalnego węża domina skonstruowanego z płytek w T , których płytki mogą iść w dowolnym miejscu na płaszczyźnie z wyjątkiem punktu U.

Korespondencja słów i równoważność składniowa

Oto dwa dodatkowe nierozstrzygnięte problemy. Pierwszy, problem korespondencji słownej, polega na tworzeniu słowa na dwa różne sposoby. Jego dane wejściowe to dwie grupy słów nad jakimś skończonym alfabetem. Nazwijcie je X i Y. Problem pyta, czy możliwe jest łączenie słów z grupy X, tworząc nowe słowo, nazwijmy je Z, tak aby łączenie odpowiednich słów spośród Y s tworzyło to samo słowo złożone Z.

	1	2	3	4	5
X	*abb*	*a*	*bab*	*baba*	*aba*
Y	*bbab*	*aa*	*ab*	*aa*	*a*

(a) Admits a correspondence: 2, 1, 1, 4, 1, 5

	1	2	3	4	5
X	*bb*	*a*	*bab*	*baba*	*aba*
Y	*bab*	*aa*	*ab*	*aa*	*a*

(b) Admits no correspondence

Rysunek (a) pokazuje przykład składający się z pięciu słów w każdej grupie, gdzie odpowiedź brzmi „tak", ponieważ wybór słów do konkatenacji zgodnie z sekwencją 2, 1, 1, 4, 1, 5 z X lub Y daje to samo słowo , „aabbabbbababbaba". Z drugiej strony dane wejściowe opisane na rysunku (b), uzyskane z rysunku (a), po prostu usuwając pierwszą literę z pierwszego słowa każdej grupy, nie pozwalają na taki wybór, co można zweryfikować. Jego odpowiedź brzmi zatem „nie". Problem korespondencji słownej jest nierozstrzygnięty. Nie ma algorytmu, który mógłby odróżnić, ogólnie rzecz biorąc, elementy takie jak na rysunkach (a) i (b). Nieograniczony charakter problemu wynika z faktu, że liczba słów, które należy wybrać, aby uzyskać wspólne słowo złożone, nie jest ograniczona. Jednak i tutaj możemy wskazać na pozornie „mniej ograniczony" wariant, w którym wydaje się, że spraw do sprawdzenia jest więcej, ale który mimo wszystko jest rozstrzygalny. W nim dane wejściowe są takie jak poprzednio, ale nie ma ograniczeń co do sposobu dokonywania wyborów na podstawie X i Y; nawet liczba wybranych słów nie musi być taka sama. Pytamy po prostu, czy możliwe jest utworzenie wspólnego słowa złożonego przez połączenie kilku słów z X i niektórych słów z Y . Na rysunku (b), który dał podstawę do „nie" dla standardowej wersji problemu, na przykład słowo „baba" można uzyskać z X za pomocą sekwencji 3, 2, 2 i z Ys o 1, 2, a więc jest to „tak" w nowej wersji. Ten bardziej liberalny problem faktycznie dopuszcza szybki algorytm wielomianowy! Drugi problem dotyczy składni języków programowania. Załóżmy, że ktoś dostarczył nam reguły składni jakiegoś języka. Jeśli ktoś inny pojawi

się z innym zestawem reguł, możemy być zainteresowani dowiedzeniem się, czy te dwie definicje są równoważne, w tym sensie, że definiują ten sam język; to znaczy ta sama klasa składni instrukcji (lub programów). Problem ten ma znaczenie dla konstrukcji kompilatorów, ponieważ kompilatory, oprócz innych zadań, są również odpowiedzialne za rozpoznawanie poprawności składniowej ich programów wejściowych. W tym celu mają wbudowany zestaw reguł składniowych. Można sobie wyobrazić, że w celu zwiększenia wydajności kompilatora jego projektant chciałby zastąpić ten zbiór reguł bardziej zwartym zbiorem. Oczywiście ważne jest, aby z góry wiedzieć, że te dwa zestawy są wymienne. Ten problem również jest nierozstrzygnięty. Nie istnieje żaden algorytm, który po odczytaniu z danych wejściowych dwóch zbiorów reguł składniowych będzie w stanie w skończonym czasie decydować, czy definiują one dokładnie ten sam język.

Problemy z wyjściami nie są lepsze

Powinniśmy być może jeszcze raz podkreślić fakt, że wygoda techniczna jest jedynym powodem ograniczania się tutaj do problemów decyzyjnych. Każdy z opisanych przez nas nierozstrzygalnych problemów ma warianty, które wymagają danych wyjściowych i które również są nieobliczalne. Warianty trywialne to te, które są w zasadzie problemami decyzyjnymi w (dość przejrzystym) przebraniu. Przykładem jest problem polegający na tym, że dla danego zestawu kolorowych kafelków typu T należy wyprowadzić rozmiar najmniejszego obszaru, którego nie można kafelkować przez T, oraz 0, jeśli każdy skończony obszar można kafelkować. Jasne jest, że problem ten nie może być obliczony, ponieważ rozróżnienie w wyniku między 0 a wszystkimi innymi liczbami jest właśnie rozróżnieniem między „tak" i „nie" w pierwotnym zadaniu.

Bardziej wyrafinowane problemy znacznie lepiej ukrywają możliwość podjęcia nierozstrzygalnej decyzji. Poniższy problem również jest nieobliczalny. Aby to zdefiniować, powiedzmy, że skończona część siatki jest ograniczonym obszarem dla zestawu T płytek, jeśli można ją ułożyć zgodnie z prawem T , ale nie można jej w żaden sposób rozszerzyć o więcej płytek bez naruszania kafelkowania zasady. Teraz problem polega na znalezieniu określonych zbiorów T, które mają duże ograniczone pola. W szczególności problemowi algorytmicznemu przypisywana jest liczba N (która powinna wynosić co najmniej 2) i jest proszony o wyprowadzenie rozmiaru największego ograniczonego obszaru dla dowolnego zestawu płytek, który obejmuje nie więcej niż N kolorów. Powinieneś przekonać się, że dla każdego N > 1 ta liczba jest dobrze zdefiniowana. Nie jest oczywiste, że aby rozwiązać problem ograniczonego obszaru, potrzebujemy umiejętności decydowania o rodzaju problemu układania płytek. I tak, chociaż problem powoduje powstanie dobrze zdefiniowanej funkcji N typu nie-tak/nie, tej funkcji po prostu nie da się obliczyć algorytmicznie.

Algorytmiczna weryfikacja programu

W Części 5 zapytaliśmy, czy komputery mogą za nas weryfikować nasze programy. Oznacza to, że szukaliśmy automatycznego weryfikatora. W szczególności interesuje nas problem decyzyjny, którego danymi wejściowymi jest opis problemu algorytmicznego oraz tekst algorytmu lub programu, który, jak się uważa, rozwiązuje dany problem. Interesuje nas algorytmiczne ustalenie, czy dany algorytm rozwiązuje dany problem, czy nie. Innymi słowy, chcemy „tak", jeśli dla każdego z legalnych danych wejściowych problemu algorytm zakończy działanie, a jego wyniki będą dokładnie takie, jak określono w problemie, i chcemy „nie", jeśli istnieje chociaż jedno wejście, dla którego algorytm albo zawiedzie zakończyć lub zakończyć z niewłaściwymi wynikami. Zauważ, że problem wymaga algorytmu, który działa dla każdego wyboru pary problem/algorytm. Oczywiście problemu weryfikacji nie można omówić bez bardziej szczegółowego określenia dozwolonych danych wejściowych. Jaki język programowania ma być używany do kodowania algorytmów wejściowych? W jakim języku specyfikacji należy opisać wejściowe problemy algorytmiczne? Tak się składa, że nawet bardzo skromne wybory

tych języków sprawiają, że problem weryfikacji jest nierozstrzygnięty. Nawet jeśli dozwolone programy mogą manipulować tylko liczbami całkowitymi lub ciągami symboli i mogą wykonywać tylko podstawowe operacje dodawania lub dołączania symboli do ciągów, nie można ich zweryfikować algorytmicznie. Kandydaci na weryfikatory algorytmiczne mogą dobrze działać w przypadku wielu przykładowych danych wejściowych, ale ogólny problem jest nierozstrzygnięty, co oznacza, że zawsze będą istniały algorytmy, których weryfikator nie będzie w stanie zweryfikować. Jak omówiono w rozdziale 5, oznacza to daremność nadziei na oprogramowanie system, który byłby zdolny do automatycznej weryfikacji programu. Zmniejsza również nadzieję na optymalizację kompilatorów zdolnych do przekształcania programów w optymalnie wydajne. Taki kompilator może nawet ogólnie nie być w stanie stwierdzić, czy nowa wersja kandydująca rozwiązuje ten sam problem, co oryginalny program, nie mówiąc już o tym, czy jest bardziej wydajny. Co więcej, jest to nie tylko weryfikacja, czy program spełnia jego pełną wymaganą specyfikację, która jest nierozstrzygalna, ale nawet weryfikacja tylko niektórych jego części. Na przykład sprawdzenie, czy programy nie mają błędu z roku 2000, jest również generalnie niemożliwe. Kandydat na wykrywacz Y2K mógłby dobrze wykonywać swoją pracę w przypadku niektórych programów wejściowych i może być w stanie zweryfikować ograniczone rodzaje problemów Y2K, ale jako ogólne rozwiązanie problemu z pewnością zawiedzie. W ten sposób możemy zapomnieć o skomputeryzowanym rozwiązanie problemu Y2K lub inne tego typu szeroko zakrojone próby ustalenia naszych oczekiwań wobec oprogramowania za pomocą komputera. Idąc dalej, nie tylko pełna lub częściowa weryfikacja jest nierozstrzygalna, ale nie możemy nawet zdecydować, czy dany algorytm kończy się tylko na swoich legalnych danych wejściowych. Co więcej, nie jest nawet rozstrzygalne, czy algorytm kończy się na jednym danym wejściu! Te problemy wypowiedzenia zasługują na szczególną uwagę.

Problem z zatrzymaniem

Rozważ następujący algorytm A:

(1) podczas gdy $X \neq 1$ wykonaj następujące czynności: $X \leftarrow X - 2$;

(2) zatrzymaj się.

Innymi słowy, A zmniejsza wejściowe X o 2, aż X stanie się równe 1. Zakładając, że jego legalne dane wejściowe składają się z dodatnich liczb całkowitych <1, 2, 3, . . . , > jest całkiem oczywiste, że A zatrzymuje się dokładnie dla nieparzystych danych wejściowych. Liczba parzysta będzie wielokrotnie zmniejszana o 2, aż osiągnie 2, a następnie „ominie" 1, przechodząc w nieskończoność przez 0,-2,-4,-6 i tak dalej. Dlatego w przypadku tego konkretnego algorytmu decydowanie, czy legalne dane wejściowe spowodują jego zakończenie, jest trywialne: wystarczy sprawdzić, czy dane wejściowe są nieparzyste, czy parzyste i odpowiednio odpowiedzieć. Oto kolejny, podobnie wyglądający algorytm B:

(1) podczas gdy $X \neq 1$ wykonaj następujące czynności:

(1.1) jeśli X jest parzyste zrób $X \leftarrow X/2$;

(1.2) w przeciwnym razie (X jest nieparzyste) wykonaj $X \leftarrow 3X + 1$;

(2) zatrzymaj się.

Algorytm B wielokrotnie zmniejsza o połowę X, jeśli jest parzysty, ale zwiększa go ponad trzykrotnie, jeśli jest nieparzysty. Na przykład, jeśli B jest uruchamiany na 7, kolejność wartości to: 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, ostatecznie skutkujące wypowiedzeniem. Właściwie, jeśli spróbujemy uruchomić algorytm na dowolnej dodatniej liczbie całkowitej, przekonamy się, że się kończy. Sekwencja wartości jest często dość nietypowa, osiąga zaskakująco wysokie wartości i zmienia

się w nieprzewidywalny sposób, zanim osiągnie 1. Rzeczywiście, przez lata B był testowany na wielu wejściach i zawsze był przerywany. Niemniej jednak nikt nie był w stanie udowodnić, że kończy się dla wszystkich dodatnich liczb całkowitych, chociaż większość ludzi uważa, że tak. Pytanie, czy tak jest, czy nie, jest w rzeczywistości trudnym otwartym problemem w dziedzinie matematyki znanej jako teoria liczb. Teraz, jeśli rzeczywiście B (lub jakikolwiek inny program) kończy działanie dla wszystkich swoich danych wejściowych, istnieje dowód tego faktu, jak omówiono w rozdziale 5, ale dla B nikt jeszcze takiego dowodu nie znalazł. Te przykłady ilustrują, jak trudno jest analizować właściwości terminacji nawet bardzo prostych algorytmów. Zdefiniujmy teraz konkretną wersję problemu terminacji algorytmicznej, zwanego problemem zatrzymania. Definiujemy to tutaj w terminach naszego uzgodnionego języka programowania wysokiego poziomu L. Problem ma dwa wejścia: tekst legalnego programu R w języku L i potencjalne wejście X do R.1 Problem zatrzymania pyta, czy R zakończyłby się, gdybyśmy uruchomili go na wejściu X, co oznaczamy przez R(X) ↓. Przypadek R nie kończącego się lub rozchodzącego się na X oznaczamy R(X)↑ .Jak już wspomniano, problem zatrzymania jest nierozstrzygnięty, co oznacza, że nie ma sposobu, aby w skończonym czasie stwierdzić, czy dane R zakończy się na danym X. W interesie rozwiązania tego problemu, kuszące jest zaproponowanie algorytmu, który po prostu uruchomi R na X i zobaczy, co się stanie. Cóż, jeśli i kiedy wykonanie się zakończy, możemy słusznie stwierdzić, że odpowiedź brzmi „tak". Trudność polega na podjęciu decyzji, kiedy przestać czekać i powiedzieć „nie". Nie możemy po prostu zrezygnować w pewnym momencie i dojść do wniosku, że skoro R nie wygasło do tej pory, to nigdy się nie stanie. Być może gdybyśmy zostawili R tylko trochę dłużej, to by się skończyło. Dlatego uruchomienie R na X nie wykonuje zadania i, jak wspomniano, nic nie może wykonać zadania, ponieważ problem jest nierozstrzygnięty.

Nic w obliczeniach nie jest obliczalne!

Fakt, że weryfikacja i wstrzymanie są nierozstrzygalne, jest tylko małą częścią znacznie bardziej ogólnego zjawiska, które w rzeczywistości jest znacznie głębsze i dużo bardziej niszczycielskie. Istnieje niezwykły wynik, zwany twierdzeniem Rice'a, który pokazuje, że nie tylko nie możemy zweryfikować programów ani określić ich stanu wstrzymania, ale tak naprawdę nie możemy nic na ich temat dowiedzieć się. Żaden algorytm nie może decydować o żadnej nietrywialnej właściwości obliczeń. Dokładniej, powiedzmy, że interesuje nas decydowanie o jakiejś właściwości programów, która jest (1) prawdziwa dla niektórych programów, ale nie dla innych, oraz (2) niewrażliwa na składnię programu i sposób jego działania lub algorytm; to znaczy, jest właściwością tego, co robi program, problemu, który rozwiązuje, a nie szczególnej formy, jaką przybiera rozwiązanie. Na przykład, możemy chcieć wiedzieć, czy program kiedykolwiek wypisuje „tak", czy zawsze generuje liczby, czy jest odpowiednikiem jakiegoś innego programu itp. itd. Twierdzenie Rice'a mówi nam, że żadna taka właściwość programów nie może być zdecydowany. Ani jeden. Wszystkie są nierozstrzygalne. Naprawdę możemy zapomnieć o możliwości automatycznego wnioskowania o programach lub wydedukowania rzeczy na temat tego, co robią nasze programy. Dzieje się tak bez względu na to, czy nasze programy są małe czy duże, proste czy złożone, czy też chcemy się dowiedzieć, czy jest to ogólna właściwość, czy coś błahego i idiosynkratycznego. Praktycznie nic w obliczeniach nie jest obliczalne! A co powiesz na to?

Udowodnienie nierozstrzygalności

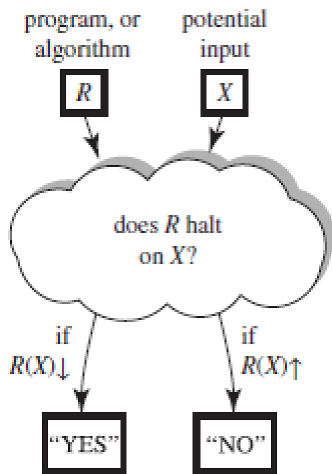
Jak udowodnić, że problem P jest nierozstrzygnięty? Jak ustalić fakt, że nie istnieje żaden algorytm do rozwiązania P, bez względu na to, jak sprytny jest projektant algorytmu? Sytuacja ta jest podobna do opisanej w Części 7 dla problemów NP-zupełnych. Najpierw musi być jeden problem początkowy, którego nierozstrzygalność ustala się jakąś bezpośrednią metodą. W przypadku nierozstrzygalności rolę tę odgrywa problem zatrzymania, który później okaże się nierozstrzygalny. Gdy pojawi się taki pierwszy nierozstrzygalny problem, nierozstrzygalność innych problemów ustala się przez wykazanie redukcji problemów, o których już wiadomo, że są nierozstrzygalne w stosunku do danych problemów.

Różnica polega na tym, że tutaj redukcja z problemu P do problemu Q niekoniecznie musi być ograniczona; może to zająć dowolną ilość czasu lub miejsca w pamięci. Wystarczy, że istnieje algorytmiczny sposób przekształcenia wejścia P w wejście Q, w taki sposób, że odpowiedź P tak/nie na dane wejściowe jest dokładnie odpowiedzią Q na przekształcone dane wejściowe. W ten sposób, jeśli wiadomo już, że P jest nierozstrzygalne, Q również musi być nierozstrzygalne. Powodem jest to, że w przeciwnym razie moglibyśmy rozwiązać P za pomocą algorytmu, który wziąłby dowolne dane wejściowe, przekształciłby je w dane wejściowe dla Q i poprosił algorytm Q o odpowiedź. Taki hipotetyczny algorytm dla Q nazywa się wyrocznią, a redukcję można traktować jako pokazanie, że P jest rozstrzygalne, biorąc pod uwagę wyrocznię do decydowania o Q. Pod względem rozstrzygalności pokazuje to, że P nie może być lepsze niż Q. Na przykład stosunkowo łatwo jest sprowadzić problem zatrzymania do problemu weryfikacji. Załóżmy, że problem zatrzymania jest nierozstrzygnięty (w rzeczywistości udowodnimy to bezpośrednio w następnej sekcji). Aby pokazać, że problem weryfikacji również jest nierozstrzygnięty, musimy pokazać, że wyrocznia weryfikacyjna umożliwiłaby nam również rozstrzygnięcie problemu zatrzymania. Cóż, mając algorytm R i jeden z jego potencjalnych wejść X, przekształcamy parę <R, X> , która jest danymi wejściowymi do problemu zatrzymania, na parę <P, R>, która jest danymi wejściowymi do problemu weryfikacji. Algorytm R pozostaje ten sam, a problem algorytmiczny P jest opisany przez określenie, że X jest jedynym dozwolonym wejściem do R, a wyjście dla tego jednego wejścia jest nieistotne. Powiedzieć, że R jest (całkowicie) poprawny w odniesieniu do tego dość uproszczonego problemu P, to po prostu powiedzieć, że R kończy się na wszystkich dozwolonych danych wejściowych (tj. na X) i wytwarza pewne dane wyjściowe, co w rzeczywistości jest po prostu stwierdzeniem, że R kończy się na X. Innymi słowy, problem weryfikacji mówi „tak” do <P, R> wtedy i tylko wtedy, gdy problem zatrzymania mówi „tak” dla <;R, X> . W związku z tym problem weryfikacji jest rozstrzygalny, ponieważ w przeciwnym razie moglibyśmy rozwiązać problem zatrzymania, konstruując algorytm, który najpierw przekształciłby dowolne <R, X>w odpowiednie <P, R> , a następnie wykorzystałby wyrocznię weryfikacyjną do określenia poprawności <P, R> .

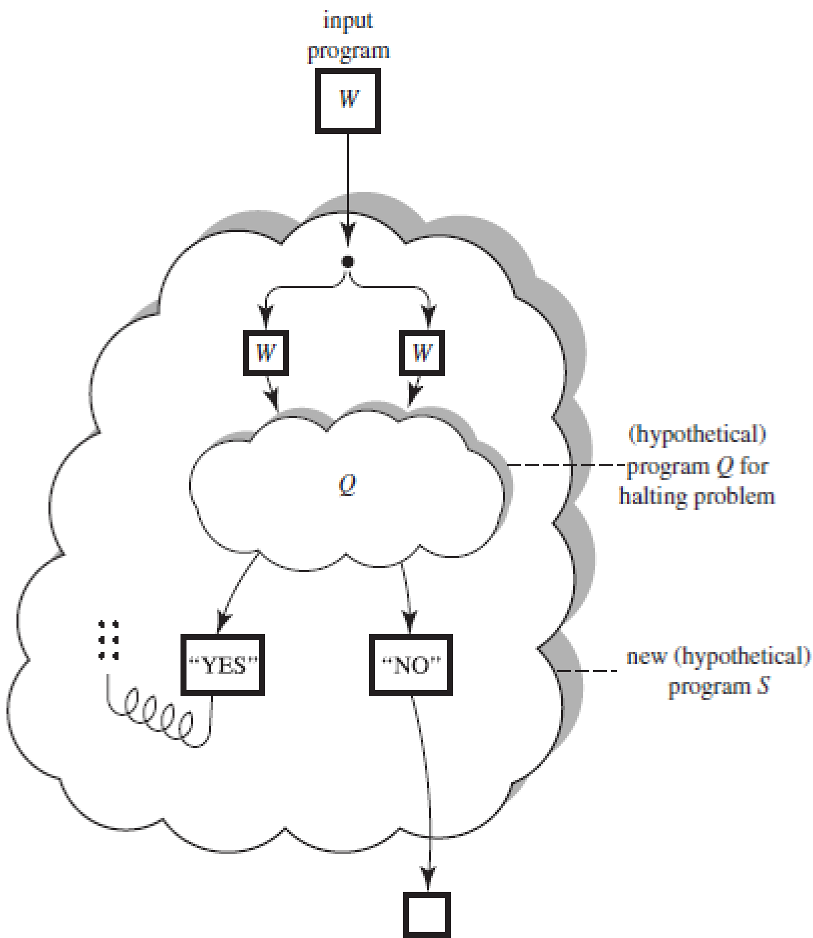
Inne redukcje są znacznie bardziej subtelne. Co, u licha, mają problemy z kafelkowaniem, które mają wspólnego z zakończeniem algorytmicznym? Jak zredukować węże domina do dwukierunkowych formacji słów? Pomimo widocznych różnic, wszystkie te problemy są ze sobą ściśle powiązane, ponieważ są wzajemnie redukowalne

Udowodnienie nierozstrzygalności problemu zatrzymania

Udowodnimy teraz, że problem zatrzymania, opisany na rysunku



, jest nierozstrzygnięty. Jak wyjaśniono wcześniej, odbywa się to bezpośrednio, a nie przez redukcję jakiegoś innego problemu. Teraz, wierni naszej konwencji terminologicznej, tak naprawdę musimy pokazać, że nie ma programu w uzgodnionym języku programowania wysokiego poziomu L, który rozwiązuje problem zatrzymania programów w L. Dokładniej, chcemy udowodnić następujące twierdzenie: Nie ma programu w L, który po zaakceptowaniu dowolnej pary <R, X> , składającej się z tekstu legalny program R w L i łańcuchu symboli X, kończy się po pewnym skończonym czasie i wyprowadza „tak", jeśli R zatrzymuje się po uruchomieniu na wejściu X i „nie", jeśli R nie zatrzymuje się po uruchomieniu na wejściu X. program, jeśli istnieje, sam jest tylko jakimś legalnym programem w L; może wykorzystywać tyle miejsca w pamięci i tyle czasu, ile zażąda, ale musi działać, zgodnie z opisem, dla każdej pary <R, X> . Udowodnimy, że program spełniający te wymagania nie istnieje, przez sprzeczność. Innymi słowy, założymy, że taki program istnieje, nazwiemy go Q i wyprowadzimy z tego założenia całkowitą sprzeczność. Przez cały czas powinieneś być ostrożny, upewniając się, że wszystko, co robimy, jest legalne i zgodne z zasadami, tak że gdy sprzeczność stanie się oczywista, będziemy usprawiedliwieni wskazując na założenie o istnieniu Q jako winowajcy. Skonstruujmy teraz nowy program w L, nazwijmy go S, jak pokazano schematycznie na Rysunek



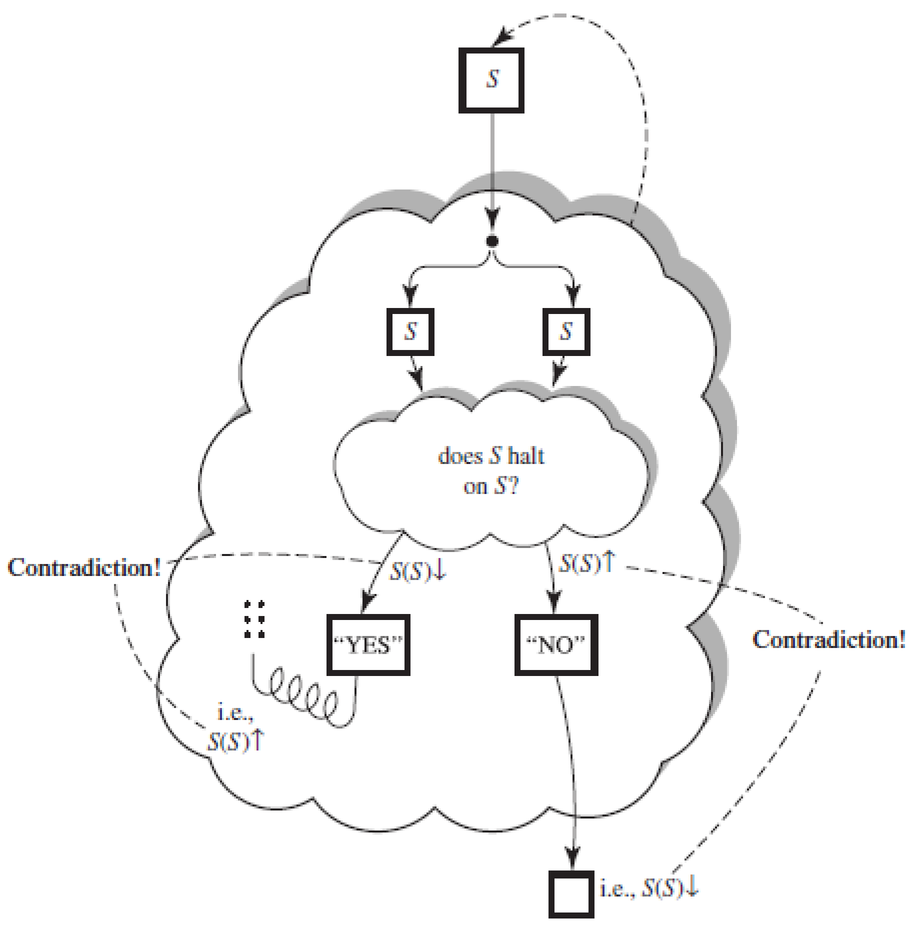
Ten program ma jedno wejście, które jest legalnym programem W w L. Po odczytaniu wejścia S tworzy jego kolejną kopię. To kopiowanie jest oczywiście możliwe w dowolnym języku programowania wysokiego poziomu, przy wystarczającej ilości pamięci. Przypominając, że (zakładając, że istnieje) program Q oczekuje dwóch danych wejściowych, z których pierwszym jest program, następną rzeczą, którą robi S, jest aktywacja Q na parze wejściowej składającej się z dwóch kopii W. jest rzeczywiście programem, jak oczekiwał Q, a drugi jest uważany za ciąg wejściowy, chociaż ten ciąg po prostu jest

tekstem tego samego programu, W. Tę aktywację Q można przeprowadzić, wywołując Q jako podprogramu z parametrami W i W, lub przez wstawienie tekstu (zakłada się, że istnieje) Q we właściwym miejscu i przypisanie wartości W i W do jego oczekiwanych zmiennych wejściowych, odpowiednio, R i X. Program S czeka teraz, aż ta aktywacja Q zakończy się, lub, używając metaforycznych terminów użytych wcześniej w książce, procesor S Runaround$_S$ czeka, aż procesor Q Runaround$_Q$ zgłosi się z powrotem do centrali. Chodzi o to, że zgodnie z naszym założeniem Q musi się zakończyć, ponieważ, jak wyjaśniono, jego pierwszym wejściem jest legalny program w L, a jego drugi, gdy jest traktowany jako ciąg symboli, jest całkowicie akceptowalnym potencjalnym wejściem do W. 2 I tak , według naszej hipotezy, Q musi w końcu zakończyć, mówiąc „tak" lub „nie". Teraz instruujemy nowy program S, aby zareagował na zakończenie Q w następujący sposób. Jeśli Q mówi „tak", S ma natychmiast wejść w narzuconą sobie pętlę nieskończoną, a jeśli powie „nie", S ma natychmiast zakończyć (wyjście jest nieistotne). Można to również osiągnąć w dowolnym języku wysokiego poziomu, np.:

…

(17) jeśli OUT = „yes" to przejdź do (17), w przeciwnym razie zatrzymaj się;

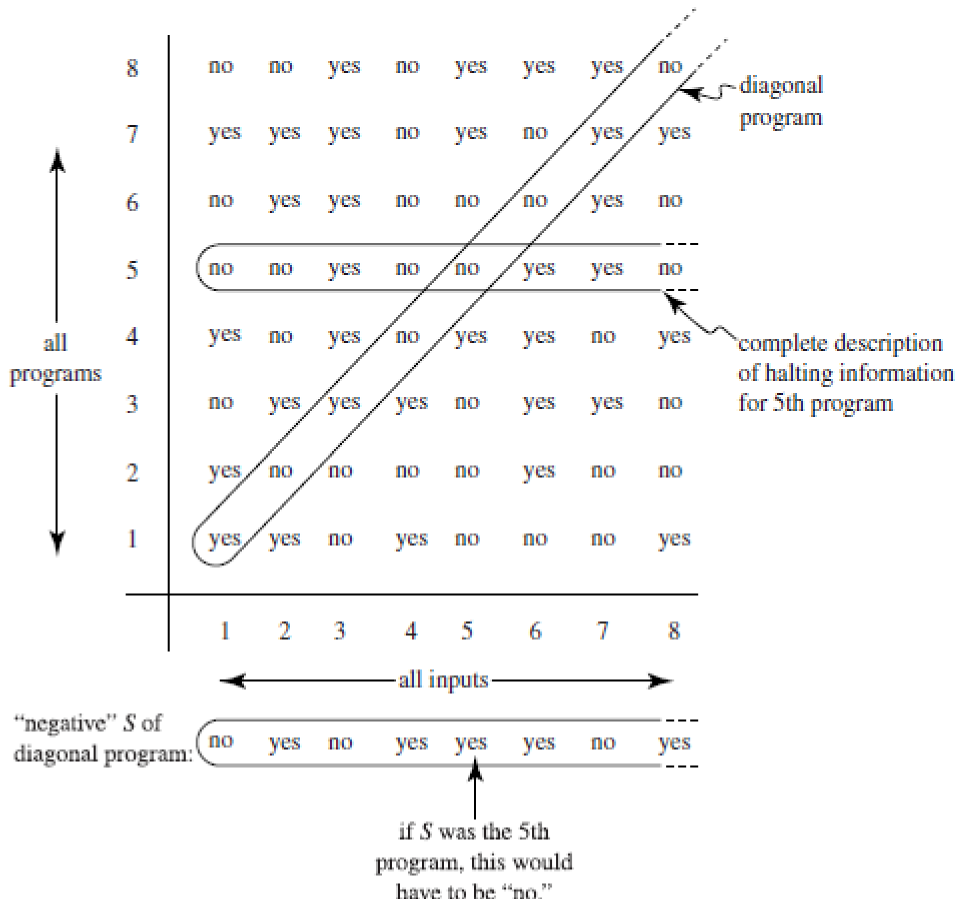
gdzie OUT jest zmienną zawierającą wyjście Q. Na tym kończy się konstrukcja (dziwnie wyglądającego) programu S, który, co należy podkreślić, jest programem legalnym w języku L, zakładając oczywiście, że Q jest. Teraz chcemy pokazać, że z S jest coś nie tak. Samo założenie, że S można skonstruować, jest logiczną niemożliwością. Ujawniając tę niemożliwość, będziemy polegać na oczywistym fakcie, że dla każdego wyboru legalnego programu wejściowego W nowy program S musi albo zakończyć się, albo nie. Pokażemy jednak, że istnieje pewien program wejściowy, dla którego S nie może zakończyć, ale też nie może się nie zakończyć! Jest to wyraźnie logiczna sprzeczność i użyjemy jej, aby wywnioskować, że nasze założenie o istnieniu Q jest błędne, dowodząc tym samym, że problem zatrzymania jest rzeczywiście nierozstrzygnięty. Programem wejściowym W, który powoduje tę niemożność, jest sam S. Aby zobaczyć, dlaczego S jako wejście do samego siebie powoduje sprzeczność, załóżmy na chwilę, że S, gdy otrzyma jako wejście własny tekst, kończy się. Przeanalizujmy teraz szczegóły tego, co naprawdę dzieje się z S, gdy jako dane wejściowe otrzymamy własny tekst. Najpierw tworzone są dwie kopie danych wejściowych S,



a następnie są one wprowadzane do programu (zakłada się, że istnieje) Q. Teraz, zgodnie z naszą hipotezą, Q musi zakończyć się po pewnym skończonym czasie, z odpowiedzią na pytanie, czy pierwsze wejście kończy się na drugim. Teraz, ponieważ Q pracuje obecnie na danych wejściowych S i S i ponieważ założyliśmy, że S rzeczywiście kończy się na S, musi się w końcu zatrzymać i powiedzieć „tak". Jednak po zakończeniu i powiedzeniu „tak", egzekucja wchodzi w nałożoną przez siebie nieskończona pętla i nigdy się nie kończy. Ale to oznacza, że przy założeniu, że S zastosowane do S kończy się (założenie, które spowodowało, że Q powiedział „tak"), odkryliśmy, że S zastosowane do S nie kończy się! W ten sposób dochodzimy do wniosku, że niemożliwe jest, aby S kończy się na S To pozostawia nam jedną pozostałą możliwość, a mianowicie, że S zastosowane do S nie wygasa. Jednak, jak można łatwo zweryfikować za pomocą rysunku 8.8, założenie to prowadzi w bardzo podobny sposób do wniosku, że S zastosowane do S rzeczywiście się kończy, ponieważ kiedy Q mówi „nie" (a powie „nie" z powodu nasze założenie) wykonanie S zastosowane wobec S natychmiast się kończy. W związku z tym niemożliwe jest również, aby S nie zakończył działania po uruchomieniu na S. Innymi słowy, program S nie może zakończyć się, gdy zostanie uruchomiony na sobie, i nie może się zakończyć! W konsekwencji coś jest bardzo nie tak z samym S. Ponieważ jednak wszystkie inne części S zostały skonstruowane całkiem legalnie, jedyną częścią, za którą można ponosić odpowiedzialność, jest program Q, którego zakładane istnienie umożliwiło nam skonstruowanie S w taki sposób, w jaki to zrobiliśmy. Wniosek jest taki, że program Q, rozwiązujący problem zatrzymania zgodnie z wymaganiami, po prostu nie może istnieć.

Metoda Diagonalizacji

Niektórzy ludzie czują się trochę nieswojo z przedstawionym właśnie dowodem i postrzegają go jako raczej okrągły. Jest to jednak rygorystyczny dowód matematyczny. Moglibyśmy pokusić się o zaproponowanie, aby dziwne programy samoodnoszące się, takie jak te, których zachowanie jest wykorzystywane w dowodzie, zostały w jakiś sposób zakazane. Być może w ten sposób problem zatrzymania „dobrze wychowanych” programów będzie rozstrzygalny. Cóż, wszystko, co można powiedzieć, to to, że dowód bardzo dobrze wytrzymuje wszelkie tego typu próby, aw rzeczywistości samoodnoszący się charakter argumentu można całkowicie ukryć przed obserwatorem. Najlepszym tego dowodem jest fakt, że inne nierozstrzygalne problemy są jedynie zamaskowanymi wersjami problemu zatrzymania, chociaż nie wydają się mieć nic wspólnego z programami, które odnoszą się do siebie. Można na przykład wykazać, że problem kafelkowania koduje problem zatrzymania całkiem bezpośrednio, co zilustrowano w Części 9. W rzeczywistości zasadnicza natura tego dowodu jest ucieleśniona w fundamentalnej technice dowodowej, która sięga czasów Georga Cantora. , wybitny dziewiętnastowieczny matematyk, który używał go w kontekście niealgorytmicznym. Ta technika, zwana diagonalizacją, jest używana w wielu innych dowodach dolnych granic algorytmiki. Spróbujmy teraz przeformułować dowód, że problem zatrzymania jest nierozstrzygalny, używając idei diagonalizacji. Powinieneś odnieść się do rysunku w procesie.



Dowód można zwizualizować, wyobrażając sobie nieskończoną tabelę, w której wykreśliliśmy wszystkie programy w naszym uzgodnionym języku programowania L ze wszystkimi możliwymi danymi wejściowymi. (Pomocne jest wyobrażenie sobie, że dane wejściowe są po prostu liczbami całkowitymi.) Programy są wymienione na rysunku pionowo po lewej stronie, a dane wejściowe poziomo na dole. Na skrzyżowaniu I rzędu i J kolumny na rysunku wskazaliśmy, czy I program zatrzymuje się na J, czy nie. W ten sposób cały I wiersz na rysunku jest kompletnym, aczkolwiek nieskończonym opisem informacji o zatrzymaniu dla I programu w L. Teraz konstruujemy nowy wyimaginowany program, który okaże się

zamaskowaną wersją wcześniej skonstruowany program S; dlatego nazwijmy to S również tutaj. Zatrzymanie S jest „ujemną” linią ukośną w tabeli na rysunku. Innymi słowy, S jest skonstruowane tak, że po uruchomieniu na dowolnym wejściu J zatrzymuje się tylko wtedy, gdy J-ty program nie zatrzymałby się na J i nie zatrzymuje się, jeśli J-ty program zatrzymałby się na J . Biorąc pod uwagę tę konfigurację, łatwo jest argumentować, że problem zatrzymania jest nierozstrzygnięty. Załóżmy, że jest rozstrzygalny, co oznacza, że możemy za pomocą programu Q zdecydować, czy dany program w L zatrzymuje się na danym wejściu. Oznacza to, że wyimaginowane S mogłoby być faktycznie napisane w języku L: biorąc pod uwagę J jako dane wejściowe, przeszłoby ono listę programów w L, znalazłoby J, a następnie przesłałoby ten program i dane wejściowe J do założony program Q dla problemu zatrzymania. Samo S przystąpiłoby następnie do zatrzymania lub wejścia w nałożoną przez siebie nieskończoną pętlę, w zależności od wyniku biegu Q, jak opisano wcześniej: pierwsza, jeśli Q powie „nie”, a druga, jeśli powie „tak”. Jak bez wątpienia widać, S rzeczywiście zachowuje się jak ujemna przekątna stołu. Prowadzi to jednak do sprzeczności, ponieważ jeśli S jest programem w L, to również musi być jednym z programów na liście na rysunku 8.9, ponieważ ta lista zawiera wszystkie programy w L. Ale nie może: jeśli S jest , powiedzmy, piąty program na liście nie może zatrzymać się na wejściu 5, ponieważ w tabeli na skrzyżowaniu piątego wiersza i piątej kolumny znajduje się „nie”. Jednak przez swoją konstrukcję S zatrzymuje się na wejściu 5 i jest to sprzeczność. Dowód taki jak ten można zwięźle opisać, mówiąc, że dokonaliśmy diagonalizacji wszystkich programów i wszystkich danych wejściowych, konstruując S jako ujemną lub przeciwną przekątną. Sprzeczność wynika zatem z niemożliwości, aby ten S był jednym z programów na liście.

Certyfikaty skończone dla nierozstrzygalnych problemów

W Części 7 widzieliśmy, że pewne problemy, które nie mają znanych rozwiązań w czasie wielomianowym, powodują jednak powstanie certyfikatów o rozmiarze wielomianowym dla danych wejściowych, które dają odpowiedź „tak”. Znalezienie takiego certyfikatu może zająć dużo czasu, ale po znalezieniu można go łatwo sprawdzić, aby był ważnym dowodem na to, że odpowiedź rzeczywiście brzmi „tak”. W kontekście niniejszej części mamy do czynienia z podobnym zjawiskiem certyfikatów, ale bez wymogu, aby certyfikaty były wielomianowe; muszą być skończone, ale mogą być nierozsądnie długie. Tak jak problemy w NP mają certyfikaty rozsądnej wielkości, sprawdzalne w rozsądnym czasie, mimo że nie wiadomo, czy dopuszczają rozsądne algorytmy, tak niektóre nierozstrzygalne problemy mają certyfikaty skończone, sprawdzalne w skończonym czasie, mimo że nie mają skończonych algorytmów. W rzeczywistości większość opisanych do tej pory nierozstrzygalnych problemów rzeczywiście dopuszcza ograniczone certyfikaty. Na przykład, aby przekonać kogoś, że niektóre dane wejściowe dają odpowiedź „tak” na problem korespondencji słów, możemy po prostu przedstawić skończoną sekwencję wskaźników, która daje początek temu samemu słowu, gdy jest utworzona z X lub Y. Co więcej, możemy łatwo sprawdzić ważność tego (być może bardzo długiego, ale skończonego) świadectwa, łącząc Xy określone przez indeksy, a następnie Y w ten sam sposób i weryfikując, czy oba z nich dają jeden i ten sam słowo. Jeśli chodzi o problem zatrzymania, aby przekonać kogoś, że program R zatrzymuje się na X, możemy po prostu pokazać sekwencję wartości wszystkich zmiennych i struktur danych, które stanowią kompletny ślad legalnego, skończonego, kończącego się wykonania R na X. może następnie sprawdzić ważność certyfikatu, symulując działanie R na X, porównując wartości osiągnięte na każdym kroku z wartościami w danej sekwencji i upewniając się, że program rzeczywiście kończy się na końcu sekwencji. Podobnie łatwo zauważyć, że wąż domina prowadzący z punktu V do punktu W jest doskonale dobrym i sprawdzalnym skończonym świadectwem tego, że odpowiednie dane wejściowe do problemu węża domina dają „tak”. Jeśli chodzi o zwykły problem kafelkowania, tutaj istnieją certyfikaty dla wejść „nie”, a nie dla wejść „tak”. Jeśli zestaw typów kafelków T nie może ułożyć wszystkich skończonych obszarów wszystkich rozmiarów, musi istnieć obszar, którego T w żaden sposób nie może ułożyć na kafelkach. Certyfikat pokazujący, że T daje „nie”, będzie po prostu tym

obszarem do zjedzenia. Aby sprawdzić, czy ten obszar, nazwijmy go E, jest rzeczywiście certyfikatem, musimy sprawdzić, czy T nie może rozmieścić E. Ponieważ zarówno T, jak i E są skończone, jest tylko skończenie wiele (choć nieuzasadniona liczba) możliwych do wypróbowania płytek , a zatem sprawdzanie może być przeprowadzane algorytmicznie w skończonym czasie. W tym sensie jest to problem braku kafelków, który jest porównywalny z innymi; mianowicie problem, który pyta, czy nie jest tak, że T może rozłożyć wszystkie obszary.

Problemy z certyfikatami dwukierunkowymi są rozstrzygane

Zauważ, że każdy z tych problemów ma certyfikaty tylko dla jednego z kierunków, albo na „tak" albo na „nie". Istnienie certyfikatu dla jednego kierunku nie przeczy nierozstrzygalności problemu, ponieważ nie wiedząc z wyprzedzeniem, czy dane wejściowe to „tak" czy „nie", nie możemy wykorzystać istnienia certyfikatu dla jednego z te, aby pomóc znaleźć algorytm dla problemu. Powodem jest to, że jakakolwiek próba sprawdzenia wszystkich kandydujących certyfikatów i sprawdzenia każdego pod kątem ważności nie zakończy się, jeśli dane wejściowe są niewłaściwego rodzaju, ponieważ w ogóle nie ma certyfikatu, a poszukiwanie jednego nigdy się nie skończy. Ten punkt był domyślnie obecny, gdy wyjaśniono wcześniej, dlaczego symulacja danego programu na danym wejściu nie może służyć do rozwiązania problemu zatrzymania. Ciekawym pytaniem jest, czy nierozstrzygalny problem może mieć certyfikaty zarówno dla „tak", jak i „nie". Cóż, nie może, ponieważ jeśli certyfikaty istnieją dla obu kierunków, można je wykorzystać do stworzenia algorytmu problemu, czyniąc to rozstrzygalnym. Aby zobaczyć jak, załóżmy, że mamy problem decyzyjny, dla którego każdy wkład prawny ma skończony certyfikat. Wejścia „tak" mają certyfikaty jednego rodzaju (nazwijmy je certyfikatami tak), a wejścia „nie" mają certyfikaty innego rodzaju (brak certyfikatów). Jeden rodzaj może składać się ze skończonych sekwencji liczb, inny z pewnych skończonych obszarów siatki liczb całkowitych i tak dalej. Załóżmy dalej, że świadectwa obu rodzajów można zweryfikować jako takie w określonym czasie. Aby zdecydować, czy problem odpowiada „tak" lub „nie" na dane wejściowe, możemy przejść systematycznie przez wszystkie certyfikaty kandydatów, naprzemiennie z tymi z dwóch rodzajów. Rozważamy zatem najpierw certyfikat tak, potem certyfikat nie, potem kolejny certyfikat tak i tak dalej, nie tracąc ani jednego . W rzeczywistości jednocześnie próbujemy znaleźć certyfikat tak lub certyfikat nie, cokolwiek, co posłuży do przekonania nas o statusie problemu na danym wejściu. Kluczowym faktem jest to, że proces ten jest gwarantowany do zakończenia, ponieważ każde wejście na pewno ma certyfikat tak lub nie, i cokolwiek to jest, prędzej czy później go znajdziemy. Po znalezieniu cały proces można zakończyć, tworząc „tak" lub „nie", w zależności od rodzaju wykrytego certyfikatu. Tak więc problemy nierozstrzygalne nie mogą mieć certyfikatów dwustronnych bez zaprzeczania ich własnej nierozstrzygalności. Problemy nierozstrzygalne, które mają certyfikaty jednokierunkowe, takie jak te, które opisaliśmy, są czasami określane jako częściowo rozstrzygalne, ponieważ dopuszczają algorytmy, które, jak można powiedzieć, zbliżają się do połowy rozwiązania. Po zastosowaniu do danych wejściowych taki algorytm gwarantuje zakończenie i powiedzenie „tak", jeśli „tak" jest rzeczywiście odpowiedzią, ale może nie zakończyć się w ogóle, jeśli odpowiedź brzmi „nie". Tak jest w przypadku osób posiadających certyfikaty „tak"; algorytm sprawdza je wszystkie, próbując znaleźć certyfikat potwierdzający „tak" danych wejściowych. W przypadku problemów, których certyfikaty są typu „nie", takich jak problem kafelkowania, role „tak" i „nie" są przełączane.

Problemy, które są jeszcze mniej rozstrzygające!

Jak się okazuje, wszystkie częściowo rozstrzygalne problemy, w tym problem zatrzymania, problem węża domina, problem korespondencji słów i problem kafelkowania (właściwie problem nieukładania, w którym chcemy „tak", jeśli płytki nie mogą wszystkie obszary) są równoważne obliczeniowo, co oznacza, że każdy z nich można skutecznie zredukować do każdego z pozostałych. W konsekwencji, chociaż wszystkie są nierozstrzygalne, każdy z nich może być rozstrzygnięty za pomocą

wyimaginowanego podprogramu lub wyroczni dla dowolnego z pozostałych: gdybyśmy mogli zdecydować, czy dany program zatrzymuje się na danym wejściu, moglibyśmy również zdecyduj, czy dany zestaw płytek może ułożyć siatkę liczb całkowitych obok siebie i czy możemy utworzyć wspólne słowo, łącząc odpowiadające im słowa X i Y, i na odwrót. Znowu sytuacja jest podobna do tej w przypadku problemów NP-zupełnych, z tym wyjątkiem, że tutaj znamy dokładny status problemów w klasie, podczas gdy tam nie. Wszystkie problemy NP-zupełne są wielomianowo równoważne, ponieważ istnieją wielomianowe redukcje od każdego z nich do wszystkich pozostałych. Z drugiej strony, częściowo rozstrzygalne problemy są jedynie algorytmicznie równoważne, bez ograniczeń co do czasu, jaki może zająć redukcje. Teraz, tak jak istnieją rozstrzygalne problemy, które są nawet gorsze niż te NPkompletne, tak też istnieją nierozstrzygalne problemy, które są jeszcze gorsze niż częściowo rozstrzygalne. Widzieliśmy wcześniej, że zatrzymanie sprowadza się do weryfikacji. Odwrotność nie jest prawdą. Można udowodnić, że nawet jeśli mamy hipotetyczną wyrocznię dla problemu zatrzymania, nie możemy algorytmicznie zweryfikować programów, ani rozwiązać problemu całościowego, który pyta, czy program zatrzymuje się na wszystkich swoich legalnych danych wejściowych. Wynika z tego, że problemy z weryfikacją i całością są jeszcze gorsze niż problem zatrzymania; są, by tak rzec, „mniej rozstrzygalne”. Oznacza to między innymi, że na przykład problem totalności nie ma żadnych certyfikatów. Jest to w rzeczywistości dość intuicyjne, ponieważ wydaje się, że nie ma możliwości skończonego sprawdzania, aby udowodnić, że program zatrzymuje się na wszystkich nieskończenie wielu danych wejściowych lub że nie zatrzymuje się (tzn. pociąga za sobą nieskończone obliczenia) na co najmniej jednym z ich. Pierwsze z tych twierdzeń może wydawać się sprzeczne ze stwierdzeniem zawartym w części 5, że każdy poprawny program rzeczywiście ma (skończony) dowód, dowodzący w szczególności, że program zatrzymuje się na wszystkich swoich wejściach. Jednakże, chociaż gwarantuje się istnienie skończonego dowodu zakończenia dla wszystkich danych wejściowych, nie może on kwalifikować się jako certyfikat, ponieważ niekoniecznie jest sprawdzalny algorytmicznie. W rzeczywistości twierdzenia logiczne, których prawdziwość musimy ustalić, aby zweryfikować słuszność dowodu, są formułami formalizmu logicznego, który sam w sobie jest nierozstrzygalny! Oto kolejny przykład. Przypomnijmy formalizm arytmetyki Presburgera, która pozwala nam mówić o dodatnich liczbach całkowitych z „+” i „=”. Problem określania prawdziwości formuł w arytmetyce Presburgera jest rozstrzygalny, ale, jak stwierdzono w części 7, ma podwójne wykładnicze ograniczenie dolne. Co zaskakujące, jeśli do formalizmu dodamy operator mnożenia „×”, uzyskując logikę zwaną arytmetyką pierwszego rzędu, problem staje się nierozstrzygnięty. Co więcej, nie jest ona nawet częściowo rozstrzygalna, więc jej status jest bardziej zbliżony do weryfikacji i totalności niż do zatrzymania, korespondencji słownej, węży domina czy kafelkowania. Właściwie jest jeszcze gorzej niż te.

Wysoce nierozstrzygnięte problemy

Tak jak problemy rozstrzygalne można pogrupować w różne klasy złożoności, tak samo problemy nierozstrzygalne można pogrupować w poziomy lub stopnie nierozstrzygalności. Są takie, które są częściowo rozstrzygalne lub, można powiedzieć, prawie rozstrzygalne i wszystkie są równoważne obliczeniowo, a są też takie, które są gorsze. Jednak wiele gorszych problemów nie jest między sobą równorzędne pod względem obliczeniowym. W rzeczywistości istnieje nieskończona ilość hierarchii nierozstrzygalnych problemów, z których każdy poziom zawiera problemy gorsze od wszystkich znajdujących się na niższych poziomach. Oprócz niskiego poziomu nierozstrzygalności, czyli częściowej rozstrzygalności, istnieje jeszcze inny, szczególnie naturalny i znaczący poziom, który okazuje się znacznie wyższy. Nie wdając się w zbyt wiele szczegółów, nazwiemy to po prostu poziomem wysokiej nierozstrzygalności i zilustrujemy to trzema przykładami4. Warto jednak zauważyć, że pomiędzy tymi dwoma poziomami, jak i zarówno poniżej, jak i poza nimi, istnieją wiele dodatkowych poziomów nierozstrzygalności. W rzeczywistości istnieje nieskończona hierarchia problemów, z których wszystkie

są coraz bardziej „mniej rozstrzygalne" niż te już opisane, ale „bardziej rozstrzygalne" niż wysoce nierozstrzygalne problemy, które teraz opisujemy. Wśród tych pośrednich problemów znajdują się problemy totalności, weryfikacji i prawdy w arytmetyce pierwszego rzędu. Podobnie istnieje nieskończona hierarchia problemów, z których wszystkie są jeszcze gorsze niż te przedstawione poniżej. Dwa z trzech przykładów, które teraz opisujemy, są nieco zaskakujące, ponieważ wydają się być nieistotnymi wariantami problemów, które już widzieliśmy. Rozważ problem spełnialności dla dynamicznej logiki zdań (PDL). W Części 7 problem został opisany jako (rozstrzygalny i) mający zarówno górną, jak i dolną granicę czasu wykładniczego. Programy, które mogą pojawić się w formułach PDL, są konstruowane z nieokreślonych programów elementarnych przy użyciu sekwencjonowania, instrukcji warunkowych i iteracji. Jeśli pozostawimy wszystkie inne aspekty formalizmu takimi, jakimi są, ale pozwolimy, aby te schematyczne programy były konstruowane również przy użyciu rekurencji, problem spełnialności staje się nie tylko nierozstrzygalny, ale wysoce nierozstrzygalny! Tak więc nie jest rozstrzygalne, nawet jeśli otrzymamy darmowe rozwiązania wszystkich częściowo rozstrzygalnych problemów opisanych wcześniej lub wielu nierozstrzygalnych problemów występujących na poziomach pośrednich. Drugi przykład to subtelny wariant regularnego kafelkowania lub problemu domina, który pyta, czy nieskończoną siatkę liczb całkowitych można ułożyć sąsiadująco przy użyciu tylko typów kafelków występujących w skończonym zestawie danych wejściowych T . (Tutaj preferowana jest ta wersja, a nie równoważna, obejmująca obszary wszystkich skończonych rozmiarów.) W nowym wariancie dodajemy mały wymóg: chcielibyśmy, aby kafelek, o którego istnienie pytamy, zawierał nieskończenie wiele kopii jednego konkretnego kafelka, powiedzmy, że pierwszy kafelek wymieniony w T. Chcemy „tak", jeśli istnieje kafelek siatki, która zawiera nieskończoną powtarzalność tego specjalnego kafelka, i chcemy „nie", jeśli takie kafelki nie istnieją, nawet jeśli istnieją inne kafelki całej siatki. Wydawałoby się, że ten dodatkowy wymóg nie powinien mieć większego znaczenia, ponieważ jeśli skończony zestaw typów płytek rzeczywiście może ułożyć całą nieskończoną siatkę, to niektóre typy w zestawie muszą występować w układaniu nieskończenie często. Zasadnicza różnica polega jednak na tym, że wskazujemy tutaj konkretny kafelek, którego powtarzalność nas interesuje. Mimo pozornego podobieństwa, ten powtarzający się problem domina jest wysoce nierozstrzygnięty (w rzeczywistości jest on obliczeniowo odpowiednikiem wspomnianego wcześniej problemu PDL z rekurencja). To również nie jest rozstrzygalne nawet przy darmowych rozwiązaniach wielu innych problemów na niższych poziomach.

Cztery podstawowe poziomy zachowań algorytmicznych

Wyłania się ciekawa wielostronna historia dotycząca problemów z układaniem płytek, czyli domino. Najpierw są problemy ograniczone, takie jak to, czy T może ułożyć kwadrat N na N dla danego N. Następnie są problemy nieograniczone, takie jak to, czy T może ułożyć obok siebie nieskończoną siatkę liczb całkowitych. Wreszcie, pojawiają się powtarzające się problemy, takie jak to, czy T może kafelkować nieskończoną siatkę tak, że dana kafelek powtarza się w nieskończoność. Można wykazać, że problemy ograniczone są NP-zupełne, a zatem przypuszczalnie nierozwiązywalne, problemy nieograniczone są nierozstrzygalne (ale częściowo rozstrzygalne), a problemy powracające są wysoce nierozstrzygalne. Aby uzupełnić obraz, istnieje również wersja problemu ograniczonego o stałej szerokości. Pyta, czy mając zbiór T i liczbę N, prostokąt o rozmiarze C na N może być utworzony z T , gdzie szerokość C jest stała i nie jest częścią danych wejściowych do problemu. (Zauważ, że w szczególnym przypadku, gdy C wynosi 1, pytamy o istnienie linii płytek przestrzegających ograniczenia kolorowania.) Dla każdego ustalonego C ten problem dopuszcza algorytm wielomianowy, którego wyszukiwanie może ci się spodobać. I tak, jak podsumowano w poniższej tabeli, mamy cztery wersje problemu kafelkowania, które przy (wiarygodnym) założeniu, że P = NP (to znaczy, że problemy NPzupełne są faktycznie nierozwiązywalne), dobrze reprezentują cztery podstawowe klasy zachowań algorytmicznych:

Rodzaj problemu: status algorytmu

Ograniczona o stałej szerokości: praktyczna

ograniczony: nieusuwalny

nieograniczony : nierozstrzygalny

cykliczne : wysoce nierozstrzygalne

Należy podkreślić, że to właściwość występująca w lewej kolumnie tabeli odpowiada za zasadniczą różnicę w statusie problemów, a nie drobne szczegóły techniczne w definicji problemu. Na poparcie tej tezy można przedstawić dwa argumenty. Po pierwsze, inne problemy można w podobny sposób uogólnić, aby uzyskać takie samo czteropoziomowe zachowanie. Na przykład problem korespondencji słów staje się NP-zupełny, jeśli długość wymaganego ciągu indeksów, według którego ma zostać skonstruowane wspólne słowo, jest ograniczona przez N; staje się wykonalne, jeśli w celu uzyskania wspólnego słowa mamy użyć ustalonej z góry liczby X, ale N z Y; i staje się wysoce nierozstrzygalne, jeśli mamy wykryć istnienie nieskończonej sekwencji tworzącej wspólne nieskończone słowo, ale z jednym konkretnym indeksem wymaganym do występowania w sekwencji nieskończenie często. Inne uzasadnienie można znaleźć w niewrażliwości tych czterech poziomów na techniczne różnice w definicji samych problemów. Można zdefiniować wiele wariantów tych problemów kafelkowania i dopóki zachowana jest podstawowa charakterystyka lewostronna, ich status algorytmiczny zwykle się nie zmienia. Na przykład możemy pracować z sześciokątnymi lub trójkątnymi płytkami zamiast kwadratów i dopasowanymi kształtami i/lub wycięciami zamiast kolorów (tak, aby puzzle z małpkami lub układankami mogły być podstawą zjawiska czterokierunkowego, a nie kolorowych płytek). Możemy wymagać zygzaków lub spiral o długości N w przypadku stałej szerokości, prostokątów lub trójkątów o podanych (wejściowych) wymiarach w przypadku ograniczonym, półsiatek lub danej płytki początkowej w przypadku nieograniczonej oraz powtarzalnego koloru zamiast płytki lub cykliczność ograniczona do jednego wiersza w przypadku cyklicznym. We wszystkich tych wariantach, a także w wielu innych, status prawej ręki pozostaje taki sam. Dlatego można śmiało powiedzieć, że ta czteropoziomowa klasyfikacja jest niezwykle solidna, a część 9 w rzeczywistości dostarczy dalszych ważnych dowodów na jej poparcie. Przed zamknięciem wspominamy trzeci przykład wysokiej (w rzeczywistości ekstremalnie wysokiej) nierozstrzygalności. Mamy na myśli ważność formuł w potężnym formalizmie logicznym, zwanym arytmetyką drugiego rzędu, który w rzeczywistości jest kombinacją cech występujących w trzech formalizmach rozumowania o liczbach całkowitych, o których mówiliśmy wcześniej: arytmetyka presburgera z jej zdolnością mówić o liczbach całkowitych obejmujących dodawanie; WS1S, z możliwością mówienia o zbiorach liczb całkowitych z dodawaniem; i arytmetyka pierwszego rzędu, z jej zdolnością do mówienia o liczbach całkowitych obejmujących zarówno dodawanie, jak i mnożenie. Arytmetyka drugiego rzędu ma to wszystko — jest w stanie mówić o zbiorach liczb całkowitych z dodawaniem i mnożeniem — a jej problem z ważnością jest bardzo nierozstrzygalny. Jak wyjaśnić ten „bardzo wysoko" kwalifikator? No cóż, unikając dodatkowych szczegółów technicznych, zacznijmy od stwierdzenia, że arytmetyka pierwszego rzędu jest o wiele gorsza niż „tylko" nierozstrzygalność; w rzeczywistości zawiera nieskończenie wiele poziomów rosnącej nierozstrzygalności poza problemami, takimi jak kafelkowanie i zatrzymywanie się (ale nie jest tak wysoce nierozstrzygalny jak PDL z programami rekurencyjnymi lub powtarzającymi się domino). W podobnym duchu arytmetyka drugiego rzędu jest o wiele gorsza niż „jedynie" wysoce nierozstrzygalne problemy (takie jak rekurencyjne PDL lub powtarzające się domino) i w rzeczywistości zawiera nieskończenie wiele poziomów o coraz gorszej złożoności niż nawet te! Obrazowe jest podsumowanie złożoności obliczeniowej tych logik w poniższej tabeli, która zapewnia inną perspektywę poziomów trudności, które pojawiają się, gdy ktoś podejmuje problem i wielokrotnie dodaje do niego cechy.

Formalizm logiczny : Mówi o : Złożoność

Arytmetyka Presburgera : liczby całkowite z + : podwójnie niewykonalne (podwójnie wykładnicza)

WSIS : zbiory liczb całkowitych z + : wysoce niepraktyczne (nieelementarne)

Arytmetyka pierwszego rzędu : liczby całkowite z + i × : bardzo nierozstrzygalne (ale nie całkiem wysoce nierozstrzygalne)

Arytmetyka drugiego rzędu : zbiory liczb całkowitych z + i × : bardzo wysoce nierozstrzygalne

Badanie nierozstrzygalności

Oprócz tego, że ma oczywiste znaczenie dla informatyki, nierozstrzygalność problemów algorytmicznych jest również interesująca dla matematyków. W rzeczywistości stanowi podstawę gałęzi logiki matematycznej znanej jako teoria funkcji rekurencyjnych. (Termin ten przypomina nieco, niestety, termin używany do opisywania podprogramów samowywoływania.) Być może zaskakujące jest to, że wiele podstawowych wyników, takich jak nierozstrzygalność problemu zatrzymania, zostało uzyskanych przez matematyków w połowie lat 30., na długo przed uruchomieniem komputerów. zostały zbudowane! Szczegółowa klasyfikacja problemów nierozstrzygalności na różne poziomy nierozstrzygalności jest nadal aktywnym kierunkiem badawczym. Wiele pracy jest poświęcone dopracowaniu i zrozumieniu różnych hierarchii nierozstrzygalności i ich wzajemnych relacji. Ponadto naukowcy są zainteresowani znalezieniem prostych problemów, które mogą służyć jako podstawa redukcji, które wykazują nierozstrzygalność lub wysoką nierozstrzygalność innych problemów. W przypadku wielu interesujących nas problemów, takich jak problem spełnialności dla PDL, czy problem równoważności składniowej języków, linia oddzielająca wersje rozstrzygalne od nierozstrzygalnych nie jest wystarczająco jasna i wymagane jest głębsze zrozumienie istotnych kwestii.

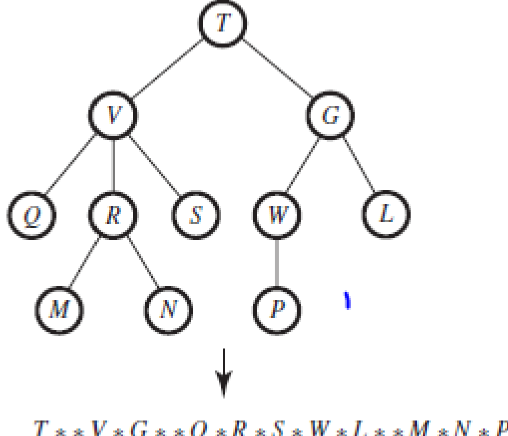
Problemy, których nie da się rozwiązać żadnym skutecznie wykonywalnym algorytmem, w rzeczywistości stanowią koniec pesymistycznej części historii i nadszedł czas, aby powrócić do szczęśliwszych, jaśniejszych problemów. Należy jednak pamiętać, że przy próbie rozwiązania problemu algorytmicznego zawsze istnieje szansa, że może on być w ogóle nie do rozwiązania lub może nie dopuszczać żadnego praktycznie akceptowalnego rozwiązania.

Uniwersalność algorytmów i ich wytrzymałość

W tej Części przyjrzymy się urządzeniom algorytmicznym najprostszym, jakie można sobie wyobrazić, uderzająco prymitywnym w przeciwieństwie do dzisiejszych komputerów i języków programowania. Niemniej jednak są wystarczająco wydajne, aby wykonywać nawet najbardziej złożone algorytmy. Biorąc pod uwagę obecny trend, w którym komputery z roku na rok stają się coraz bardziej skomplikowane i wyrafinowane, cel ten może wydawać się jedynie eksperymentem myślowym i prawdopodobnie zupełnie bezużytecznym. Jednak nasz cel jest trojaki. Po pierwsze, intelektualnie satysfakcjonujące jest odkrywanie przedmiotów, które są tak proste, jak to tylko możliwe, a jednocześnie tak potężne, jak wszystko w swoim rodzaju. Po drugie, powinniśmy naprawdę uzasadnić szeroki charakter negatywnych twierdzeń wysuwanych w dwóch ostatnich rozdziałach, dotyczących problemów, dla których nie ma sensownych rozwiązań, oraz innych, dla których nie ma żadnych rozwiązań. Opisane tutaj fakty będą stanowić ważne dowody na poparcie tych twierdzeń. Wreszcie, z powodów czysto technicznych, okaże się, że te prymitywne urządzenia dają początek rygorystycznym dowodom wielu z nierozstrzygalności, o których była mowa wcześniej. Zobaczmy najpierw, jak daleko możemy się posunąć w bezpośredniej próbie uproszczenia rzeczy.

Ćwiczenie z upraszczania danych

Pierwszą rzeczą, na którą należy zwrócić uwagę, jest to, że każdy element danych używany przez algorytm, czy to jako wartość wejściowa, wyjściowa czy pośrednia, może być traktowany jako ciąg symboli. Liczba całkowita to tylko ciąg cyfr, a liczbę ułamkową można zdefiniować jako dwa ciągi cyfr oddzielone ukośnikiem. Słowo w języku angielskim to ciąg liter, a cały tekst to nic innego jak ciąg symboli składający się z liter i znaków interpunkcyjnych ze spacjami. Inne pozycje, z którymi mieliśmy okazję wcześniej się spotkać to kolory, węzły na wykresie, linie, połówki małp, pierścienie, kołki, boki kwadratów, odcinki dróg, figury szachowe, operatory logiczne i oczywiście sam teksty programów. We wszystkich tych przypadkach moglibyśmy z łatwością zakodować takie obiekty, jak liczby, słowa lub teksty i traktować je symbolicznie. Liczba różnych symboli używanych we wszystkich takich kodowaniach jest w rzeczywistości skończona i zawsze można ją ustalić z wyprzedzeniem. Jest to pomysłowość standardowego systemu liczbowego, takiego jak system dziesiętny. Nie potrzebujemy nieskończenie wielu symboli, po jednym dla każdej liczby - wystarczy 10 symboli, aby zakodować je wszystkie. (System binarny używa tylko dwóch, 0 i 1.) To samo oczywiście dotyczy słów i tekstów. W konsekwencji, możemy zapisać dowolny element danych na taśmie jednowymiarowej, być może długą, która składa się z sekwencji kwadratów, z których każdy zawiera pojedynczy symbol, który jest elementem jakiegoś skończonego alfabetu. Ten pomysł można posunąć znacznie dalej. Nietrudno zauważyć, że nawet najbardziej skomplikowane struktury danych można „zlinearyzować” w ten sposób. Na przykład wektor jest po prostu listą elementów danych i może być przedstawiony jako sekwencja zlinearyzowanych wersji każdego z elementów, oddzielonych specjalnym symbolem, takim jak „∗”. Dwuwymiarową tablicę można rozłożyć wiersz po wierszu wzdłuż taśmy, używając „∗” do oddzielenia elementów w każdym wierszu i powiedzmy „∗∗” do oddzielenia wierszy. Linearyzacja drzew wymaga większej troski. Jeśli spróbujemy naiwnie wyliczyć elementy drzewa poziom po poziomie, dokładna struktura drzewa może zostać utracona, ponieważ liczba elementów na danym poziomie nie jest ustalona. Na przykład w kodowaniu poziom po poziomie na rysunku



nie ma możliwości sprawdzenia, czy S jest potomkiem V czy G. Jednym ze sposobów uniknięcia tego problemu jest przyjęcie wariantu podejścia LISP z listami zagnieżdżonymi , jak pokazano na przykładach programów SCHEME . (Nawiasy są traktowane jako symbole specjalne, takie jak „∗” i „∗∗”.) Jest to szczególnie korzystne, gdy drzewa mają informacje tylko w liściach. Alternatywnie, możemy udoskonalić metodę z rysunku, zaznaczając skupiska bezpośredniego potomstwa, poziom po poziomie, zawsze zaczynając od lewej. Oto wynikowa linearyzacja drzewa z rysunku:

(T)(V, G)(Q, R, S)(W, L)( )(M, N)( )(P)( )

Zachęcamy do opracowania algorytmów zarówno do reprezentowania drzew przez takie listy, jak i do rekonstrukcji drzew z list. Podobne przekształcenia można przeprowadzić dla dowolnego rodzaju struktury danych, nawet złożonej. Wiele zmiennych lub struktur danych można opisać w sposób liniowy za pomocą nowego symbolu oddzielającego ich zlinearyzowane wersje od siebie. W rzeczywistości całe bazy danych, składające się z dziesiątek tabel, rekordów i plików, mogą być zakodowane jako długie listy symboli, z odpowiednimi kodami symboli specjalnych oznaczającymi punkty przerwania między różnymi częściami. Oczywiście praca z liniową wersją wysoce ustrukturyzowanego zbioru danych może być bardzo nieefektywna. Nawet podana właśnie prosta reprezentacja drzewa klastrowego wymaga dość nieprzyjemnej ilości biegania w celu wykonania standardowych zadań zorientowanych na drzewo, takich jak przemierzanie ścieżek lub izolowanie poddrzew zakorzenionych w danych węzłach. Jednak wydajność nie jest obecnie jednym z naszych zmartwień; jest to uproszczenie pojęciowe i po raz kolejny stwierdzamy, że wystarczy taśma liniowa z symbolami ze skończonego alfabetu. Teraz algorytmy nie zajmują się tylko stałą ilością danych. Mogą prosić o więcej w miarę postępów. Struktury danych mogą się powiększać, a zmiennym można przypisywać coraz większe elementy danych; informacje mogą być przechowywane do późniejszego wykorzystania w algorytmie i tak dalej. Jednak biorąc pod uwagę, że wszelkie takie dodatkowe dane można również zlinearyzować, wystarczy, że nasza taśma z zaznaczonymi kwadratami ma nieograniczoną długość. Przechowywanie niektórych informacji będzie żmudne, biegnąc do jakiejś zdalnej, nieużywanej części taśmy i tam ją przechowując. Wniosek jest taki. Jeśli chodzi o dane manipulowane przez algorytm, dowolny skutecznie wykonywalny algorytm, wystarczy mieć jednowymiarową taśmę o potencjalnie nieograniczonej długości, podzieloną na kwadraty, z których każdy zawiera symbol zaczerpnięty ze skończonego alfabetu. Alfabet ten zawiera „prawdziwe” symbole, które tworzą same elementy danych, a także specjalne symbole do oznaczania punktów przerwania. Zakłada się również, że zawiera specjalny pusty symbol wskazujący na brak informacji, który oznaczymy # i który należy rozumieć jako różny od symbolu spacji oddzielającego słowa w tekście. Ponieważ w dowolnym momencie algorytm zajmuje się tylko skończoną ilością danych, nasze taśmy zawsze będą zawierać skończoną znaczną część danych, otoczoną z obu stron nieskończonymi

sekwencjami pustych miejsc. Ta część może być bardzo długa i może rosnąć w miarę postępu egzekucji, ale zawsze będzie skończona.

Ćwiczenie w upraszczaniu kontroli

Jak możemy uprościć część kontrolną algorytmu? Jak widzieliśmy, różne języki obsługują różne struktury kontrolne, takie jak sekwencjonowanie, warunkowe rozgałęzianie, podprogramy i rekurencja. Nasze pytanie tak naprawdę dotyczy uproszczenia pracy procesora Runaround, który biega dookoła wykonując podstawowe instrukcje. Na razie zignorujemy same podstawowe instrukcje i skoncentrujemy się na uproszczeniu samego trudu biegania. Jedną z rzeczy kluczowych dla uproszczenia sterowania jest skończoność tekstu algorytmu. Procesor może znajdować się w jednym ze skończenie wielu miejsc w tym tekście, a więc możemy zadowolić się dość prymitywnym mechanizmem, zawierającym jakiś rodzaj gearboxa, który może znajdować się w jednej ze skończenie wielu pozycji lub stanów. Jeśli pomyślimy o stanach skrzyni biegów jako o kodowaniu lokalizacji w algorytmie, to poruszanie się w algorytmie można modelować po prostu zmieniając stany. W dowolnym momencie wykonywania algorytmu kolejna lokalizacja do odwiedzenia zależy od aktualnej lokalizacji, tak więc kolejny stan gearboxa naszego mechanizmu musi zależeć od jego aktualnego stanu. Jednak nowa lokalizacja może również zależeć od wartości niektórych elementów danych; wiele struktur kontrolnych testuje wartości zmiennych w celu skierowania kontroli do określonych miejsc w tekście (na przykład do klauzul then lub else instrukcji if albo do początku lub końca pętli while). Oznacza to, że zmiana stanu naszego mechanizmu musi zależeć zarówno od części danych, jak i od aktualnego stanu. Ale ponieważ zakodowaliśmy wszystkie nasze dane na jednej, długiej taśmie, musimy pozwolić naszemu prymitywnemu mechanizmowi na sprawdzenie taśmy przed podjęciem decyzji o nowym stanie. W trosce o minimalizm i prostotę ta inspekcja będzie przeprowadzana tylko po jednym kwadracie na raz. W danym momencie tylko jeden symbol zostanie „odczytany". Nasz mechanizm może być zatem postrzegany jako posiadający „oko" o bardzo ograniczonej mocy, kontemplujące co najwyżej jeden kwadrat taśmy na raz, oraz widzące i rozpoznające symbol tam rezydujący. W zależności od tego symbolu i aktualnego stanu mechanizmu, może on „zmienić bieg", wchodząc w nowy stan. W konsekwencji skończoności alfabetu zobaczymy później, że mechanizm może faktycznie „zapamiętać" symbol, który widział, wchodząc w odpowiednio znaczący nowy stan. Dzięki temu może działać zgodnie z połączonymi informacjami zebranymi z kilku kwadratów taśmy. Jednak w celu sprawdzenia różnych części danych, musimy pozwolić naszemu mechanizmowi poruszać się po taśmie. Znowu będziemy bardzo nieżyczliwi, pozwalając, aby ruch odbywał się tylko na jednym polu taśmy na raz. Kierunek ruchu (w prawo lub w lewo) będzie również zależał zarówno od aktualnego stanu skrzyni biegów, jak i od symbolu, który właśnie widziało oko. Obserwacje te znacznie upraszczają element sterujący algorytmu. Mamy do czynienia z prostym mechanizmem, zdolnym do bycia w jednym ze skończonej liczby biegów lub stanów, szarpiąc taśmę po jednym kwadracie. W trakcie tego procesu zmienia stany i przełącza kierunki w zależności od aktualnego stanu i pojedynczego symbolu, który akurat widzi przed nim.

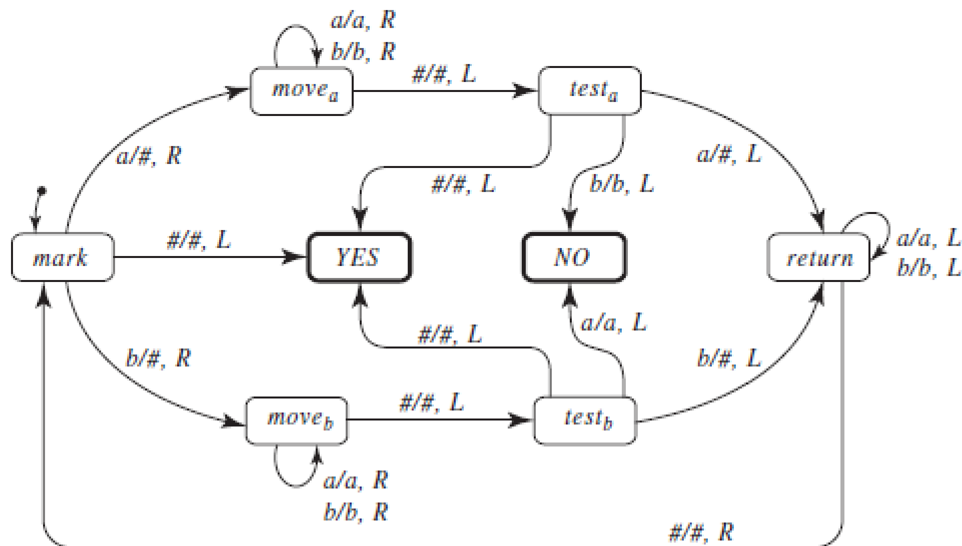
Uproszczenie podstawowych operacji

Upraszczając w ten sposób dane i elementy sterujące algorytmów, pozostajemy z podstawowymi operacjami, które faktycznie realizują zadania. Jeśli procesory miałyby tylko biegać w kółko, odczytując części danych i zmieniając biegi, algorytmy nie mogłyby wiele zrobić. Potrzebujemy możliwości manipulowania danymi, stosowania do nich transformacji, usuwania, pisania lub przepisywania ich części, stosowania do nich operacji arytmetycznych lub tekstowych i tak dalej. Nie dając teraz żadnego uzasadnienia, wyposażymy nasz mechanizm w tylko najbardziej trywialne możliwości manipulacji. Poza zmianą stanów i przesunięciem o jedno pole w prawo lub w lewo, jedyne, co można zrobić, gdy w danym stanie i spojrzy na konkretny symbol na taśmie, to przekształcenie tego symbolu w jeden z

pozostałych skończenie wielu. dostępne symbole. To wszystko. Mechanizm wynikający z tej długiej sekwencji uproszczeń nazywa się maszyną Turinga, od nazwiska brytyjskiego matematyka Alana M. Turinga, który wynalazł ją w 1936 roku.

Maszyna Turinga

Bądźmy nieco bardziej precyzyjni w definicji maszyn Turinga. Maszyna Turinga M składa się z (skończonego) zbioru stanów, (skończonego) alfabetu symboli, nieskończonej taśmy z jej wydzielonymi kwadratami oraz głowica do pisania, która może poruszać się po taśmie, po jednym kwadracie. Ponadto sercem maszyny jest diagram przejść stanów, czasami nazywany po prostu diagramem przejść, zawierający instrukcje powodujące zmiany zachodzące na każdym kroku. Diagram przejścia może być postrzegany jako graf ukierunkowany, którego węzły reprezentują stany. Używamy zaokrąglonych prostokątów (rountangles w sequelu) dla stanów

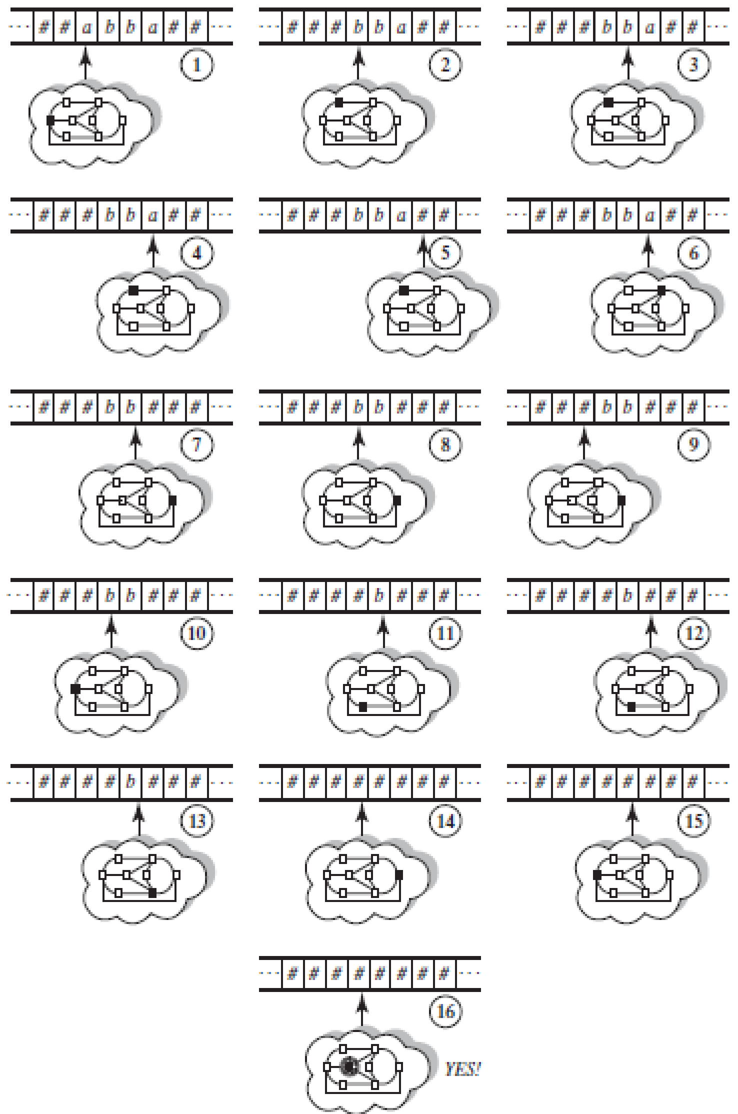


Krawędź prowadząca ze stanu s do stanu t nazywana jest przejściem i jest oznaczona kodem postaci a/b, L lub a/b, R , gdzie a i b są symbolami. Część etykiety nazywana jest wyzwalaczem przejścia i oznacza literę odczytaną z taśmy. Część b to akcja i oznacza literę zapisaną na taśmie. Wreszcie część L i R określa kierunek ruchu, przy czym L oznacza „lewo", a R „prawo". Dokładne znaczenie przejścia od s do t oznaczonego a/b, L jest następujące (przypadek a/b, R jest podobny): Podczas pracy, ilekroć maszyna Turinga znajduje się w stanie s, a jest symbolem wyczuwanym w tym momencie przez głowę, maszyna usunie symbol a, wpisując w jego miejsce b, przesunie się o jedno pole w lewo i wejdzie w stan t. Aby zapobiec niejasności co do następnego ruchu maszyny (to znaczy, aby jej zachowanie było deterministyczne), wymagamy, aby żadne dwa przejścia z tym samym wyzwalaczem nie pochodziły z jednego stanu. Jeden ze stanów na diagramie (oznaczenie na rysunku) jest oznaczony specjalną małą strzałką wejściową i nazywany jest stanem początkowym. Również stany, które nie mają przejść wychodzących (YES i NO na rysunku) nazywane są stanami zatrzymania i są podkreślone grubymi prostokątami. Dla wygody do jednego przejścia można przymocować kilka etykiet (jak w trzech strzałkach samocyklu na rysunku). Zakłada się, że maszyna uruchomi się w swoim stanie początkowym na skrajnym lewym niepustym kwadracie taśmy i będzie postępować krok po kroku zgodnie z zaleceniami diagramu. Zatrzymuje się, jeśli i kiedy wejdzie w stan zatrzymania.

Wykrywanie palindromów: Przykład

Przyjrzyjmy się bliżej maszynie Turinga, której schemat przejścia pokazano na rysunku powyżej

W szczególności zasymulujmy jej działania na taśmie składającej się ze słowa „a b b a” (otoczonej, jak wyjaśniono powyżej, nieskończenie wieloma pustymi miejscami po obu stronach). Głowica maszyny znajduje się skrajnie po lewej stronie a, a znak jest stanem początkowym. Rysunek



przedstawia całą symulację, krok po kroku, z aktualnym stanem wskazanym przez ciągłą prostokąt w miniaturowej wersji diagramu przejścia. Intuicyjnie maszyna „zapamiętuje” pierwszy widziany symbol (wprowadzając movea lub moveb, w zależności od tego, czy był to symbol a, czy b), usuwa go, zastępując go spacją, a następnie biegnie do końca w prawo, aż do osiągnie puste miejsce (klatka 5). Następnie przesuwa jeden symbol w lewo, tak że znajduje się on na najbardziej prawym symbolu w stanie testa lub testb, w zależności od symbolu, który zapamiętał (klatka 6). Gdyby teraz wyczuwał inny symbol niż ten, który zapamiętał, wszedłby w specjalny stan NIE. W naszym przypadku jednak zapamiętał a, a teraz widzi także a, więc wymazuje to skrajne prawe a i wprowadza powrót, który sprowadza go w lewo aż do pierwszego pustego miejsca, w znaku stanu (klatka 9). Tak jak poprzednio, maszyna przesuwa się teraz o jedno pole w prawo i zapamiętuje widziany symbol. Ten symbol to jednak drugi z oryginalnego ciągu, ponieważ pierwszy został wcześniej wymazany. Obecny bieg w prawo porównuje teraz ten drugi symbol z przedostatnim symbolem ciągu (klatka 13). To porównanie również jest udane, a maszyna wymazuje b i przesuwa się w lewo w poszukiwaniu dalszych symboli. Nie znajdując żadnego i nie osiągając stanu NIE z powodu niedopasowania, wpisuje TAK, wskazując, że

wszystkie pary pasują. Zasadniczo ta maszyna Turinga sprawdza palindromy; to znaczy dla słów, które czytają to samo z obu stron. (Palindrom pozostałby taki sam po poddaniu go odwrotnemu algorytmowi z rozdziału 5.) Za każdym razem, gdy symbol skończonego słowa składającego się z a i b zaczyna się od skrajnego lewej strony, zatrzymuje się w stanie TAK, jeśli słowo to jest palindromem, a w przypadku NIE jeśli nie jest. Powinieneś symulować maszynę również na „a b a b a" i „a b a b b a a", aby lepiej wyczuć jej zachowanie. Zauważ, że ta konkretna maszyna używa czterech stałych stanów i dodatkowych dwóch dla każdej litery alfabetu. Możesz również rozszerzyć rysunek 9.5, aby poradzić sobie z przypadkiem trzyliterowego alfabetu.

Maszyny Turinga jako algorytmy

Uważne spojrzenie pokazuje, że maszyna Turinga może być postrzegana jako komputer z jednym ustalonym programem. Oprogramowanie jest diagramem przejścia, a sprzęt składa się z taśmy i głowicy, a także (ukrytego) mechanizmu, który faktycznie porusza się po diagramie przejścia, zmieniając stany i kontrolując czynności czytania, pisania, kasowania i przenoszenia głowicy. Komputer jest więc taki sam dla wszystkich maszyn Turinga; to programy są inne. W konsekwencji ludzie czasami mówią o programowaniu maszyny Turinga, a nie o jej budowaniu. Tak więc przykład palindromu faktycznie pokazuje, jak można zaprogramować maszynę Turinga, aby rozwiązać problem decyzyjny. W przypadku problemu P, którego prawidłowy zbiór danych wejściowych został zakodowany jako zbiór zlinearyzowanych ciągów, próbujemy opracować maszynę Turinga M ze stanem początkowym s i dwoma stanami specjalnymi TAK i NIE, która wykonuje następujące czynności: Dla dowolnego dozwolonego ciągu wejściowego X, jeśli M jest uruchamiany w stanie s przy skrajnym lewym symbolu X na pustej taśmie zawierającej jedną kopię X, w końcu wchodzi TAK lub NIE, w zależności od tego, czy odpowiedź P na X brzmi „tak" czy „nie". Wykrywanie palindromów za pomocą maszyny Turinga może wydawać się łatwe, ale inne problemy nie są. Jak wykrywamy ciągi z następującą właściwością. Jedyne pojawienie się litery a w łańcuchu występuje w blokach, których długość stanowi pewną początkową część ciągu liczb pierwszych 2, 3, 5, 7, 11, . . . W obrębie X bloki nie muszą pojawiać się w określonej kolejności. Tutaj nie możemy po prostu biegać tam i z powrotem, wymazując symbole; potrzebujemy umiejętności liczenia i obliczania, a skończona liczba stanów nie wystarcza, aby „zapamiętać" dowolnie duże liczby. (Słowo wejściowe może być oczywiście dowolnie długie). Sztuczka polega na obliczeniu następnej liczby pierwszej na pustej części taśmy, powiedzmy po prawej stronie słowa wejściowego X, a następnie wyszukaniu bloku z dokładnie wymaganej długości. Zakładając, że bieżąca liczba pierwsza została obliczona i pojawia się jako ciąg jedynek na prawo od X na taśmie2, wyszukiwanie odpowiedniego bloku a można przeprowadzić w następujący sposób. Maszyna wielokrotnie przechodzi przez każdy blok a, porównując każdy z jednym z jedynek, biegając tam iz powrotem i zmieniając je tymczasowo w jakiś nowy symbol. Jeśli zostanie znalezione idealne dopasowanie, cały blok jest usuwany i obliczana jest następna liczba pierwsza. Jeśli nie, blok jest przywracany do swojej pierwotnej formy i wypróbowywany jest następny blok. Jeśli w całym łańcuchu nie zostanie znaleziony pasujący blok, maszyna wprowadzi NIE. Jeśli blok pasuje, a na taśmie nie ma już a, maszyna wpisuje TAK. Zwróć uwagę, jak używamy potencjalnie nieskończonej pustej części taśmy jako skrawka papieru, zarówno do obliczenia, jak i zapisania naszej aktualnej liczby a. Maszyny Turinga można również zaprogramować do rozwiązywania problemów algorytmicznych, które nie są tak/nie. Jedyna różnica polega na tym, że trzeba wyprodukować dane wyjściowe. Zgodnie z konwencją możemy: zgadzać się, że kiedy maszyna Turinga zatrzymuje się (z racji wejścia w dowolny stan zatrzymania), wyjściem jest ciąg znaków zawarty między dwoma znakami „!" na taśmie. Jeśli w momencie zatrzymania na taśmie nie ma dokładnie dwóch „!", zgadzamy się, że to tak, jakby maszyna weszła w nieskończoną pętlę i dlatego nigdy się nie zatrzyma. Innymi słowy, jeśli M chce wygenerować wynik, lepiej zadbać o to, aby wynik był ujęty między dwoma znakami „!" i aby nie używał znaku „!" w żadnym innym celu. (Oczywiście, możliwe jest użycie tej konwencji również do problemów decyzyjnych, pisząc „!tak!" lub „!nie!" na taśmie i

wchodząc w jakiś stan wstrzymania, zamiast wchodzić w specjalne stany TAK lub NIE, jak to zrobiliśmy w przypadku maszyna palindrom.)

Mając na uwadze tę definicję, pouczającym ćwiczeniem jest zaprogramowanie maszyny Turinga w celu dodania dwóch liczb dziesiętnych X i Y . Jeden ze sposobów postępowania jest następujący. Maszyna dobiega do skrajnej prawej cyfry X (dochodząc do symbolu rozdzielającego „∗” i przesuwając o jeden kwadrat w lewo) i kasuje go, „pamiętając” go w jego stanie; do tego będzie potrzebnych 10 różnych stanów, powiedzmy cyfra-jest-0 do cyfra-jest-9. Następnie biegnie do skrajnej prawej cyfry Y i również ją usuwa, jednocześnie wchodząc w stan, który zapamiętuje cyfrę sumy dwóch liczb i czy istnieje przeniesienie. (Oczywiście zależą one tylko od aktualnej i zapamiętanej cyfry i mogą być zakodowane w stanach sum-is-0-nocarry do sumis-9-nocarry i sum-is-0-carry do sum- is-9-carry). Następnie maszyna przesuwa się na lewo od tego, co pozostało z X i zapisuje cyfrę sumy w dół, po przygotowaniu „!” jako ogranicznik. Następny krok jest podobny, ale obejmuje cyfry znajdujące się obecnie najbardziej na prawo i uwzględnia przeniesienie (jeśli istnieje). Nowa cyfra sumy jest zapisywana po lewej stronie poprzedniej i proces jest kontynuowany.

Oczywiście musimy pamiętać, że każdej z liczb może zabraknąć cyfr przed drugą, w takim przypadku po dodaniu przeniesienia (jeśli jest) do pozostałej części większej liczby, ta część jest po prostu kopiowana na lewo. Wreszcie drugie „!” jest napisane po lewej stronie i maszyna zatrzymuje się. Oto główne konfiguracje taśmy dla numerów 736 i 63519:

```
... # # # # # # # # # 7 3 6 ∗ 6 3 5 1 9 # # ...
... # # # # # # # 5 ! 7 3 # ∗ 6 3 5 1 # # # ...
... # # # # # # 5 5 ! 7 # # ∗ 6 3 5 # # # # ...
... # # # # # 2 5 5 ! # # # ∗ 6 3 # # # # # ...
... # # # # 4 2 5 5 ! # # # ∗ 6 # # # # # # ...
... # # # 6 4 2 5 5 ! # # # ∗ # # # # # # # ...
... # # ! 6 4 2 5 5 ! # # # ∗ # # # # # # # ...
```

Ponownie, czytelnik (masochistyczny) może być zainteresowany skonstruowaniem całego diagramu przejścia maszyny Turinga do dodawania dziesiętnego.

Teza Churcha/Turinga

Te przykłady mogą być trochę zaskakujące. Automatycznie ustawiana maszyna ma tylko skończenie wiele stanów i może przepisywać symbole na taśmie liniowej tylko jeden na raz. Niemniej jednak możemy zaprogramować go do dodawania liczb. Programowanie może być żmudne (spróbuj skonstruować maszynę Turinga do mnożenia liczb), a przejęcie kontroli i przeprowadzenie symulacji działania maszyny wcale nie jest łatwiejsze. Niemniej jednak wykonuje swoją pracę. Mając to na uwadze, zapomnijmy na razie o nudzie i wydajności i zadajmy sobie pytanie, co tak naprawdę można zrobić z maszynami Turinga, za wszelką cenę? Jakie problemy algorytmiczne może rozwiązać odpowiednio zaprogramowana maszyna Turinga? Odpowiedź nie jest trochę zaskakująca, ale rzeczywiście bardzo zaskakująca. Maszyny Turinga są w stanie rozwiązać każdy skutecznie rozwiązywalny problem algorytmiczny! Innymi słowy, każdy problem algorytmiczny, dla którego możemy znaleźć algorytm, który można zaprogramować w jakimś języku programowania, dowolnym języku, działającym na jakimś komputerze, dowolnym komputerze, nawet takim, który nie został jeszcze zbudowany, ale można go zbudować, a nawet taki, który będzie wymagać nieograniczonej ilości czasu i miejsca w pamięci dla coraz większych danych wejściowych, jest również rozwiązywalny przez maszynę Turinga. Stwierdzenie to jest jedną z wersji tzw. tezy Churcha/Turinga, po Alonzo Churchu i Turingu, którzy doszli do niej niezależnie w połowie lat 30. XX wieku. Ważne jest, aby zdać sobie

sprawę, że teza CT, jak ją czasem nazwiemy (zarówno w przypadku teorii Churcha/Turinga, jak i teorii obliczalności), jest tezą, a nie twierdzeniem, a zatem nie może być udowodniona w matematycznym sensie tego słowa. Powodem tego jest to, że wśród pojęć, które obejmuje, jest jedno, które jest nieformalne i nieprecyzyjne, a mianowicie „efektywna obliczalność". Teza ta zrównuje matematycznie precyzyjne pojęcie „rozwiązywalne przez maszynę Turinga" z nieformalnym, intuicyjnym pojęciem „efektywnie rozwiązywalne", które odnosi się do wszystkich prawdziwych komputerów i wszystkich języków programowania, zarówno tych, o których wiemy obecnie, jak i tych, które my nie. Brzmi to zatem bardziej jak szalona spekulacja niż tym, czym jest w rzeczywistości: głębokim i dalekosiężnym stwierdzeniem, wysuniętym przez dwóch najbardziej szanowanych pionierów informatyki teoretycznej. Pouczające jest nakreślenie analogii między maszynami Turinga a maszynami do pisania. Maszyna do pisania jest również bardzo prymitywnym rodzajem maszyny, umożliwiającą nam jedynie wpisywanie sekwencji symboli na kartce papieru, która ma potencjalnie nieskończone rozmiary i jest początkowo pusta. (Maszyna do pisania ma również skończenie wiele „stanów" lub trybów działania - wielkie lub małe litery, czerwona lub czarna wstążka itp.) Mimo to, każda maszyna do pisania może być użyta do napisania Hamleta, Wojny i pokoju lub innego wysoce wyrafinowanego ciąg symboli. Oczywiście może to wymagać Szekspira lub Tołstoja, aby „poinstruować" maszynę, aby to zrobiła, ale można to zrobić. Analogicznie, programowanie maszyn Turinga do rozwiązywania trudnych problemów algorytmicznych może wymagać bardzo utalentowanych ludzi, ale podstawowy model, jak mówi nam teza CT, wystarcza dla wszystkich problemów, które w ogóle można rozwiązać. Nasze ćwiczenia w zakresie uproszczenia okazały się więc mieć poważne konsekwencje. Uproszczenie danych do sekwencji w skończonym alfabecie, uproszczenie kontroli do skończonej liczby stanów, które określają ruchy kwadrat po kwadracie na taśmie, oraz przyjęcie przepisywania symboli jako jedynej podstawowej operacji, daje mechanizm, który jest równie potężny jak każda inna urządzenie algorytmiczne.

Dowody dla Tezy Churcha / Turinga

Dlaczego mielibyśmy wierzyć w tę tezę, zwłaszcza gdy nie można jej udowodnić? Jakie są na to dowody i jak te dowody sprawdzają się w dobie codziennego postępu zarówno w sprzęcie, jak i oprogramowaniu? Od wczesnych lat 30-tych XX wieku badacze sugerowali modele wszechmocnego komputera absolutnego, czyli uniwersalnego. Intencją była próba uchwycenia tego śliskiego i nieuchwytnego pojęcia „skutecznej obliczalności", a mianowicie zdolności do obliczeń mechanicznych. Na długo przed wynalezieniem pierwszych komputerów cyfrowych Turing zasugerował swoje prymitywne maszyny, a Church opracował prosty matematyczny formalizm funkcji zwany rachunkiem lambda (jako podstawa języków programowania funkcjonalnego). Mniej więcej w tym samym czasie Emil Post zdefiniował pewien rodzaj systemu produkcji manipulującego symbolami, a Stephen Kleene zdefiniował klasę obiektów zwanych funkcjami rekurencyjnymi. (Jak wspomniano w części poświęconej badaniom w Części 8, ta „rekurencja" ma znaczenie nieco inne niż użyte tu). Znane były algorytmy „skutecznie wykonywalne". Inni ludzie zaproponowali od tego czasu wiele różnych modeli absolutnego, uniwersalnego urządzenia algorytmicznego. Niektóre z tych modeli są bardziej podobne do prawdziwych komputerów, posiadających abstrakcyjny odpowiednik jednostek pamięci i jednostek arytmetycznych oraz zdolność do manipulowania danymi za pomocą struktur kontrolnych, takich jak pętle i podprogramy, a niektóre mają charakter czysto matematyczny, definiując klasy funkcji, które są możliwe do zrealizowania krok po kroku. krok mody. Kluczowym faktem dotyczącym tych modeli jest to, że wszystkie okazały się równoważne pod względem klasy problemów algorytmicznych, które mogą rozwiązać. I ten fakt jest nadal aktualny, nawet w przypadku najpotężniejszych wymyślonych modeli. To, że tak wielu ludzi, pracujących z tak różnorodnymi narzędziami i koncepcjami, w gruncie rzeczy uchwyciło to samo pojęcie, jest dowodem na głębię tego pojęcia. To, że wszyscy szli za tą samą intuicyjną koncepcją i otrzymali różnie wyglądające, ale równoważne definicje, jest

usprawiedliwieniem dla zrównania tego intuicyjnego pojęcia z wynikami tych precyzyjnych definicji. Stąd teza CT.

Obliczalność jest solidna

Teza CT sugeruje, że najpotężniejszy superkomputer, z najbardziej wyrafinowaną gamą języków programowania, interpreterów, kompilatorów, asemblerów i wszystkiego innego, nie jest potężniejszy niż komputer domowy ze swoim uproszczonym językiem programowania! Mając nieograniczoną ilość czasu i miejsca w pamięci, oba mogą rozwiązać dokładnie te same problemy algorytmiczne. Nieobliczalne (lub nierozstrzygalne) problemy z rozdziału 8 nie są rozwiązywalne na żadnym z nich, a problemy obliczeniowe (lub rozstrzygalne) wspomniane w dalszej części są rozwiązywalne na obu. W wyniku tezy CT klasa problemów algorytmicznych, które są obliczalne, efektywnie rozwiązywalne lub rozstrzygalne, staje się niezwykle wytrzymała. Jest to niezmienne w przypadku zmian w modelu komputerowym lub języku programowania, o czym wspomniano w rozdziale 8. Zwolennicy określonej architektury komputerowej lub dyscypliny programowania muszą znaleźć powody inne niż czysta moc obliczeniowa, aby uzasadnić swoje zalecenia, ponieważ problemy, które można rozwiązać za pomocą jednej z nich, są również rozwiązywalne z innymi i wszystkie są równoważne prymitywnym maszynom Turinga lub różnym formalizmom Churcha, Posta, Kleene′a i innych. Granica wytyczona między rozstrzygalnym a nierozstrzygalnym w Części 8 jest zatem w pełni uzasadniona, podobnie jak nasze poleganie na nieokreślonym języku L przy omawianiu w nim nierozstrzygalności. Co więcej, intelektualnie satysfakcjonujące jest móc wskazać najprostszy model, który jest tak potężny jak wszystko w swoim rodzaju.

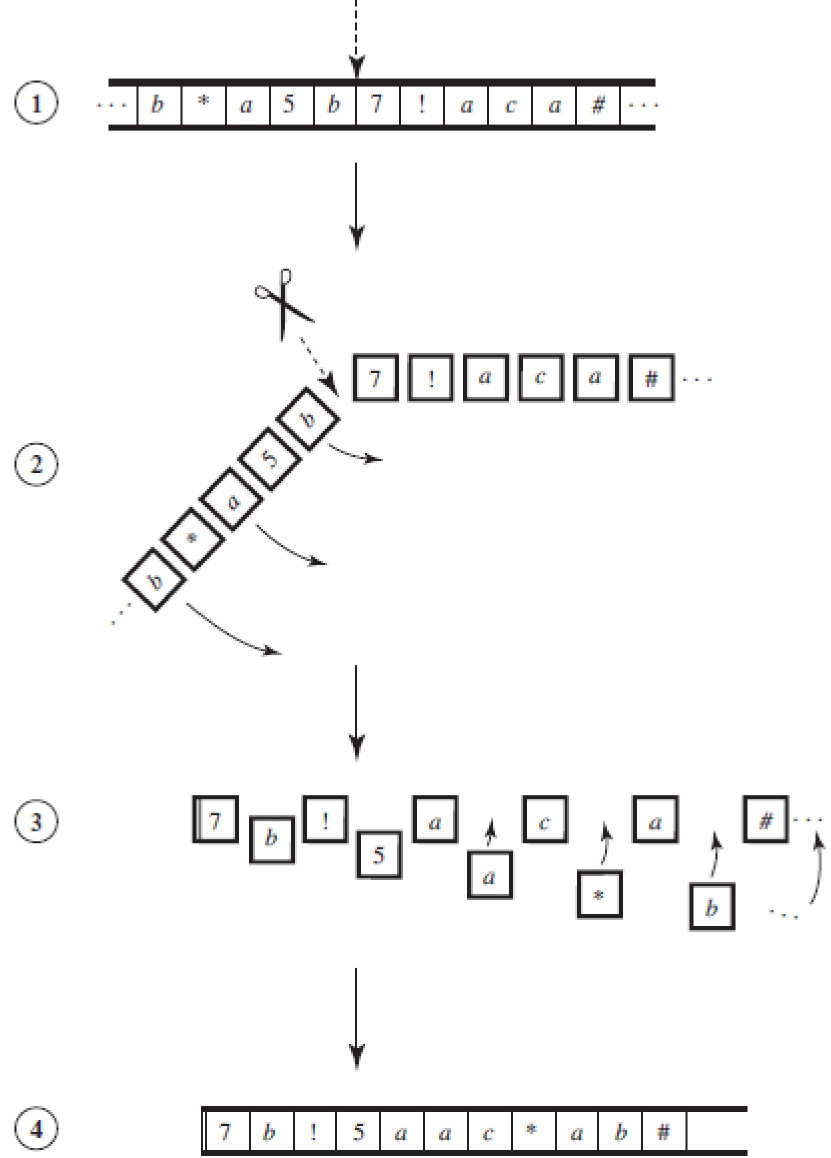
Warianty modelu maszyny Turinga

Solidność zapewniona przez tezę CT zaczyna się od wariantów samych maszyn Turinga. Jak się okazuje, możliwe jest ograniczanie maszyn na wiele sposobów bez zmniejszania klasy problemów, które mogą rozwiązać. Na przykład możemy wymagać, aby (w przeciwieństwie do efektu kasowania maszyny palindromowej) dane wejściowe były nienaruszone, a „obszary robocze” taśmy zostały oczyszczone, aby po zatrzymaniu taśma zawierała tylko wejście i wyjście, otoczone pustymi miejscami. Możemy zdefiniować maszyny Turinga z taśmą, która jest nieskończona tylko po prawej stronie, dane wejściowe wydają się wyrównane po lewej stronie, a maszyna jest ograniczona, aby nigdy nie próbowała „zjechać” z najbardziej wysuniętego na lewo kwadratu. Oba warianty potrafią rozwiązać dokładnie te same problemy, co model podstawowy, dlatego tak naprawdę nie są słabsze. W podobnym duchu, dodanie jakiejkolwiek potężnej (ale „skutecznie wykonywalnej”) funkcji do maszyn również daje dokładnie tę samą klasę problemów, które można rozwiązać, tak że w kontekście surowej obliczalności ta dodatkowa moc jest jedynie iluzją. Na przykład, możemy dopuścić wiele taśm, każda z własną głowicą odczytu/zapisu, w taki sposób, że przejścia bazują na całym zestawie symboli widzianych jednocześnie przez głowice; akcje określają nowy symbol do zapisania na każdej taśmie i kierunek poruszania się każdej głowy. Podobnie możemy zdefiniować maszyny wykorzystujące dwuwymiarowe taśmy, dające początek czterem, a nie dwóm możliwym kierunkom poruszania się i tak dalej. Żadne z tych rozszerzeń nie rozwiąże problemów, których nie może rozwiązać model podstawowy. Jedno z najciekawszych rozszerzeń dotyczy niedeterministycznych maszyn Turinga. Chodzi o to, aby ze stanu emanować wiele przejść z tym samym wyzwalaczem. Maszyna ma wtedy możliwość wyboru przejścia. Sposób, w jaki mówi się, że niedeterministyczna maszyna rozwiązuje problem decyzyjny, jest bardzo podobny do sposobu, w jaki „magiczny niedeterminizm” został zdefiniowany w Części 7: ilekroć istnieje wybór do dokonania, można uznać, że maszyna dokonuje najlepszego – że to taki, który ostatecznie doprowadzi do odpowiedzi „tak”, jeśli jest to w ogóle możliwe. W ten sposób niedeterministyczna maszyna Turinga mówi „tak”, aby wprowadzić X dokładnie, jeśli istnieje pewna sekwencja wyborów, która prowadzi do stanu TAK, nawet jeśli istnieją inne, które tego nie robią (na przykład prowadzą do

stanów NIE lub do nieskończone pętle). Tak więc to, co naprawdę się dzieje, polega na tym, że maszyna rozważa wszystkie możliwe ścieżki obliczeniowe, mówiąc „tak", jeśli przynajmniej jedna z nich daje „tak" i „nie" w przeciwnym razie. Tutaj również, być może nieco zaskakująco, nie zyskuje się zdolności rozwiązywania. Nawet to „magiczne" pojęcie obliczeń nie pozwala nam rozwiązać żadnych problemów algorytmicznych, których bez tego nie można byłoby rozwiązać.

Składanie na nieskończonej taśmie: przykład

Jak wyjaśniono wcześniej, przez lata sugerowano dziesiątki różnych modeli obliczeniowych, często o radykalnie odmiennym charakterze, i wszystkie okazały się równoważne, dostarczając w ten sposób ważkich dowodów prawdziwości tezy o TK. Jak ustalamy takie równoważności? Jak pokazać, że nawet dwa podobne warianty modelu maszyny Turinga (nie mówiąc już o dwóch zupełnie różnych modelach) powodują powstanie tej samej klasy rozwiązywalnych problemów? Odpowiedź tkwi w pojęciu symulacji, w której pokazujemy, że dla każdej maszyny jednego typu istnieje równoważna maszyna drugiego. Innymi słowy, pokazujemy, jak symulować jeden typ maszyny na innym. Załóżmy na przykład, że chcemy pokazać, że maszyny z dwukierunkową nieskończoną taśmą nie są potężniejsze niż te z taśmą, która jest nieskończona tylko po prawej stronie. Załóżmy, że otrzymaliśmy maszynę, która wykorzystuje dwukierunkową nieskończoną taśmę. Możemy skonstruować równoważną nową maszynę, której jednostronna taśma jest „postrzegana" jako dwustronna taśma złożona na dwie połączone połówki. Rysunek



pokazuje zgodność między taśmami. Maszyna symulująca najpierw rozłoży dane wejściowe, tak aby ich zawartość była w kwadratach o numerach nieparzystych, a reszta zawierała puste miejsca. Ta pusta część będzie odpowiadać lewej, całkowicie pustej części symulowanej taśmy, zagiętej. Nowa maszyna będzie wtedy symulować starą, ale poruszającą się o dwa kwadraty na raz, tak jakby znajdowała się na prawej części oryginalnej taśmy, aż dotrze do kwadratu znajdującego się najbardziej po lewej stronie. Następnie „zmienia bieg", również poruszając się o dwa pola na raz, ale po polach parzystych, tak jakby znajdował się na lewej części oryginalnej taśmy. Dokładne szczegóły są nieco żmudne i zostały tutaj pominięte, ale koncepcyjnie symulacja jest dość prosta. Inne symulacje mogą być dość skomplikowane, nawet koncepcyjne. We wszystkich jednak przypadkach istnieją techniki symulacyjne; dlatego istnieją maszyny Turinga do rozwiązywania każdego problemu, który można rozwiązać nawet na najbardziej wyrafinowanych nowoczesnych komputerach.

Zwijanie nieskończonej taśmy: przykład

Jak wyjaśniono wcześniej, przez lata sugerowano dziesiątki różnych modeli obliczeniowych, często o radykalnie odmiennym charakterze, i wszystkie okazały się równoważne, dostarczając w ten sposób ważkich dowodów prawdziwości tezy o TK. Jak ustalamy takie równoważności? Jak pokazać, że nawet

dwa podobne warianty modelu maszyny Turinga (nie mówiąc już o dwóch zupełnie różnych modelach) powodują powstanie tej samej klasy rozwiązywalnych problemów? Odpowiedź tkwi w pojęciu symulacji, w której pokazujemy, że dla każdej maszyny jednego typu istnieje równoważna maszyna drugiego. Innymi słowy, pokazujemy, jak symulować jeden typ maszyny na innym. Załóżmy na przykład, że chcemy pokazać, że maszyny z dwukierunkową nieskończoną taśmą nie są potężniejsze niż te z taśmą, która jest nieskończona tylko po prawej stronie. Załóżmy, że otrzymaliśmy maszynę, która wykorzystuje dwukierunkową nieskończoną taśmę. Możemy skonstruować równoważną nową maszynę, której jednostronna taśma jest „postrzegana" jako dwustronna taśma złożona na dwie połączone połówki. Rysunek poniżej pokazuje zgodność między taśmami. Maszyna symulująca najpierw rozłoży dane wejściowe, tak aby ich zawartość była w kwadratach o numerach nieparzystych, a reszta zawierała puste miejsca. Ta pusta część będzie odpowiadać lewej, całkowicie pustej części symulowanej taśmy, zagiętej. Nowa maszyna będzie wtedy symulować starą, ale poruszającą się o dwa kwadraty na raz, tak jakby znajdowała się na prawej części oryginalnej taśmy, aż dotrze do kwadratu znajdującego się najbardziej po lewej stronie. Następnie „zmienia bieg", również poruszając się o dwa pola na raz, ale na parzyste kwadraty, jakby znajdowały się po lewej stronie oryginalnej taśmy. Dokładne szczegóły są nieco żmudne i zostały tutaj pominięte, ale koncepcyjnie symulacja jest dość prosta. Inne symulacje mogą być dość skomplikowane, nawet koncepcyjne. We wszystkich jednak przypadkach istnieją techniki symulacyjne; dlatego istnieją maszyny Turinga do rozwiązywania każdego problemu, który można rozwiązać nawet na najbardziej wyrafinowanych nowoczesnych komputerach.

Programy kontrujące: kolejny bardzo podstawowy model

Na pierwszy rzut oka nie ma powodu, aby wybrać model maszyny Turinga ponad wszystkie inne jako model, o którym wprost wspomina teza CT. Teza mogła mówić o modelu leżącym u podstaw dużego komputera IBM lub Cray. W rzeczywistości jedno z najbardziej uderzających sformułowań w tezie w ogóle nie wymienia żadnego konkretnego modelu, a raczej stwierdza po prostu, że wszystkie komputery i wszystkie języki programowania są równoważne pod względem mocy obliczeniowej, biorąc pod uwagę nieograniczony czas i przestrzeń pamięci. Jednak, jak zobaczymy później, istnieją techniczne powody do badania niezwykle prymitywnych modeli. W związku z tym opiszemy teraz model programu licznika, który jest kolejnym z naprawdę prostych, ale uniwersalnych modeli obliczeń. Zamiast docierać do nich, zaczynając od ogólnych algorytmów i upraszczając rzeczy, najpierw zdefiniujemy same programy liczników, a następnie spróbujemy zobaczyć, jak odnoszą się one do maszyn Turinga. Program licznika może manipulować nieujemnymi liczbami całkowitymi przechowywanymi w zmiennych. Model lub język dopuszcza tylko trzy rodzaje podstawowych operacji na zmiennych, interpretowanych w standardowy sposób (gdzie, zgodnie z konwencją, Y − 1 jest zdefiniowane jako 0, jeśli Y wynosi już 0):

X ←; 0, X ←Y + 1 i X ←Y – 1

Zmienne nazywane są licznikami, ponieważ ograniczone operacje umożliwiają im w zasadzie tylko liczenie. Struktury kontrolne programu licznikowego obejmują proste sekwencjonowanie i warunkową instrukcję goto:

if X = 0 goto G

gdzie X jest zmienną, a G jest etykietą, która może być dołączona do instrukcji. Program licznika jest po prostu skończoną sekwencją opcjonalnie oznaczonych instrukcji. Wykonanie odbywa się w kolejności, jedna instrukcja na raz, rozgałęziając się do określonej instrukcji, gdy napotkane zostanie goto, a odpowiednia zmienna ma rzeczywiście wartość zero. Program licznika zatrzymuje się, jeśli i kiedy próbuje wykonać nieistniejącą instrukcję, osiągając koniec sekwencji lub próbując przejść do

nieistniejącej etykiety. Oto program licznika, który oblicza X × Y , iloczyn znajdujący się w Z po zakończeniu:

U ← 0

Z ← 0

A : if X = 0 goto G

X ← X − 1

V ← T + 1

V ← V − 1

B : if V = 0 goto A

V ← V − 1

Z ← Z + 1

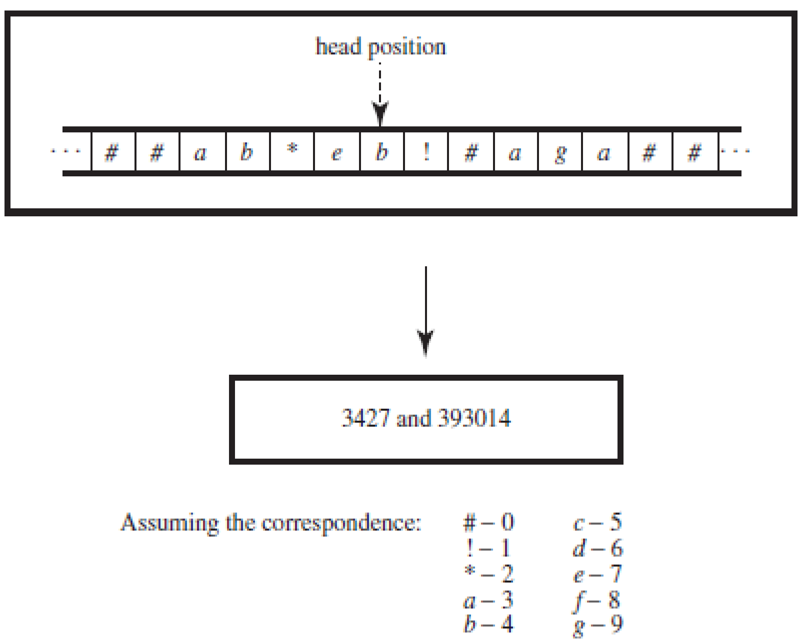
if U = 0 goto B

Możesz przez chwilę zastanowić się nad tym programem. Po pierwsze, istnieje goto G, ale nie ma instrukcji oznaczonej literą G. Jest to zgodne z naszą konwencją, a program zatrzymuje się w normalny sposób, gdy próbuje wykonać to goto. Użyliśmy również dwóch małych sztuczek, osiągając odpowiednio: efekt instrukcji V ← Y i instrukcji bezwarunkowej goto b, z których oba nie są tak naprawdę dozwolone przez formalną składnię. Iloczyn X × Y jest obliczany przez dwie zagnieżdżone pętle. Zewnętrzna pętla wielokrotnie dodaje Y do początkowo zerowanego Z, X razy, a wewnętrzna wykonuje dodawanie, wielokrotnie dodając 1 do Z, V razy, gdzie V jest inicjowane do Y za każdym razem. Może trudne, ale działa. Jak potężne są programy licznikowe? Czy potrafią rozwiązać naprawdę skomplikowane problemy? Odpowiedź brzmi: są dokładnie tak potężne, jak maszyny Turinga, a zatem tak samo potężne, jak jakikolwiek inny komputer.

Maszyny Turinga a programy liczników

Ponieważ programy licznikowe manipulują tylko liczbami, ta ostatnia instrukcja wymaga wyjaśnienia. Może mieć sens twierdzenie, że programy licznikowe mogą rozwiązywać problemy numeryczne, które są rozwiązywane przez maszyny Turinga (zwłaszcza, gdy widziałem, jak maszyny Turinga z trudem dodają liczby). Ale jak na przykład program licznikowy znajduje najkrótsze ścieżki na wykresie lub wystąpienia słowa „pieniądze" w tekście? Maszyny Turinga są zdolne do tych wyczynów, ponieważ, jak pokazano wcześniej, każdy rodzaj struktury danych, w tym wykresów i tekstów, może być zakodowany jako sekwencja symboli na taśmie. Ale czy takie obiekty mogą być zakodowane również jako liczby, którymi można manipulować za pomocą tylko jednostopniowych przyrostów i ubytków? Odpowiedź brzmi tak. Gdyby alfabet użyty w tych ciągach zawierał dokładnie 10 symboli, moglibyśmy łatwo skojarzyć je z 10 cyframi dziesiętnymi, a wtedy nie mielibyśmy problemu z odczytaniem dowolnego (skończonego) ciągu jako liczby. Korzystając ze standardowych metod, to samo można zrobić dla alfabetu o dowolnym (skończonym) rozmiarze, ponieważ nieujemne liczby całkowite mogą być reprezentowane w jednolity i jednoznaczny sposób przy użyciu dowolnej stałej liczby cyfr. Liczby binarne składają się tylko z dwóch cyfr, a te zawierające 16 nazywane są liczbami szesnastkowymi. Tak więc, korzystając z łatwo programowalnego mechanizmu kodowania, dowolna skończona sekwencja symboli w skończonym alfabecie może być postrzegana jako liczba. Aby zobaczyć, jak taśma maszyny Turinga może być postrzegana jako składająca się z liczb, przypomnij sobie, że w dowolnym momencie podczas wykonywania maszyny Turinga (zakładając, że zaczyna się na skończonym wejściu) tylko

skończona część taśmy zawiera niepuste informacje; reszta jest pusta. W konsekwencji tę znaczną część taśmy, wraz z położeniem głowicy, można po prostu przedstawić za pomocą dwóch liczb, kodujących dwie części taśmy leżące po obu stronach głowicy maszyny. Aby było łatwiej, dobrze jest przedstawić prawą część w odwrotnej kolejności, tak aby najmniej znaczące cyfry obu liczb znajdowały się blisko głowy. Rysunek



przedstawia numeryczną reprezentację taśmy, dla uproszczenia założono alfabet składający się z 10 symboli. Na tej podstawie można przeprowadzić symulację dowolnej maszyny Turinga z programem licznikowym.

Jak? Cóż, dwie zmienne są używane do przenoszenia dwóch kluczowych wartości kodujących niepuste części taśmy po obu stronach głowicy i (domyślnie) samej pozycji głowicy; inna zmienna jest używana dla stanu. Subtelność symulacji polega na tym, że efekt jednego kroku maszyny Turinga jest dość „lokalny". Jedna strona taśmy staje się „dłuższa", ponieważ głowa oddala się od niej, dodając na jej „końcu" nowy symbol, a druga staje się krótsza, tracąc swój ostatni symbol (chyba że maszyna przesunie się z niepustej części do całości pusty obszar, w którym to przypadku strona, która miała stać się pusta, otrzymuje za darmo nowy pusty symbol, w efekcie wydłużając całą odpowiednią część taśmy o jeden symbol). Wszystkie te zmiany można zasymulować w programie licznika przez stosunkowo prostą manipulację arytmetyczną dwóch głównych zmiennych.

W konsekwencji cały diagram przejścia maszyny Turinga można „przepracować" w programie licznika za pomocą „kawałków", z których każdy symuluje jedno przejście diagramu. Aby faktycznie zasymulować działanie maszyny, program wielokrotnie sprawdza zmienną przenoszącą stan i symbol widziany przez głowicę (czyli najmniej znaczącą cyfrę liczby po prawej stronie), wykonuje odpowiednią porcję i zmienia wartość zmiennej stanu. Tak więc programy, które mogą jedynie zwiększać i zmniejszać liczby całkowite o 1 i testować ich wartość względem 0, mogą być używane do robienia wszystkiego, co może zrobić każdy komputer. Mogą nie tylko obliczać numerycznie, ale w zasadzie mogą również reprezentować, przemierzać i manipulować dowolnym rodzajem struktury danych, w tym listami, wykresami, drzewami, a nawet całymi bazami danych. Jeśli chodzi o drugi kierunek, a mianowicie, że maszyny Turinga mogą robić wszystko, co potrafią programy liczników, możemy symulować programy liczników za pomocą maszyn Turinga w następujący sposób. Wartości różnych

liczników są rozłożone na taśmie, oddzielone znakami „*". Maszyna symulująca używa stanów specjalnych do reprezentowania różnych instrukcji w programie. Wejście w każdy taki stan wyzwala sekwencję przejść, która sama wykonuje instrukcję. Ponownie pominięto szczegóły. Być może zainteresuje minimalistów wśród nas, że zawsze możemy zrobić tylko z dwoma licznikami. Możliwe jest symulowanie dowolnego programu licznikowego z takim, który używa tylko dwóch zmiennych, chociaż ta symulacja jest bardziej skomplikowana. Zarówno maszyny Turinga, jak i programy licznikowe osiągają uniwersalność, wykorzystując potencjalnie nieskończoną ilość pamięci, ale na różne sposoby. W przypadku maszyn Turinga liczba obiektów zawierających informacje (kwadraty taśmy) jest potencjalnie nieograniczona, ale ilość informacji w każdym z nich jest skończona i ograniczone. W przypadku programów licznikowych jest odwrotnie. W danym programie istnieje tylko skończenie wiele zmiennych, ale każda z nich może zawierać dowolnie dużą wartość, w efekcie kodując potencjalnie nieograniczoną ilość informacji.

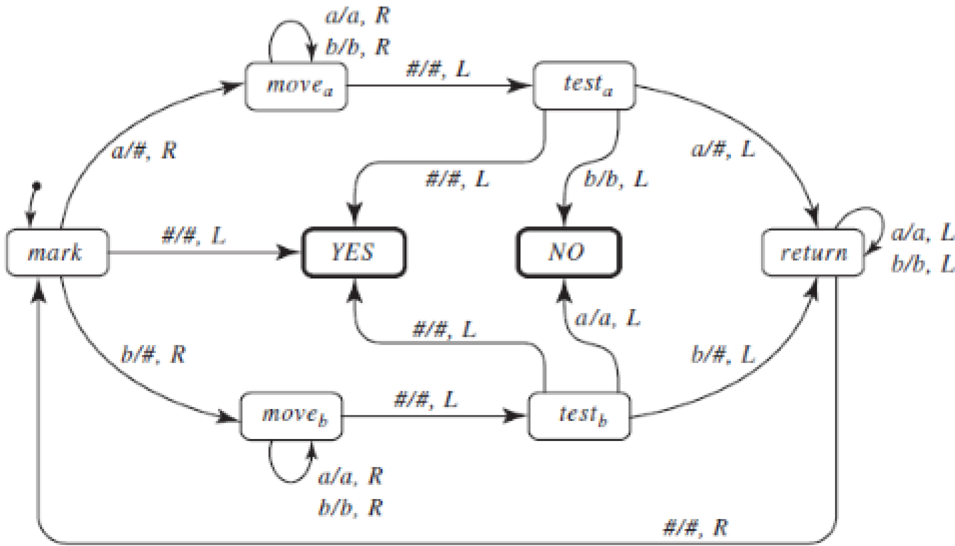
Symulacje jako redukcje

Kiedy mówimy, że jeden model obliczeniowy może symulować inny, tak naprawdę mówimy, że mamy redukcję (w sensie części 8) między tymi dwoma modelami. Ten punkt widzenia zapewnia solidną podstawę matematyczną dla niektórych dyskusji we wcześniejszych rozdziałach. Ilekroć mówiliśmy o programach, które akceptują inne programy jako dane wejściowe, wcale nie byliśmy restrykcyjni. Ponieważ skuteczne symulacje istnieją między dowolnymi dwoma wystarczająco potężnymi modelami obliczeniowymi. Ograniczenia i odporność programu lub algorytmu w dowolnym języku lub modelu i przetłumacz go na język, z którym akurat pracujemy. Rozważmy na przykład nierozstrzygalność problemu zatrzymania, jak wykazano w Części 8. Właściwe wykorzystanie redukcji związanych z tezą CT pozwala pokazać, że wynik ten jest niezwykle ogólny. Można to najpierw rygorystycznie udowodnić dla maszyn Turinga, (1) zakładając, że program wejściowy W jest opisem jakiejś maszyny Turinga, oraz (2) konstruując hipotetyczny program S również jako maszynę Turinga (dość łatwa adaptacja podanej konstrukcji w nim). Następnie ustala się sprzeczny charakter tej konstrukcji, co prowadzi do pozornie skromnego wniosku, że nie istnieje maszyna Turinga, która rozwiązuje problem zatrzymania maszyn Turinga. Ten wynik nie jest jednak wcale skromny. W rzeczywistości dowodzi to, że problem zatrzymania jest nierozstrzygalny w bardzo silnym sensie: żaden efektywny język lub model L₁ nie jest w stanie rozwiązać problemu zatrzymania programów w języku uniwersalnym lub modelu L₂. (Szczególny przypadek ma miejsce, gdy L₁ i L₂ są jednym i tym samym językiem.) Dzieje się tak dlatego, że jeśli dla niektórych uniwersalnych L₁ i L₂ ta wersja problemu zatrzymania byłaby rozstrzygalna, tak samo byłaby wersja maszyny Turinga. (Czy widzisz dlaczego?) To właśnie te niezależne od języka i modelu fakty, w połączeniu z zaufaniem, jakim darzymy tezę Kościoła/Turinga, uzasadniają użycie w poprzednich częściach zwrotów dotyczących nieistnienia, dla pewnych problemów, jakichkolwiek algorytmy, napisane w dowolnych językach i działające na dowolnych komputerach, teraz lub w przyszłości.

Uniwersalne algorytmy

Teza CT mówi o uniwersalnych modelach, czyli językach. Jedną z jego najciekawszych konsekwencji jest istnienie uniwersalnych algorytmów. Algorytm uniwersalny może zachowywać się jak każdy inny algorytm. Akceptuje jako dane wejściowe opis dowolnego algorytmu A i dowolne legalne dane wejściowe X i po prostu uruchamia lub symuluje A na X, zatrzymując się, jeśli i kiedy A się zatrzyma, i wytwarzając wyniki, które zostałyby utworzone, gdyby A rzeczywiście został uruchomiony na X Tak więc ustalenie algorytmu wejściowego A i pozwolenie X na zmianę skutkuje tym, że algorytm uniwersalny będzie zachowywał się dokładnie tak, jak A. W pewnym sensie komputer lub interpreter jest bardzo podobny do algorytmu uniwersalnego: przedstawiamy go za pomocą programu i wejście i

uruchamia pierwszy na drugim. Jednak termin „uniwersalny” sugeruje niezależność, co oznacza, że uniwersalny algorytm powinien być niewrażliwy na wybór języka lub maszyny, w przeciwieństwie do komputerów i interpretatorów. Wydawałoby się zatem, że żaden uniwersalny algorytm nie mógłby być kiedykolwiek zaimplementowany, ponieważ zarówno sam uniwersalny algorytm, jak i jego algorytmy wejściowe muszą być napisane w jakimś języku, przeznaczonym dla jakiejś maszyny. Z pomocą przychodzi nam ponownie teza CT, z jej symulacjami między modelami i twierdzeniem, że wszystkie języki programowania i modele obliczeniowe są równoważne. Aby uzyskać uniwersalny algorytm, wystarczy użyć języka L₁, aby napisać efektywnie wykonywalny program U, który akceptuje jako dane wejściowe dowolny program napisany w jakimś ustalonym uniwersalnym języku lub modelu L₂ oraz dowolne dane wejściowe i symuluje działanie tego programu na tym Wejście. Po napisaniu U można uznać za niezależne od języka i maszyny, ponieważ zgodnie z tezą (1) można go było napisać w dowolnym uniwersalnym języku, działającym na dowolnej maszynie, oraz (2) może symulować każdy efektywnie wykonywalny algorytm, napisane w dowolnym języku. Oznacza to, że mając algorytm A i wejście X, przepisz A w języku L2 (jest to możliwe dzięki pracy magisterskiej) i prześlij nowy program za pomocą X do U. Maszyny Turinga są idealnym kandydatem zarówno dla L₁, jak i L₁, i rzeczywiście nie jest zbyt trudno skonstruować tzw. uniwersalną maszynę Turinga, czyli taką, która potrafi symulować wpływ dowolnych maszyn Turinga na dowolne dane wejściowe. Ale aby to zrobić, musimy najpierw znaleźć sposób na opisanie dowolnej maszyny Turinga jako liniowej sekwencji symboli, odpowiedniej do wprowadzenia na taśmę. W zasadzie pozostaje nam tylko opisać schemat przejścia maszyny, który musi zostać zlinearyzowany w ciąg symboli, nadający się do zapisania na taśmie maszyny Turinga. Można to łatwo zrobić, ponieważ każde przejście może być podane przez jego stan źródłowy i docelowy, po których następuje etykieta ⟨a/b, L⟩ lub ⟨a/b, R⟩. Umownie lista przejść będzie poprzedzona nazwą stanu startowego. Oto na przykład początkowa część kodowania maszyny Turinga z rysunku



mark ∗∗ mark YES ⟨ #/#, L ⟩ ∗ mark movea ⟨ a/#, R ⟩ ∗ movea movea ⟨ a/a, R ⟩ ∗ …

Uniwersalna maszyna Turinga U przyjmuje jako dane wejściowe taki opis dowolnej maszyny Turinga M, po której następuje sekwencja skończona X, która jest postrzegana jako potencjalne wejście do M (opis M i wejście X są oddzielone, powiedzmy, „$”) . Następnie przystępuje do symulacji działania M na taśmie zawierającej wejście X otoczone odstępami, z tymi samymi konsekwencjami: gdyby M nie zatrzymał się na takiej taśmie, to także U nie, a jeśli tak, to U Co więcej, jeśli i kiedy U zatrzyma się, taśma wygląda dokładnie tak, jak wyglądałaby na zakończeniu M, wliczając w to wyjście ujęte między

„!”. Naprawdę skonstruowanie uniwersalnej maszyny Turinga jest interesującym ćwiczeniem, podobnie jak zadanie napisania interpretera dla prostego języka programowania L w samym języku L. W rzeczywistości istnieją niezwykle zwięzłe, uniwersalne maszyny Turinga o bardzo niewielu stanach. W podobnym duchu można oczywiście skonstruować uniwersalny program licznikowy, który przyjmuje jako dane wejściowe dwie liczby, pierwsza koduje program licznika wejściowego, a druga jego potencjalne wejście i symuluje jedną na drugiej. W rzeczywistości, będąc po prostu kolejnym programem licznika, uniwersalny program licznika może zatem być skonstruowany tylko z dwoma licznikami, jak już wspomniano. Ważniejsza od liczby stanów czy liczników jest jednak głęboka natura uniwersalnego algorytmu lub machineU. Raz skonstruowany U jest pojedynczym obiektem o maksymalnej mocy algorytmicznej, a jego znaczenia nie sposób przecenić. Gdyby napisać uniwersalny algorytm do użytku na komputerze osobistym, byłby on w stanie dosłownie zasymulować nawet największy i najbardziej wyrafinowany komputer typu mainframe, pod warunkiem, że miałby wystarczająco dużo czasu, aby jako pierwsze dane wejściowe podano opis symulowanej maszyny i że dostępna jest wystarczająca liczba jednostek pamięci.

Niewielka modyfikacja programów licznikowych

Dokonamy teraz niewielkiej zmiany w pierwotnych instrukcjach programów licznikowych; przyczyna zostanie wyjaśniona w następnej sekcji. Jak zdefiniowano wcześniej, programy licznikowe mogą tylko dodawać lub odejmować 1 od licznika. Tak więc, jeśli mają pracować na liczbach reprezentujących taśmy maszyny Turinga, musieliby to robić pojedynczo, podczas gdy maszyny Turinga pracują na swoich taśmach po jednym symbolu na raz. A ponieważ taśma maszyny Turinga jest uważana za reprezentującą liczby w, powiedzmy, zapisie dziesiętnym, numery maszyny Turinga są manipulowane jedną cyfrą na raz, co jest wykładniczo bardziej wydajne niż praca nad nimi pojedynczo (chyba że Maszyna Turinga działa na jednoliterowym alfabecie, co jest bardzo nieciekawym przypadkiem). I tak, chociaż programy licznikowe oparte na samych instrukcjach +1 i -1 są rzeczywiście tak potężne, jak maszyny Turinga, są wykładniczo wolniejsze. Nie dzieje się tak z powodu pewnych nieodłącznych ograniczeń samego modelu programu kontrującego, ale dlatego, że instrukcje pierwotne są wykładniczo słabsze. Aby usunąć tę rozbieżność, programy liczników muszą mieć możliwość manipulowania całymi cyframi, zarówno binarnymi, dziesiętnymi, jak i innymi. (Dlaczego szczególny wybór podstawy liczbowej jest tutaj nieistotny?) Zasadniczo programy mają mieć możliwość pracy na liczbach niejednoznacznych i muszą mieć możliwość dołączania i odłączania cyfr do liczb w stałym czasie. W związku z tym dodajmy do repertuaru podstawowych operacji programów licznikowych dwie instrukcje:

$X \leftarrow X \times 10$ i $X \leftarrow X/10$

(Umownie, operator dzielenia ignoruje ułamki). Nowe operacje wyraźnie nie dodają mocy obliczeniowej do modelu, ponieważ mogły być symulowane przez operacje +1 i -1. Umożliwiają one jednak porównywanie maszyn Turinga i programów licznikowych w bardziej realistyczny sposób, co zostanie teraz pokazane.

Podatność jest również solidna

Wprowadzając redukcje pomiędzy wszystkimi wystarczająco potężnymi modelami obliczeń, przekonujemy się, że klasa problemów rozwiązywanych przez te modele jest niewrażliwa na różnice między nimi. Oznaczając problemy w klasie jako obliczalne (lub rozstrzygalne, jeśli interesuje nas przypadek tak/nie), wyrażamy naszą opinię, że pojęcie, które zdobyliśmy, jest ważne i głębokie. To jest sedno tezy Churcha/Turinga. Jednak przy niewielkim wysiłku możemy zrobić jeszcze lepiej. Dokładna inspekcja pokazuje, że jeśli oba modele biorące udział w takiej redukcji radzą sobie z liczbami (lub jakąkolwiek ich reprezentacją wspieraną przez model) w sposób niejednostajny, to wszystkie te

redukcje zajmują czas wielomianowy; to znaczy, są rozsądne w sensie Części 7. Na przykład opisana wcześniej transformacja z maszyny Turinga i jej wejście do równoważnego programu licznika i odpowiadające mu wejście zajmuje czas, który jest tylko wielomianem długości opisów tego pierwszego. Co więcej - i fakt ten nie jest prawdziwy bez zezwolenia na operacje X × 10 i X/10 - czas, w którym wynikowy program licznika działa na przekształconym wejściu (przy założeniu, że się zatrzymuje) jest co najwyżej wielomianowo dłuższy niż czas maszyna Turinga musiałaby działać na oryginalnym wejściu. Wynika z tego oczywiście, że jeśli maszyna Turinga rozwiąże jakiś problem algorytmiczny w czasie wielomianowym, to nie tylko odpowiedni program licznika rozwiąże ten problem, ale także rozwiąże to w czasie wielomianowym. Odwrotna redukcja, od programów liczników do maszyn Turinga, jest również wielomianem, tak więc słuszny jest również podwójny fakt: jeśli program licznika rozwiązuje jakiś problem w czasie wielomianu, to samo dzieje się z powstałą równoważną maszyną Turinga. Wniosek jest taki, że maszyny Turinga i programy licznikowe (z instrukcjami X × 10 i X/10) są wielomianowo równoważne. Klasa problemów mających rozwiązania sensowne (tj. wielomianowe) jest taka sama dla obu modeli. Naprawdę zaskakujący jest fakt, że ta równoważność wielomianowa odnosi się nie tylko do redukcji między tymi bardzo prymitywnymi modelami, ale także do redukcji między nimi, a nawet do najbardziej wyrafinowanych modeli. Maszyny Turinga i programy licznikowe są oczywiście bardzo nieefektywne, nawet w przypadku dość trywialnych zadań, wymagających przemieszczania się tam i z powrotem na taśmie lub wielokrotnego zwiększania i zmniejszania liczników. Jednak są one tylko wielomianowo mniej wydajne niż nawet najszybsze i najbardziej skomplikowane komputery, które obsługują najbardziej zaawansowane języki programowania z najbardziej wyrafinowanymi kompilatorami. Przy rozwiązywaniu jakiegoś problemu algorytmicznego, maszyna Turinga lub program licznikowy wynikający z odpowiedniej redukcji może zająć dwa razy więcej czasu niż szybki komputer, lub tysiąc razy więcej, lub nawet taką ilość czasu do kwadratu, sześcianu lub podniesienia do moc 1000, ale nie tak wykładniczo.

Mówiąc bardziej zwięźle, jeśli szybki komputer rozwiąże pewien problem w czasie O(f(N)), dla pewnej funkcji f o długości wejściowej N, to istnieje równoważna maszyna Turinga, która zajmie nie więcej niż czas O(p(f(N))), dla pewnej ustalonej funkcji wielomianowej p. W szczególności, jeśli f samo w sobie jest wielomianem, co oznacza, że szybki komputer rozwiązuje problem rozsądnie, to jakaś bardzo prymitywnie wyglądająca maszyna Turinga również rozwiązuje go rozsądnie — w czasie wielomianu p(f(N)), ściślej. Zatem czas może wzrosnąć od, powiedzmy, $O(N^2)$ do $O(N^5)$ lub O(N⁸⁵), ale nie do $O(2^N)$. W rzeczywistości większość znanych redukcji obejmuje wielomiany niższego rzędu nie większe niż około N^4 lub N^5, tak że „dobre" algorytmy wielomianowe w jednym modelu nie staną się niedopuszczalnie gorsze w innym. W przypadku programów licznikowych czas mierzony jest liczbą wykonanych instrukcji, a dla maszyn Turinga liczbą wykonanych kroków. Wniosek jest taki: nie tylko klasa problemów obliczalnych jest odporna (tj. niewrażliwa na model lub język), ale także klasa problemów wykonalnych. Jest to naprawdę udoskonalenie tezy CT, które uwzględnia również czas wykonania. Powinniśmy zauważyć, że udoskonalona teza nie dotyczy niektórych modeli, które zawierają nieograniczoną liczbę współbieżności, jak wyjaśniono w Części 10, i z tego powodu jest czasami nazywana tezą obliczeń sekwencyjnych. Termin „sekwencyjny" ma na celu uchwycenie algorytmicznych egzekucji, które przebiegają w sposób sekwencyjny, krok po kroku, a nie poprzez wykonywanie wielu rzeczy jednocześnie. Udoskonalona teza CT twierdzi zatem, że wszystkie sekwencyjne uniwersalne modele obliczeń, w tym te, które jeszcze nie zostały wynalezione, wykazują wielomianowe zachowania czasowe, tak że klasa problemów możliwych do rozwiązania w rozsądnym czasie jest taka sama dla wszystkich modeli. Wyrafinowana teza uzasadnia więc kolejną z linii pojawiających się w naszej sferze problemów algorytmicznych, oddzielającą to, co wykonalne od niewykonalnego. Wiele innych klas złożoności, takich jak NP, PSPACE i EXPTIME, jest również solidnych w tym samym sensie, co uzasadnia wiele niezależnych od języka i modeli badań w teorii złożoności.

Jednak niektóre klasy, takie jak problemy rozwiązywalne w czasie liniowym, nie są i mogą być bardzo wrażliwe na zmiany w modelu. Na przykład maszyna Turinga z dodatkowym licznikiem może porównać liczbę a i b pojawiających się w sekwencji w czasie liniowym, ale same maszyny Turinga wymagają O(N × log N), jeśli zastosuje się raczej sprytną metodę, i przyjmują kwadratową czas, jeśli naiwnie biegają tam i z powrotem. (Czy możesz znaleźć metodę O(N × log N)?) Nawiasem mówiąc, wielomianowa wersja tezy o TK nie mogła zostać sformułowana już w latach 30. XX wieku, ponieważ nie istniała wtedy teoria złożoności. Znaczenie klasy P zostało dostrzeżone dopiero pod koniec lat sześćdziesiątych i zyskało wiarygodność kilka lat później, gdy uświadomiono sobie wagę problemu P vs. NP.

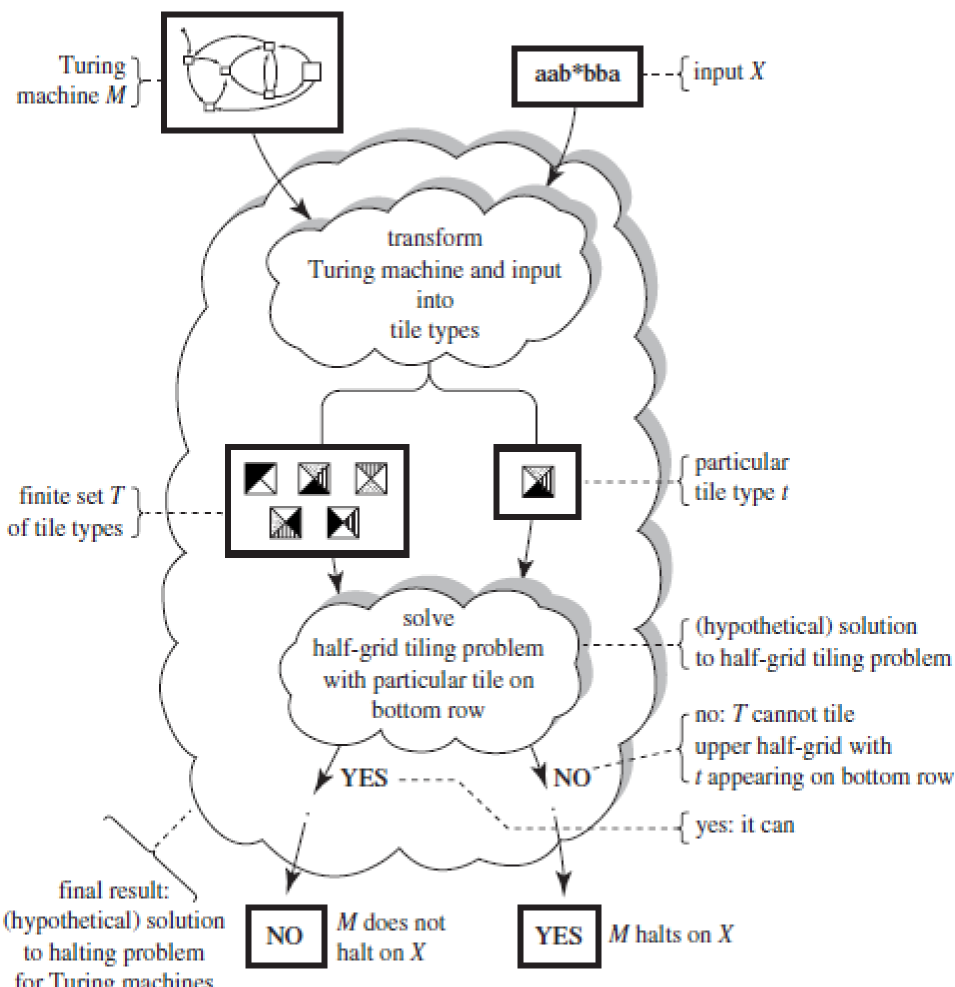
Maszyny Turinga i problem P vs. NP.

W tym miejscu na szczególną wzmiankę zasługują problemy NP-zupełne z Części 7. W większości podręczników P i NP są wprowadzane w terminach rygorystycznego pojęcia obliczeń maszyny Turinga; NP jest zdefiniowane tak, aby zawierało dokładnie te problemy decyzyjne, które można rozwiązać przez niedeterministyczne maszyny Turinga działające w czasie wielomianowym, podczas gdy P jest zdefiniowane jako zawierające te, które można rozwiązać przez zwykłe maszyny Turinga w czasie wielomianowym. Po podaniu takich definicji form, udoskonalona teza CT stwierdza, że wszystkie rozsądne modele obliczeń sekwencyjnych są wielomianowo równoważne zwykłym maszynom Turinga, a następnie można omówić konsekwencje tego faktu. Natomiast pojęcia P i NP wprowadziliśmy w rozdziale 7 bez dokładnego modelu, najpierw podając tezę CT, a dopiero później wprowadzając modele formalne, takie jak maszyny Turinga. Przyczyny tego mają charakter pedagogiczny; konsekwencje pozostają takie same. Teraz, gdyby niedeterministyczne maszyny Turinga spełniały kryterium sekwencyjności, udoskonalona teza sugerowałaby pozytywne rozwiązanie problemu P vs. NP, ponieważ zrównałaby klasy problemów rozwiązywalnych w czasie wielomianowym na obu wersjach maszyn Turinga. Tak się składa, że maszyny niedeterministyczne nie są uważane za sekwencyjne, ponieważ wykorzystują „magię" do dokonywania najlepszych wyborów, a bez magii musiałyby wypróbować wiele możliwości jednocześnie, aby znaleźć najlepszą. (Wypróbowanie ich sekwencyjnie zajęłoby czas wykładniczy.) Dlatego teza ta nie dotyczy takich maszyn, a zatem nie może wiele pomóc w odniesieniu do P vs. NP. Sformułowanie problemu P vs. NP w ten formalny sposób jest interesujące, ponieważ oznacza, że aby rozwiązać problem w sposób negatywny (to znaczy pokazać, że P ≠NP) wystarczy pokazać, że prosty i prymitywny model maszyn Turinga nie może rozwiązać jakiś problem NP-zupełny w czasie krótszym niż wykładniczy. Na przykład, jeśli ktoś miałby wykazać, że problemu małpiej łamigłówki nie można rozwiązać na żadnej maszynie Turinga w czasie wielomianowym, wynikałoby z udoskonalonej tezy, że nie można go rozwiązać w czasie wielomianowym na żadnym modelu sekwencyjnym, a zatem jest to naprawdę trudny. A jak wiemy, jeśli problem małpiej łamigłówki jest nierozwiązywalny, to tak samo są z wszystkimi problemami NP-zupełnymi, co daje P ≠ NP.

Używanie maszyn Turinga do dolnych wiązań próbnych

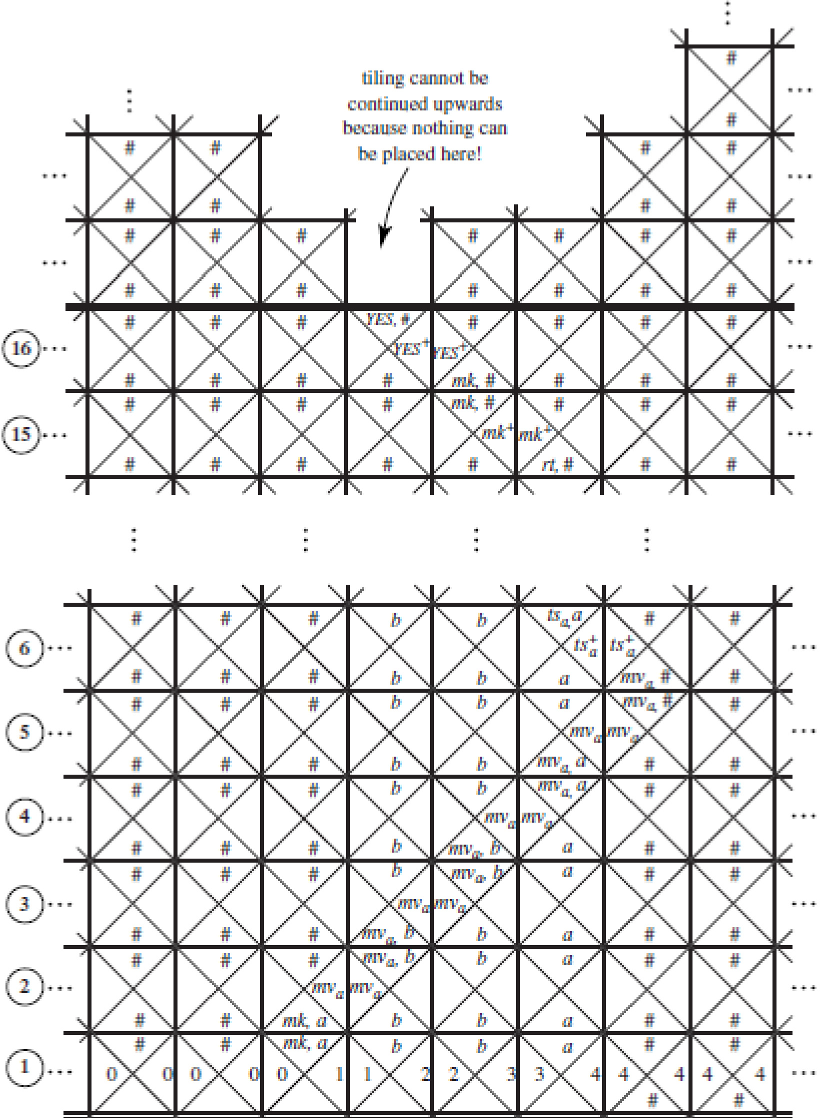
Aby udowodnić górną granicę problemu algorytmicznego - to znaczy znaleźć dobry algorytm - należy użyć najbogatszego i najpotężniejszego dostępnego formalizmu. W rzeczywistości naukowcy zazwyczaj stosują konstrukcje programistyczne bardzo wysokiego poziomu i skomplikowane struktury danych, aby opracowywać nietrywialne algorytmy. Następnie opierają się na solidności, jaką dają warianty tezy Churcha/Turinga, aby wysuwać twierdzenia o implikacjach tych algorytmów dla wszystkich modeli. Tak więc, jeśli komuś uda się kiedykolwiek udowodnić, że P = NP, istnieje prawdopodobieństwo, że zostanie to zrobione przy użyciu modelu bardzo wysokiego poziomu, z być może skomplikowanymi strukturami danych, w celu opisania złożonego algorytmu wielomianowego dla jakiegoś problemu NP-zupełnego. Jednak przy udowadnianiu niższych granic modele prymitywne są znacznie lepiej dopasowane, ponieważ wykorzystują tylko kilka niezwykle prostych konstrukcji i nie trzeba się tym martwić, że tak powiem. Tak więc, jeśli ktoś kiedykolwiek udowodni, że P ≠ NP, są szanse, że zostanie to wykonane

przy użyciu bardzo prymitywnego modelu, takiego jak maszyny Turinga lub programy licznikowe. Maszyny Turinga są w rzeczywistości szeroko stosowane we wszelkiego rodzaju dowodach z dolnym ograniczeniem. Na przykład pouczające jest wyczucie, w jaki sposób maszyny Turinga mogą być używane do wykazania nierozstrzygalności problemu kafelkowania. (Oczywiście nierozstrzygalność jest rodzajem dolnej granicy – przynosi złe wieści o stanie problemu.) Łatwiej jest omówić wersję problemu kafelkowania, która jest nieco mniej ogólna niż prosty nieograniczony problem domina z Części 8 Ta konkretna wersja problemu pyta, czy zestaw typów kafelków T może być użyty do ułożenia górnej połowy siatki nieskończonych liczb całkowitych, ale z dodatkowym wymaganiem, aby pierwszy kafelek w T , nazwijmy go t, miał być umieszczony gdzieś w dolnym rzędzie. Aby pokazać, że ten problem kafelkowania jest nierozstrzygalny, opisujemy redukcję problemu zatrzymywania się maszyn Turinga do niego (w rzeczywistości problem braku zatrzymywania, jak wyjaśniono poniżej). Innymi słowy, chcemy pokazać, że gdybyśmy mogli rozwiązać problem kafelkowania, moglibyśmy również zdecydować, czy dana maszyna Turinga M może zatrzymać się na danym wejściu X. Załóżmy zatem, że mamy hipotetyczne rozwiązanie opisanego problemu kafelkowania . Dostajemy teraz maszynę Turinga M i słowo wejściowe X. Chcielibyśmy pokazać, jak skonstruować zbiór typów płytek T , zawierający konkretną płytkę t, taką że M nie zatrzymuje się dokładnie na X, jeśli T może ułożyć górną pół-siatkę, przy czym t pojawia się gdzieś w dolnym rzędzie.



Nie będziemy opisywać szczegółów samej transformacji, a jedynie jej nadrzędną ideę. Zapraszamy do wypróbowania tych szczegółów. Pomysł jest bardzo prosty. Zestaw płytek T jest skonstruowany w taki sposób, że ułożenie półsiatki w górę odpowiada postępowi w obliczeniach maszyny Turinga M na X.

Efekt uzyskuje się poprzez zakodowanie każdego rzędu płytek za pomocą odpowiednich kolorów , bieżąca zawartość nieskończonej taśmy M, wraz z bieżącym stanem i położeniem głowicy na taśmie. (Ponieważ istnieje tylko skończenie wiele stanów i symboli, ich kombinacje mogą być zakodowane za pomocą skończonych wielu kolorów.) W ten sposób każde prawidłowe ułożenie kafelków w rzędzie reprezentuje prawidłową konfigurację (to znaczy pełny stan) maszyny Turinga. Co więcej, płytki są skonstruowane tak, aby zagwarantować, że przejście w górę z rzędu wyłożonego płytkami do rzędu wyłożonego płytkami jest możliwe tylko zgodnie z instrukcjami podanymi na schemacie przejścia M. Odbywa się to poprzez dopuszczenie w T tylko płytek, których dolne kolory (które jedynie „kopiują" poprzednią konfigurację M z poprzedniego rzędu płytek przez ograniczenie dopasowania kolorów) są powiązane z górnymi kolorami (które kodują nową konfigurację M) przez prawo przejścia w diagramie przejść M. W ten sposób możliwość przedłużenia części płytki w górę o jeszcze jeden rząd jest dokładnie tym samym, co możliwość wykonania jeszcze jednego kroku obliczeń M, osiągając nową konfigurację. Specjalny kafelek t, który musi pojawić się w dolnym rzędzie, jest skonstruowany w celu zakodowania stanu początkowego maszyny Turinga i początku słowa wejściowego X. Inne kafelki dolnego rzędu kodują resztę X, gwarantując, że pierwszy rząd w każdym możliwym ułożeniu płytek wiernie przedstawia początkową konfigurację maszyny. W ten sposób mapujemy obliczenia maszyny Turinga na kafelkowych częściach górnej połowy siatki, gdzie wymiar poziomy odpowiada przestrzeni pamięci (taśmie), a wymiar pionowy czasowi. Można więc śmiało powiedzieć, że kafelkowanie z kolorowymi kafelkami i obliczanie za pomocą algorytmów to prawie to samo. Rysunek pokazuje kafelkowanie, które wynika z zestawu kafelków skonstruowanego dla maszyny palindromowej z i sekwencji wejściowej „a bba.



W tym przykładzie układanie płytek nie może być kontynuowane w górę, ponieważ maszyna zatrzymuje się i nie może kontynuować obliczeń. Z tego wszystkiego wynika, że każde możliwe ułożenie całej górnej pół-siatki z typami płytek T , w których t pojawia się w dolnym rzędzie, odpowiada bezpośrednio nieskończonemu obliczeniu M na X. W konsekwencji, jeśli pół-siatka problem kafelkowania był rozstrzygalny, tj. gdybyśmy mogli zdecydować, czy T może ułożyć pół-siatkę z t pojawiającym się w pierwszym rzędzie, problem braku zatrzymania dla maszyn Turinga również byłby rozstrzygalny, jak pokazano na pierwszym rysunku. Ale ponieważ problem z zatrzymaniem nie jest rozstrzygalny, tak samo jak problem braku zatrzymania (dlaczego?), tak że w rezultacie problem kafelkowania również nie może być rozstrzygalny.

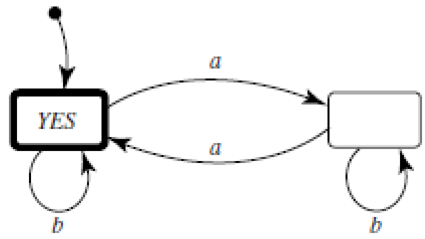
Jednokierunkowe maszyny Turinga lub automaty skończone

Widzieliśmy, że pewne ograniczenia dotyczące maszyn Turinga (takie jak używanie taśmy, która jest nieskończona tylko po prawej stronie) nie umniejszają uniwersalności modelu; klasa problemów, które można rozwiązać, pozostaje taka sama, nawet gdy model jest tak ograniczony. Oczywiście nie wszystkie ograniczenia mają tę właściwość. Maszyny, które muszą się zatrzymać natychmiast po uruchomieniu, nie mogą wiele zrobić, to samo dotyczy tych, którym w ogóle nie wolno się zatrzymać. Te przykłady nie są jednak zbyt interesujące. W międzyczasie można nałożyć wiele rodzajów ograniczeń na uniwersalne

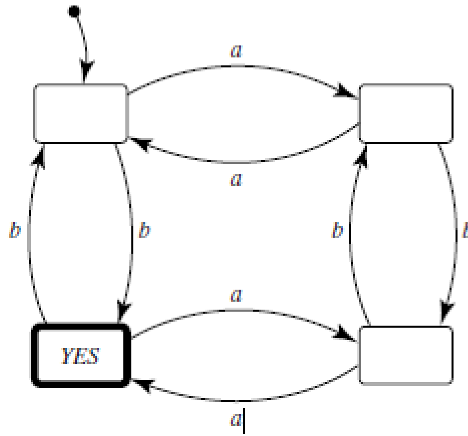
modele obliczeń, a w szczególności na maszyny Turinga, co skutkuje słabszymi klasami problemów, które jednak są bardzo interesujące. Jednym z oczywistych podejść jest ograniczenie wykorzystania zasobów maszyny. Wiemy już, że klasę P uzyskuje się, dopuszczając tylko maszyny Turinga, które zatrzymują się w określonym czasie wielomianu. PSPACE to klasa uzyskana przez umożliwienie maszynom Turinga dostępu tylko do wielomianowej ilości taśmy (każda próba przekroczenia limitu jest karana, powiedzmy, zatrzymaniem w stanie NO). Inne klasy złożoności można podobnie zdefiniować jako składające się z problemów, które można rozwiązać za pomocą odpowiednio ograniczonych zasobów maszyn Turinga. Istnieje jednak inne podejście do ograniczania modelu maszyny Turinga. Polega na ograniczeniu samego mechanizmu maszyny. Jednym z najciekawszych z tych obniżeń jest umożliwienie maszynom Turinga poruszania się wzdłuż taśmy tylko w jednym ustalonym kierunku - powiedzmy w prawo. Rezultatem jest urządzenie zwane automatem skończonym lub w skrócie automatem skończonym. Zastanówmy się nad tym przez chwilę. Jeśli maszyna nie może poruszać się w lewo, nie może skontrolować żadnego pola więcej niż raz. W szczególności nie może korzystać z niczego, co sam zapisuje, z wyjątkiem tworzenia danych wyjściowych, ponieważ nie może wrócić do czytania własnych notatek, że tak powiem. Dlatego, jeśli dyskusja ogranicza się do problemów decyzyjnych, które i tak nie dają wyników, możemy założyć, że automaty skończone w ogóle nie piszą; powiedzenie „tak” lub „nie” można osiągnąć poprzez dwa specjalne stany zatrzymania, a nowe symbole, które zapisuje w miarę poruszania się, są bezwartościowe i nie mają żadnego wpływu na ostateczny werdykt automatu. Co więcej, po prawej stronie sekwencji wejściowej taśma zawiera tylko puste miejsca; w ten sposób automat może równie dobrze zatrzymać się, gdy osiągnie koniec sekwencji wejściowej, ponieważ nie zobaczy niczego nowego, co miałoby znaczenie. Podsumowując, automat skończony rozwiązujący problem decyzyjny działa w następujący sposób. Po prostu przechodzi przez sekwencję wejściową symboli, jeden po drugim, zmieniając stany w wyniku bieżącego stanu i nowego symbolu, który widzi. Po osiągnięciu końca sekwencji wejściowej zatrzymuje się, a odpowiedź zależy od tego, czy zatrzymała się w stanie TAK, czy NIE. (Umownie rozważymy zatrzymanie się w jakimkolwiek innym stanie, aby było jak mówienie „nie”, aby automat skończony nie musiał mieć stanu NIE; jeśli jest w stanie TAK, gdy dochodzi do końca danych wejściowych, odpowiedź brzmi „tak”, w przeciwnym razie „nie”.) Możemy opisać automat skończony jako diagram przejść stanów, tak jak to zrobiliśmy dla maszyn Turinga , ale teraz nie potrzebujemy części ⟨b, L⟩ lub ⟨b, R⟩etykiet; przejście jest oznaczone tylko symbolem, który je wyzwala.

Potęga automatów skończonych

Do czego zdolne są automaty skończone? Rysunek

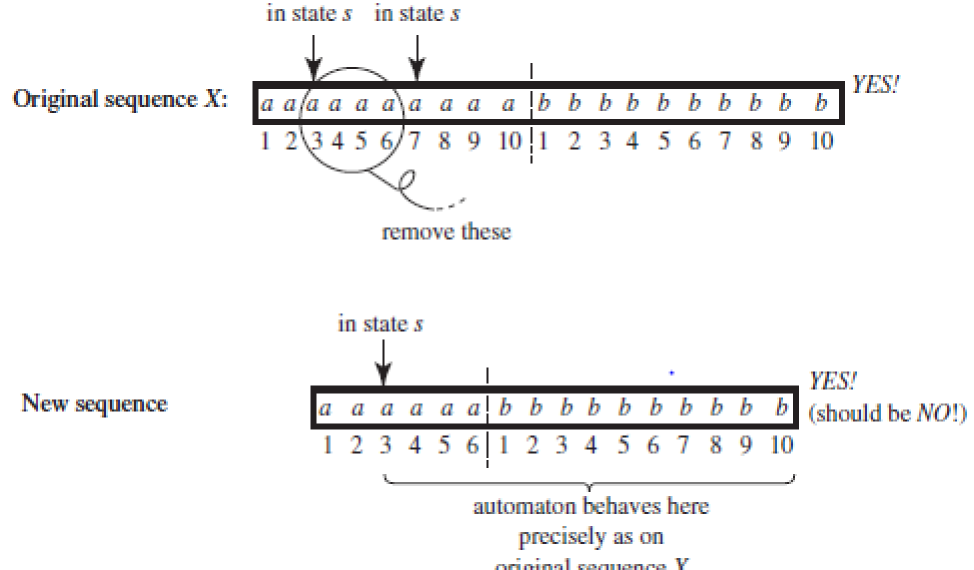


pokazuje automat skończony, który decyduje, czy ciąg wejściowy a i b zawiera parzystą liczbę a. (Co robi automat z rysunku 2?)

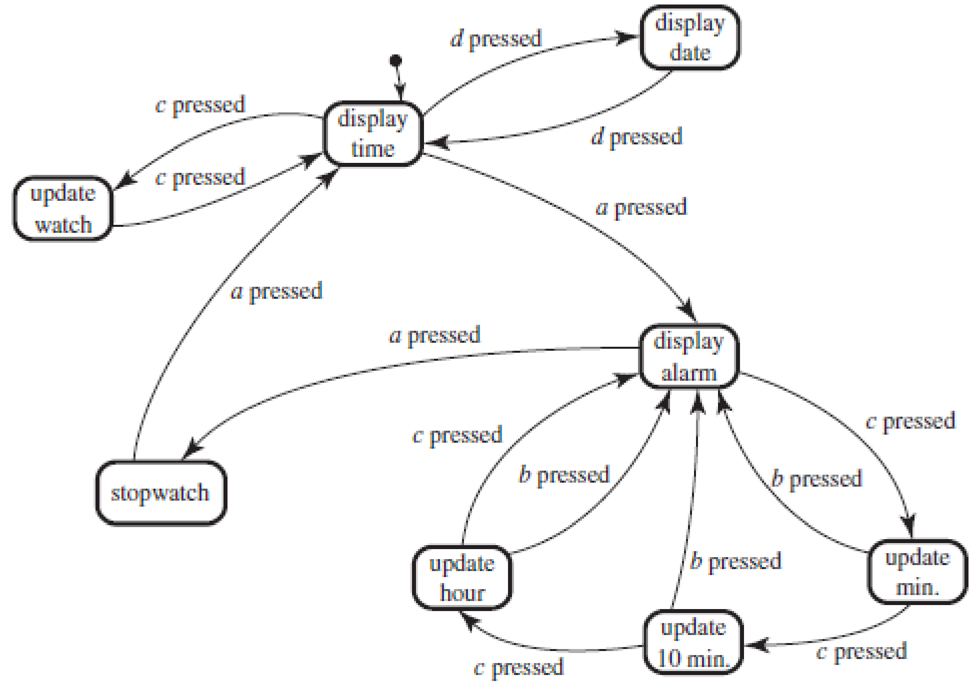


Odpowiadanie na pytania dotyczące parzystości (czyli parzystości i nieparzystości) może na początku wymagać umiejętności liczenia. To nie. Automat z rysunku 1 wykonuje to zadanie nie przez zliczanie liczby a, które widział, ale przez naprzemienne przechodzenie między dwoma stanami, które wskazują aktualny stan parzystości tej liczby. W rzeczywistości dość łatwo pokazać, że automaty skończone nie mogą liczyć. Jako ilustrację przekonajmy się, że żaden automat skończony nie może rozstrzygnąć, czy jego ciąg wejściowy zawiera dokładnie taką samą liczbę a i b.

Dowód będzie w drodze sprzeczności. Załóżmy, że jakiś automat F rzeczywiście może rozwiązać ten problem decyzyjny, co oznacza, że dla dowolnego ciągu a i b F mówi „tak" dokładnie, jeśli liczba wystąpień obu jest równa. Oznacz przez N liczbę stanów w F. Rozważ ciąg X, który składa się dokładnie z N + 1 a, po którym następuje N + 1 b. Oczywiście nasz automat musi powiedzieć „tak", gdy przekazuje X jako dane wejściowe, ponieważ X ma taką samą liczbę a i b. Argument ten wykorzystuje teraz tak zwaną zasadę przegródek: jeśli wszystkie gołębie muszą wejść do przegródek, ale jest więcej gołębi niż norek, to przynajmniej jedna dziurka będzie musiała pomieścić więcej niż jednego gołębia. W naszym przypadku gołębie to N + 1 a stanowiące pierwszą połowę ciągu wejściowego X, a dziury to stany N automatu F. Ponieważ F porusza się wzdłuż X, muszą być co najmniej dwa różne kwadraty taśmy lub pozycje , w początkowej sekwencji a, gdzie F będzie w tym samym stanie. Dla pewności załóżmy, że N wynosi 9, że dwie pozycje to trzecia i siódma, oraz że wspólny stan wprowadzony w tych dwóch przypadkach to s.



Teraz F nie może się cofać, nie może zapisywać żadnych informacji i działa wyłącznie w oparciu o jeden symbol, który widzi i jego aktualny stan. Jest zatem oczywiste, że gdy osiągnie siódmy kwadrat, jego zachowanie na pozostałych danych wejściowych nie będzie zależeć od niczego, co widział wcześniej, z wyjątkiem faktu, że jest teraz w stanie s. W konsekwencji, gdybyśmy usunęli kwadraty od 3 do 6, w wyniku czego powstałaby sekwencja 6 a, a nie 10, po której następują 10 b, automat F nadal osiągnąłby trzecie a w stanie s i postępowałby tak, jakby osiągnął siódme a w oryginalnej kolejności. W szczególności, skoro powiedział „tak" w oryginalnej sekwencji, powie „tak" w nowej. Ale nowa sekwencja ma 6 a i 10 b, a zatem nasze założenie, że F mówi „tak" tylko w przypadku sekwencji o tej samej liczbie a i b, jest sprzeczne. Wniosek jest taki, że żaden automat skończony nie jest w stanie rozwiązać tego problemu decyzyjnego. Możesz cieszyć się konstruowaniem podobnego dowodu, również używając zasady szufladki, aby pokazać, że automaty skończone nie mogą wykryć palindromów. Automaty skończone są rzeczywiście bardzo ograniczone, ale oddają sposób działania wielu codziennych systemów. Na przykład rysunek 4 pokazuje automat skończony opisujący część zachowania prostego zegarka cyfrowego, z czterema przyciskami sterującymi, a, b, c i d.



Nazwy stanów nie wymagają wyjaśnień i z każdym stanem możemy powiązać szczegółowe opisy działań, które są w nim wykonywane. Zauważ, że symbole wejściowe są tak naprawdę sygnałami lub zdarzeniami, które wchodzą do systemu z jego otoczenia - w tym przypadku użytkownika naciskającego przyciski. Słowo wejściowe jest zatem tylko sekwencją przychodzących sygnałów.

Badania nad abstrakcyjnymi modelami obliczeń

W pewnym sensie materiał tego rozdziału stanowi podstawę niemal wszystkich tematów poruszonych do tej pory w książce. Prymitywne modele obliczeń, niezależnie od tego, czy są uniwersalne, jak maszyny Turinga i programy licznikowe, czy mniej wydajne, jak automaty skończone lub modele ograniczone zasobami, są niezwykle ważnymi obiektami dla badań w dziedzinie informatyki. Skupiają uwagę na podstawowych właściwościach rzeczywistych urządzeń algorytmicznych, takich jak komputery i języki programowania, i dają początek podstawowym i solidnym pojęciom, które są niewrażliwe na konkretne używane urządzenie. W częściach 7 i 8 poruszyliśmy obszary badawcze złożoności obliczeniowej i nierozstrzygalności, z których oba w zasadniczy sposób wykorzystują maszyny Turinga i programy licznikowe. Automaty skończone są podstawą bardzo obszernych badań, zarówno natury teoretycznej, jak i praktycznej, które mają fundamentalne implikacje dla tak różnorodnych dziedzin, jak projektowanie elektrowni i rozumienie mózgu. Inne ograniczenia maszyn Turinga były i są nadal intensywnie badane w obszarach badawczych dotyczących automatów i teorii języka formalnego. Wśród nich godne uwagi są automaty pushdown, które mogą być postrzegane jako automaty skończone wyposażone w stos, na który symbole mogą być wpychane, odczytywane z góry i wyskakiwane. Automaty pushdown mają znaczenie w różnych przedsięwzięciach, w tym w projektowaniu kompilatorów i językoznawstwie strukturalnym, ponieważ mogą być używane do reprezentowania rekurencyjnie zdefiniowanych struktur składniowych, takich jak programy komputerowe lub zdania prawne w języku angielskim. Okazuje się na przykład, że pod względem klas problemów, które mogą rozwiązać, niedeterministyczne automaty ze stopniowaniem są silniejsze niż automaty deterministyczne, co nie jest prawdą ani dla słabszych automatów skończonych, ani dla silniejszych maszyn Turinga. Na przykład można skonstruować niedeterministyczny automat spychający do wykrywania palindromów (jak?), ale można udowodnić, że żaden deterministyczny

automat spychający nie jest w stanie tego zrobić. Badaczy interesują problemy decyzyjne dotyczące samych modeli. Na przykład wiemy, że równoważność algorytmiczna jest nierozstrzygalna, a więc w szczególności nierozstrzygnięte jest, czy dwie dane maszyny Turinga rozwiązują ten sam problem algorytmiczny. Jeśli jednak zejdziemy aż do automatów skończonych, równoważność staje się rozstrzygalna. Jeśli chodzi o automaty pushdown, historia jest następująca. W przypadku niedeterministycznym równoważność jest nierozstrzygalna, co jest starym i dobrze znanym wynikiem. W przeciwieństwie do tego, dla deterministycznych automatów ze zsuwaniem pytanie to zostało uznane za jeden z najtrudniejszych nierozwiązanych problemów w teorii automatów i języków formalnych. Kilka lat temu zostało ostatecznie rozwiązane, twierdząco. Stąd wiemy teraz, że równoważność jest rozstrzygalna również dla deterministycznych automatów ze stopniowaniem, co skutkuje interesującą różnicą rozstrzygalności między deterministyczną i niedeterministyczną wersją tego samego modelu obliczeń. Nawiasem mówiąc, ilekroć takie problemy decyzyjne okazują się rozstrzygalne, badania sprowadzają się do zadania określenia ich złożoności obliczeniowej i tutaj również pojawiło się wiele interesujących i przydatnych wyników. Naukowcy są również zainteresowani znalezieniem właściwej „tezy CT” dla pewnych wariantów standardowych pojęć obliczalności i wykonalności. Jeden przypadek, który został rozwiązany, dotyczy baz danych, w przypadku których interesujące są następujące pytania. Jakie jest podstawowe uniwersalne pojęcie „obliczalnego” zapytania w bazie danych lub bazie wiedzy? Czym są „maszyny Turinga” baz danych? Próba odpowiedzi na pytanie naiwnie przez kodowanie baz danych na taśmach maszyn Turinga i odwoływanie się do standardowej teorii maszyn Turinga nie ma żadnej wartości. Chociaż można to zrobić, oczywiście wynikająca z tego linearyzacja baz danych może, ze swej natury, wymusić porządek na zestawach danych, które nie mają być uporządkowane w żaden szczególny sposób (powiedzmy listę równie ważnych pracowników). . Zapytania nie powinny mieć możliwości korzystania z kolejności na takich elementach danych, a każdy proponowany uniwersalny model lub język zapytań do bazy danych powinien odzwierciedlać ten fakt. Obszary badawcze teorii obliczalności, teorii automatów i języka formalnego, teorii funkcji rekurencyjnych i teorii złożoności są zatem ściśle ze sobą powiązane. Od czasu pionierskich prac z lat 30. abstrakcyjne modele obliczeń odgrywały kluczową rolę w uzyskiwaniu fundamentalnych wyników w tych dziedzinach, stanowiąc w ten sposób podstawę wielu najważniejszych osiągnięć w algorytmice.

Łagodzenie zasad

Modele równoległości, współbieżności i alternatywne

Fakt, że informatyka nie przynosi tylko dobrych wiadomości, popchnął badaczy w wielu kierunkach, mających na celu próbę złagodzenia problemu. Omówimy niektóre z najbardziej interesujących z nich: równoległość i współbieżność, obliczenia kwantowe i obliczenia molekularne. Każdy z nich reprezentuje nowy paradygmat algorytmiczny i wszystkie robią to, rozluźniając podstawowe założenie leżące u podstaw konwencjonalnego przetwarzania, a mianowicie, że algorytm jest wykonywany przez mały procesor Runaround, który pracuje sam. Równoległość i współbieżność dotyczą bezpośredniego ustawiania tak, aby kilka procesorów (lub kilka małych Runarounds) pracowało razem, we współpracy. Obliczenia kwantowe przenoszą obliczenia do tajemniczej dziedziny mechaniki kwantowej, w której równoległość wynika ze zdolności cząstek do przebywania w więcej niż jednym miejscu jednocześnie. A obliczenia molekularne, czyli biologiczne, są próbą nakłonienia molekuł do wykonywania pracy za nas poprzez masowy, pozornie nadmiarowy paralelizm. Aby wyczuć równoległość, rozważ następujące kwestie. Kilka lat temu w rejonie Los Angeles odbył się konkurs o tytuł mistrza świata w budownictwie szybkim. Należało przestrzegać pewnych sztywnych zasad, takich jak liczba pokoi, wymagane media i dozwolone materiały budowlane. Prefabrykacja nie była dozwolona, ale fundamenty można było przygotować wcześniej. Dom uznano za skończony, kiedy ludzie mogli w nim dosłownie zacząć żyć; cała instalacja wodno-kanalizacyjna i elektryczność musiały być na miejscu i działać idealnie, drzewa i trawa musiały ozdabiać podwórko i tak dalej. Nie nałożono żadnych ograniczeń na wielkość zespołu budowlanego. Zwycięska firma wykorzystała zespół około 200 budowniczych i przygotowała dom w nieco ponad cztery godziny! To uderzająca ilustracja korzyści płynących z równoległości: jedna osoba pracująca sama potrzebowałaby znacznie więcej czasu na ukończenie domu. Tylko dzięki wspólnej pracy, wśród niesamowitych wyczynów współpracy, koordynacji i wzajemnego wysiłku, zadanie mogło zostać zrealizowane w tak krótkim czasie. Obliczenia równoległe umożliwiają równoległą pracę wielu komputerów lub wielu procesorów w jednym komputerze. Obliczenia kwantowe to zupełnie nowe podejście do obliczeń, oparte na mechanice kwantowej, tym kuszącym i paradoksalnym dziele fizyki XX wieku. Jak dotąd odkryto kilka zaskakująco wydajnych algorytmów kwantowych do rozwiązywania problemów, o których nie wiadomo, że są wykonalne w „klasycznym" sensie. Jednak do pracy wymagają zbudowania specjalnego komputera kwantowego, czego na razie nie ma. Obliczenia molekularne, kolejny bardzo niedawny paradygmat, umożliwiły naukowcom nakłonienie rozpuszczalnika molekularnego do rozwiązania przypadków pewnych problemów NP-zupełnych, co stwarza interesujące i ekscytujące możliwości. W pozostałej części omówiono te idee, przy czym paralelizm i współbieżność rozproszona - bardziej klasyczna z nich - zostały potraktowane bardziej szczegółowo.

Równoległość lub łączenie sił

Naszym głównym celem jest zbadanie konsekwencji tego, że wiele procesorów wspólnie osiąga cele algorytmiczne, współpracując ze sobą. To rozluźnienie jest częściowo motywowane chęcią wykorzystania paralelizmu w sprzęcie, a mianowicie dostępności tak zwanych komputerów równoległych, które składają się z wielu oddzielnych elementów przetwarzających, współpracujących i pracujących równolegle. Później uogólnimy również sztywne pojęcie wejścia/wyjścia problemu algorytmicznego na przypadki związane z wiecznością lub ciągłym zachowaniem, które wcale nie musi prowadzić do zakończenia. Takie problemy wynikają z rozproszonych środowisk o z natury równoległej naturze, takich jak systemy rezerwacji lotów czy sieci telefoniczne, a także są związane z rozwojem systemu.

Równoległość pomaga

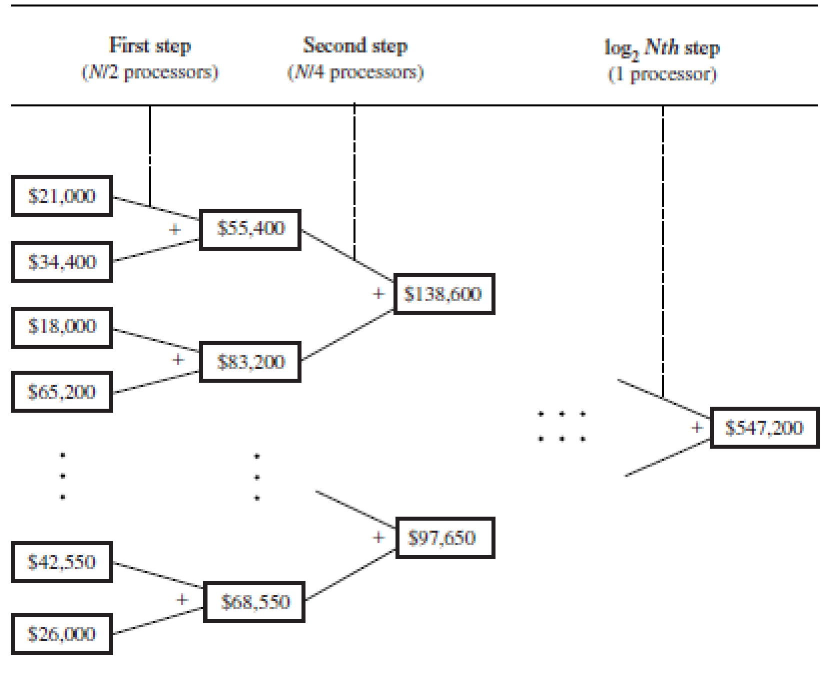
Historia budowy domu pokazuje, że robienie rzeczy równolegle może zdziałać cuda. Zobaczmy, do czego sprowadzają się te cuda pod względem wydajności algorytmicznej. Jeśli algorytm wymaga sekwencji instrukcji:

X ← 3; Y ← 4,

wtedy oczywiście moglibyśmy zaoszczędzić czas, wykonując je jednocześnie, równolegle. Musimy jednak być bardzo ostrożni, aby nie „paralelizować" byle czego; jeśli instrukcje były:

X ← 3; Y ← X

historia byłaby zupełnie inna, ponieważ po równoległym wykonaniu nowa wartość Y może być starą wartością X, a nie nową , 3. Tutaj efekt drugiej instrukcji zależy od wyników pierwszej, a więc dwóch nie może być zrównoleglony. Oczywiście można je modyfikować w sposób, który pozwoli obejść problem, ale w swojej pierwotnej formie muszą być wykonane w odpowiedniej kolejności. Aby lepiej zilustrować tę kwestię, rozważ problem kopania rowu o głębokości jednej stopy, szerokości jednej stopy i długości 10 stóp. Jeśli jedna osoba jest w stanie wykopać rów jeden po drugim w, powiedzmy, godzinę, 10 osób mogłoby w oczywisty sposób wykopać pożądany rów w godzinę. Paralelizm jest tutaj najlepszy. Jeśli jednak pożądany rów ma mieć jedną stopę szerokości, jedną stopę długości i 10 stóp głębokości, równoległość nic nie da, a nawet 100 osób potrzebowałoby 10 godzin, aby wykonać zadanie. Niektóre problemy algorytmiczne można łatwo zrównoleglić, mimo że pierwsze rozwiązania, które przychodzą na myśl, są mocno sekwencyjne; tak naprawdę nie są one z natury sekwencyjne. Wiele z nich można rozwiązać znacznie efektywniej dzięki przetwarzaniu równoległemu. Rozważmy problem sumowania wynagrodzeń z rozdziału 1. Być może konieczne byłoby przejrzenie listy pracowników w kolejności, w celu uzyskania opisanego tam algorytmu liniowego. Bynajmniej. Można wymyślić równoległy algorytm, który będzie działał w czasie logarytmicznym — rzeczywiście jest to doniosłe ulepszenie, jak pokazano w Części 6. Metoda polega na rozważeniu najpierw całej listy N pracowników w parach, <pierwszy, drugi> <trzeci, czwarty> i tak dalej, a także sumując jednocześnie dwie pensje we wszystkich parach, uzyskując w ten sposób listę N/2 nowych liczb (przypomnijmy, że N to całkowita liczba pracowników). Zajmuje to czas tylko jednego dodania, który tutaj liczymy jako pojedynczą jednostkę czasu. Nowa lista jest wtedy również rozpatrywana w parach, a dwie liczby w każdej z nich są ponownie dodawane jednocześnie, co daje nową listę N/4 liczb. Trwa to tak długo, aż pozostanie tylko jeden numer; jest to suma wynagrodzeń z całej listy. Rysunek



ilustruje ten prosty pomysł. Jak wyjaśniono w rozdziale 6, liczba razy N można podzielić przez 2, aż osiągnie 1, wynosi około $\log_2$ N, a zatem logarytmiczne ograniczenie czasowe. Mając na uwadze liczby z Części 6, wynika z tego, że 1000 pensji można zsumować w czasie potrzebnym na wykonanie zaledwie 10 uzupełnień, a milion pensji można zsumować w czasie zaledwie 20 uzupełnień.

Stały a rozszerzający się równoległość

Zauważ, że aby jednocześnie wykonać 500 dodatków wymaganych w pierwszym kroku sumowania 1000 wynagrodzeń, potrzebujemy 500 procesorów. Te same mogą być następnie użyte do przeprowadzenia równolegle 250 dodawania drugiego etapu (połowa z nich byłaby oczywiście bezczynna), następnie 125 dodanych trzeciego etapu i tak dalej. Oczywiście sama duża liczba procesorów nie wystarczy; musimy również ułożyć dane w taki sposób, aby odpowiednie liczby były szybko dostępne dla właściwych procesorów, gdy ich potrzebują. Dlatego dobre struktury danych i metody komunikacji są kluczowe dla szybkich algorytmów równoległych. Koncentrując się jednak na liczbie procesorów w tej chwili, widzimy, że aby osiągnąć skrócenie z czasu liniowego do logarytmicznego, potrzebujemy N/2 procesorów, liczby zależnej od N, czyli długości wejścia. Rzeczywiście, tak jest z konieczności, ponieważ gdybyśmy mieli tylko stałą liczbę procesorów, nie moglibyśmy poprawić rzeczy poza stałą ukrytą pod dużym-O: moglibyśmy być w stanie robić rzeczy dwa razy szybciej lub 100 razy szybciej, ale nadal byłaby liniowa, to znaczy O(N), i nie byłoby poprawy o rząd wielkości. Można by twierdzić, że rosnąca liczba procesorów jest po prostu niewykonalna. Ale nie ma też rosnącej ilości czasu ani pamięci. Celem miar złożoności jest zapewnienie środków do oszacowania nieodłącznej trudności w rozwiązywaniu problemów algorytmicznych w miarę zwiększania się danych wejściowych, przy czym oszacowanie jest podawane jako funkcja wielkości danych wejściowych. Jeśli mamy podsumowywać listy z nie więcej niż milionem pensji i jeśli mamy pod ręką pół miliona procesorów, to będziemy potrzebować bardzo mało czasu (mniej więcej z 20 dodatkami). Jeśli mamy mniej procesorów, możemy zrównoleglać do pewnego punktu; to znaczy do pewnej głębokości w drzewie rysunku powyżej. Od tego momentu trzeba będzie stosować mieszankę równoległości i sekwencyjności. Oczywiście wynik nie będzie tak dobry. Osiągnięcie poprawy o rząd wielkości wymaga zatem rozszerzania równoległości - to znaczy liczby procesorów, która rośnie wraz

ze wzrostem N. Jednak ta liczba nie musi koniecznie wynosić N/2. Na przykład można dodać pensje na liście o długości N w czasie O(√N), jeśli dostępnych jest √ N procesorów - na przykład milion pensji w czasie około 1000 uzupełnień z 1000 procesorów. (Widzisz jak?) Oczywiście, czas 1000 dodatków nie jest tak dobry jak 20, ale wtedy 1000 procesorów to mniej niż pół miliona, więc dostajemy to, za co płacimy, że tak powiem.

Sortowanie równoległe

Sortowanie listy o długości N (na przykład porządkowanie pomieszanej książki telefonicznej) to doskonały problem do omówienia korzyści z przetwarzania równoległego. Rozważ algorytm scalania z Części 4. Wymagał podzielenia listy danych wejściowych na połowy, posortowania ich rekurencyjnie, a następnie scalenia posortowanych połówek. Scalenie uzyskuje się poprzez wielokrotne porównywanie aktualnie najmniejszych (pierwszych) elementów w połówkach, co skutkuje wysłaniem jednego z nich na wyjście. Algorytm można opisać schematycznie w następujący sposób:

podprogram sort-L:

(1) jeśli L składa się z jednego elementu, to jest sortowane;

(2) w przeciwnym razie wykonaj następujące czynności:

(2.1) podzielić L na dwie połowy, L₁ i L₂;

(2.2) wywołanie sort-L₁;

(2.3) wywołanie sort-L₂;

(2.4) scalić listy wynikowe w jedną posortowaną listę.

W Części 6 stwierdzono, że złożoność czasowa tego algorytmu wynosi O(N × log N). Teraz oczywiście dwie czynności sortowania połówek (wiersze (2.2) i (2.3) w algorytmie) nie kolidują ze sobą, ponieważ dotyczą rozłącznych zbiorów elementów. Dlatego można je przeprowadzać równolegle. Nawet zrównoleglenie sortowania połówek tylko raz, na najwyższym poziomie algorytmu, skutkujące potrzebą tylko dwóch procesorów, poprawiłoby sytuację, ale tylko w obrębie stałej big-O, jak już wyjaśniono. Możemy jednak przeprowadzić oba rodzaje równolegle na wszystkich poziomach rekurencji, uzyskując następujące wyniki:

podprogram równoległy-sort-L:

(1) jeśli L składa się z jednego elementu, to jest sortowane;

(2) w przeciwnym razie wykonaj następujące czynności:

(2.1) podzielić L na dwie połowy, L₁ i L₂;

(2.2) jednocześnie wywołaj Parallel-sort-L₁ i Parallel-sort-L₂;

(2.3) scalić listy wynikowe w jedną posortowaną listę.

Przy skrupulatnym przestrzeganiu algorytm ten działa bardzo podobnie do algorytmu sumowania równoległego opartego na drzewie z rysunku powyżej. Po pierwsze, po zejściu do najniższego poziomu rekurencji, jednocześnie porównuje dwa elementy w każdej z par N/2, układając każdą parę we właściwej kolejności. Następnie, ponownie jednocześnie, łączy każdą parę par w posortowaną czwórkę, a następnie każdą parę z czterech krotek w ósemkę i tak dalej. Pierwszy krok zajmuje tylko jedno porównanie, drugi zajmuje trzy (dlaczego?), trzeci siedem, czwarty piętnaście i tak dalej. Zakładając dla uproszczenia, że N jest potęgą liczby 2, łączna liczba porównań wynosi:

1 + 3 + 7 + 15 + … (N − 1)

czyli mniej niż 2BA. Stąd całkowity czas jest liniowy. Ceną zapłaconą za ulepszenie O(N × log N) do O(N) jest zapotrzebowanie na N/2 procesorów. Tutaj też jest kompromis; możemy posortować N elementów w czasie O(N × log N) z jednym procesorem lub w czasie liniowym z liniową liczbą procesorów. Właściwie, jak zobaczymy później, możemy zrobić jeszcze lepiej.

Złożoność produktu: czas × rozmiar

Liczba procesorów wymaganych przez algorytm równoległy jako funkcja długości jego danych wejściowych jest jednym ze sposobów pomiaru złożoności sprzętu wymaganego do uruchomienia algorytmu. Nieco nadużywając terminologii, możemy nazwać tę miarę po prostu złożonością rozmiaru algorytmu, rozumiejąc, że nie mamy na myśli ani długości algorytmu, ani ilości pamięci, jaką wymaga, ale raczej wielkość wymaganej armii procesorów. Ponieważ zarówno miara czasu, jak i rozmiar odgrywają rolę w analizie algorytmów równoległych, nie jest jasne, w jaki sposób powinniśmy określić względną wyższość takich algorytmów. Czy nieco szybszy algorytm jest lepszy, nawet jeśli używa o wiele więcej procesorów? A może powinniśmy poświęcić małą wielkość, aby uzyskać znaczną oszczędność czasu? Jednym ze sposobów szacowania jakości algorytmu równoległego jest połączenie obu miar – pomnożenie czasu przez rozmiar

	Name	Size (no. of processors)	Time (worst case)	Product (time × size)
SEQUENTIAL	Bubblesort	1	$O(N^2)$	$O(N^2)$
	Mergesort	1	$O(N \times \log N)$	$O(N \times \log N)$ (optimal)
PARALLEL	Parallelized mergesort	$O(N)$	$O(N)$	$O(N^2)$
	Odd-even sorting network	$O(N \times (\log N)^2)$	$O((\log N)^2)$	$O(N \times (\log N)^4)$
	"Optimal" sorting network	$O(N)$	$O(\log N)$	$O(N \times \log N)$ (optimal)

Algorytm o lepszej złożoności produktu jest uważany za lepszy. Warto zauważyć, że najlepsza miara produktu nie może być mniejsza niż dolna granica złożoności sekwencyjno-czasowej problemu, ponieważ możliwe jest sekwencjonowanie dowolnego algorytmu równoległego. Odbywa się to poprzez symulację działań różnych procesorów na pojedynczym procesorze w kolejności zgodnej z kolejnością zaleconą przez algorytm równoległy. (Na przykład, gdybyśmy sekwencjonowali algorytm równoległego sortowania przez scalanie, moglibyśmy posortować dwie połówki w dowolnej kolejności, ale oba sortowania musiałyby zostać zakończone przed przeprowadzeniem scalania.) Całkowity czas trwania symulacji wynosi z grubsza suma całego czasu zajętego przez wszystkie procesory; to znaczy

czas produktu × rozmiar oryginalnego algorytmu równoległego. Tak więc, gdybyśmy hipotetycznie mogli znaleźć algorytm sortowania równoległego, który zajmowałby czas logarytmiczny i używał tylko procesorów O(N/log N), można by wyprowadzić zsekwencjonowaną wersję, która działałaby w czasie zgodnym z kolejnością produktu, który jest liniowa (dlaczego?). Ale byłoby to sprzeczne z dolnym ograniczeniem O(N × log N) przy sortowaniu sekwencyjnym. Najlepsze, na co możemy więc liczyć, to algorytm sortowania równoległego, który wykazuje optymalną wydajność produktu O(N × log N). W tym sensie algorytm sekwencyjnego sortowania przez scalanie jest na przykład optymalny. Ale z drugiej strony nie jest to algorytm równoległy. Szczególnie intrygujące jest pytanie, czy istnieje równoległy, oparty na porównaniach algorytm sortowania, który działa w czasie logarytmicznym, ale wymaga tylko liniowej liczby procesorów. W pewnym sensie reprezentowałoby to idealną procedurę sortowania równoległego - niezwykle szybką, ale rozsądnej wielkości. Jak zobaczymy, problem ten został rozwiązany twierdząco.

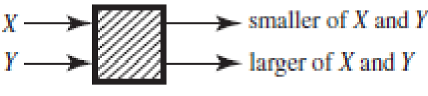
Sieci: równoległość połączenia stałego

Jak dotąd nie powiedzieliśmy nic o współpracy różnych procesorów. Oczywiście nie działają one w całkowitym odosobnieniu, ponieważ muszą przekazywać sobie wyniki pośrednie. Nawet prosty algorytm sumowania równoległego wymaga pewnego rodzaju współpracy między procesorami wytwarzającymi sumy pośrednie i tymi, które używają ich w następnej kolejności. Wydajność algorytmu równoległego może się znacznie różnić w zależności od różnych metod współpracy. Jedno z podejść opowiada się za pamięcią współdzieloną, co oznacza z grubsza, że pewne zmienne lub struktury danych są współdzielone przez wiele procesorów. W ramach tego podejścia ważne jest określenie, czy udostępnianie polega tylko na odczytaniu wartości z odpowiedniej pamięci, czy też ich zapisie. Jeśli zostanie wybrane drugie podejście, musimy zdecydować, jak rozwiązać konflikty zapisu (na przykład dwa procesory próbują jednocześnie pisać do tej samej zmiennej lub lokalizacji pamięci), a konkretna wybrana metoda może mieć duże znaczenie w wynikowym algorytmie moc. Nieograniczona pamięć współdzielona jest uważana za nierealistyczną, głównie dlatego, że jeśli chodzi o budowanie prawdziwych komputerów równoległych, wzorzec połączeń staje się niemożliwie skomplikowany. Albo każdy procesor musi być podłączony do zasadniczo każdej lokalizacji pamięci, albo (jeśli pamięć jest fizycznie rozdzielona między procesorami) każdy procesor musi być podłączony do każdego innego. W obu przypadkach jest to zazwyczaj sytuacja beznadziejna: jeśli wymagana liczba procesorów rośnie wraz z N, jak to ma miejsce w przypadku wszystkich nietrywialnych algorytmów równoległych, połączenia szybko stają się nierozsądnie skomplikowane. Bardziej realistycznym podejściem jest wykorzystanie sieci o stałym połączeniu lub w skrócie tylko sieci i zaprojektowanie specjalnie dla nich algorytmów równoległych. Słowo „stały” oznacza, że każdy procesor jest podłączony do co najwyżej pewnej stałej liczby sąsiednich procesorów. W wielu przypadkach oznacza to również, że cała sieć jest skonstruowana jako maszyna specjalnego przeznaczenia, bardzo wydajnie rozwiązująca jeden konkretny problem algorytmiczny. Procesory w sieci specjalnego przeznaczenia mają zazwyczaj bardzo ograniczone możliwości obliczeniowe. Jedną dobrze znaną klasą sieci są sieci logiczne lub obwody logiczne, nazwane na cześć dziewiętnastowiecznego logika George'a Boole'a, który wynalazł zasady manipulowania wartościami logicznymi prawda i fałsz (a zatem również zasady manipulowania odpowiadającymi im wartościami bitowymi). , 1 i 0). W sieci logicznej procesory nazywane są bramkami i obliczają proste funkcje logiczne jedno- lub dwubitowych wartości. Bramka AND wytwarza 1 dokładnie wtedy, gdy oba jej wejścia mają wartość 1, bramka OR wytwarza 1 dokładnie, gdy co najmniej jedno z jej dwóch wejść ma wartość 1, a bramka NOT odwraca wartość pojedynczego wejścia. W pewnym sensie każdy efektywnie rozwiązywalny problem algorytmiczny P może być rozwiązany przez efektywnie obliczalny jednolity zbiór sieci logicznych. Aby móc dokładniej określić ten fakt, wyobraź sobie, że dane wejściowe P są zakodowane przy użyciu tylko zer i jedynek. Twierdzi się, że dla każdego takiego problemu P istnieje algorytm (powiedzmy maszyna Turinga), który

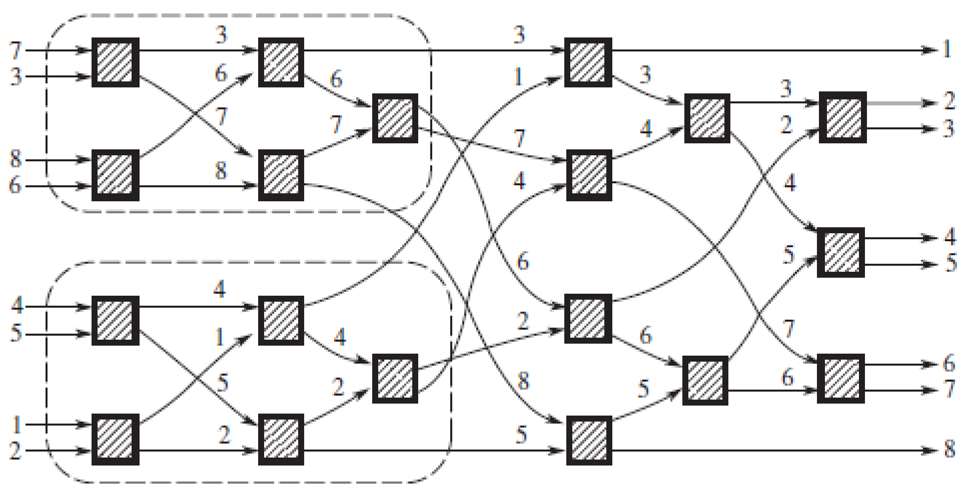
przyjmuje liczbę wejściową N i wyprowadza opis sieci logicznej, która rozwiązuje problem P dla danych wejściowych składających się z N bitów. Jednak ten nieco dziwny rodzaj uniwersalności nie jest powodem, dla którego wprowadzamy sieci. Bardziej interesują nas sieci, które są specjalnie zaprojektowane w celu zapewnienia skutecznych rozwiązań konkretnych problemów.

Nieparzysta i parzysta sieć sortowania

Sieci specjalnego przeznaczenia zostały znalezione dla wielu problemów, ale przede wszystkim do sortowania i łączenia. Sieć sortującą można postrzegać jako architekturę komputerową specjalnego przeznaczenia, zapewniającą stały wzorzec połączeń niezwykle prostych procesorów, które współpracują w celu równoległego sortowania N elementów. Większość sieci sortujących wykorzystuje tylko jeden bardzo prosty rodzaj procesora, zwany komparatorem, który wprowadza dwa elementy i wyprowadza je w kolejności, jak pokazano na rysunku



Zilustrujmy to podejście tak zwaną siecią sortowania nieparzysto-parzyste. Sieć ta jest konstruowana rekursywnie, stosując zasadę konstruowania sieci dla N elementów z sieci dla N/2 elementów. Nie będziemy tu podawać opisu tej ogólnej zasady, ale zilustrujemy ją przykładem. Rysunek

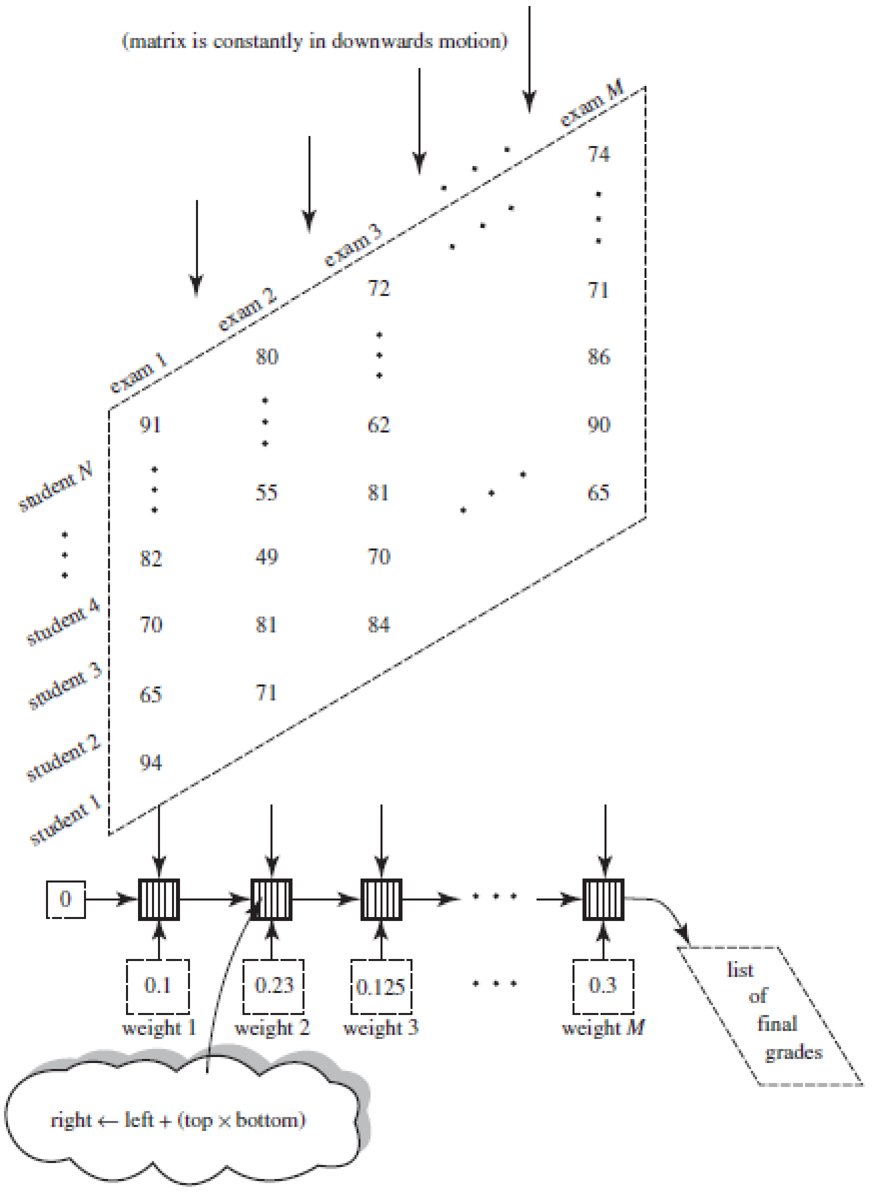


przedstawia sieć dla przypadku, gdy N wynosi 8, z przykładową listą wejść zapisaną wzdłuż linii. Części sieci ujęte w linie przerywane reprezentują dwie sieci nieparzyste-parzyste po 4 elementy każda. W podobnym duchu sieć dla 16 elementów miałaby po lewej stronie dwie sieci dla 8 elementów, identyczne jak na rysunku. Pozostałe części sieci nieparzysto-parzystej składają się z różnych podsieci do łączenia posortowanych list. To, co jest w tym wszystkim sprytne, to sposób, w jaki podsieci są ze sobą połączone, co odkryjesz, jeśli spróbujesz zdefiniować ogólną regułę konstrukcji rekurencyjnej. Czas zajmowany przez nieparzystą sieć sortującą można wykazać jako $O((\log N)_2)$, a jej rozmiar (liczba procesorów) to $O(N \times (\log N)_2)$, stąd nie jest optymalny, ponieważ ich iloczyn jest większy niż optymalny. Przełom w rodzaju został osiągnięty w 1983 r. wraz z pojawieniem się pomysłowej sieci sortowania, która wykorzystuje procesory $O(N \times \log N)$ i zajmuje tylko czas logarytmiczny. Później to rozwiązanie połączono z odmianą sieci nieparzysto-parzystej, aby ostatecznie uzyskać optymalną sieć logarytmiczną o rozmiarze liniowym. Jest to zatem optymalny algorytm sortowania równoległego

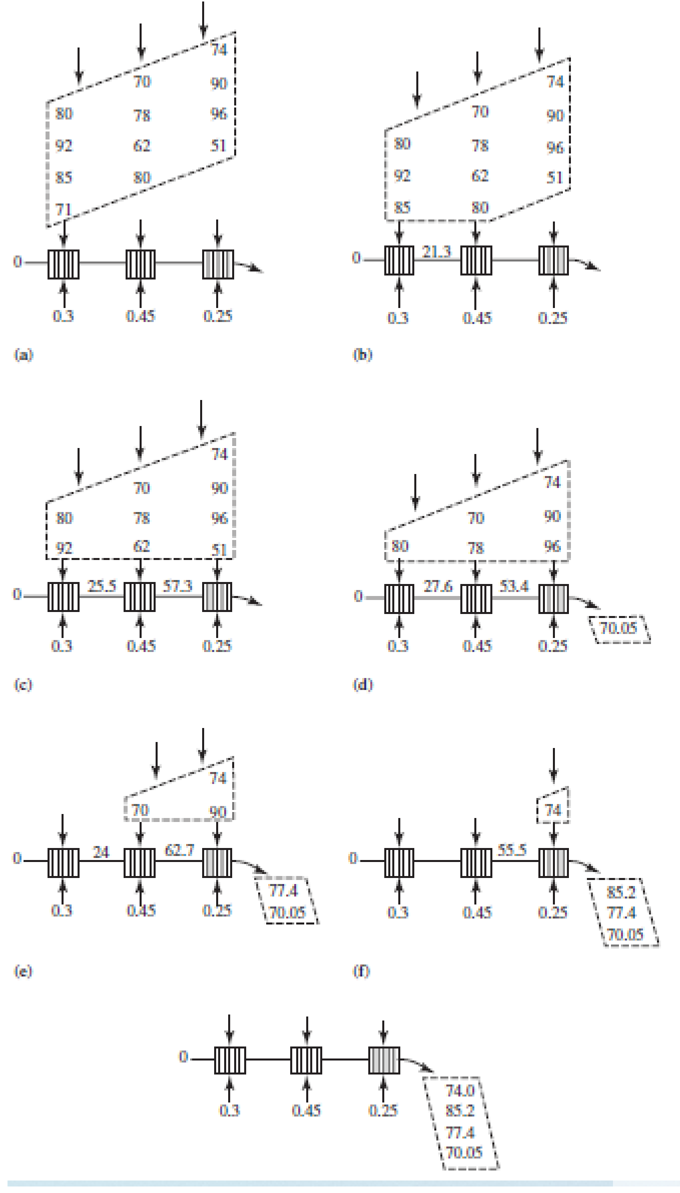
wspomniany wcześniej. Niestety, poza tym, że są niezwykle skomplikowane, stałe big-O w obu tych sieciach logarytmicznych są ogromne, co czyni je w praktyce całkiem bezużytecznymi dla rozsądnej wielkości N. Dla kontrastu, stałe dla wielu teoretycznie gorszych sieci są bardzo małe ; na przykład te z sieci nieparzystych i parzystych są mniejsze niż 1 (około 1/2 dla miary czasu i 1/4 dla rozmiaru). W szczególności możemy zbudować nieparzystą sieć sortującą z nieco ponad 1000 komparatorów, która posortuje 100 elementów w czasie potrzebnym do przeprowadzenia zaledwie 25 porównań. Dla 1000 elementów zajęłoby to tylko około 55 porównań, ale potrzebowalibyśmy około 23 000 komparatorów. Jeśli czas jest kluczowym czynnikiem, a wszystkie listy do posortowania będą mniej więcej tej samej długości, sieć sortująca, taka jak ta, staje się całkiem praktyczna.

Więcej o sieciach: Obliczanie średnich ważonych

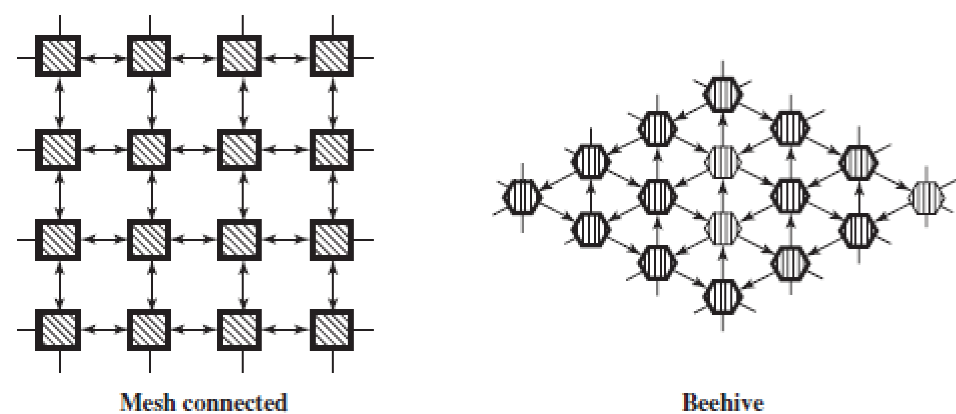
W sieci nieparzystej każdy komparator jest używany tylko raz przy sortowaniu danej listy. Lista wejść zbiera się razem, a cały cykl życia każdego komparatora polega na oczekiwaniu na dwa własne wejścia, porównywaniu ich, wysyłaniu maksimum i minimum wzdłuż odpowiednich linii wyjściowych i wyłączaniu się. Inne rodzaje sieci charakteryzują się tym, że procesory są wielokrotnie aktywowane w ramach jednego przebiegu, w regularny sposób. Takie sieci są czasami nazywane skurczowymi, co wywodzi się od fizjologicznego terminu „skurcz”, który odnosi się do powtarzających się skurczów odpowiedzialnych za pompowanie krwi przez nasze ciała. Jako przykład rozważmy nauczyciela, który jest zainteresowany obliczeniem końcowych ocen N uczniów z kursu, na którym są egzaminy M. Każdy egzamin ma inną wagę, a ocena końcowa ma być średnią ważoną tych egzaminów M. Oceny wejściowe są ułożone w tablicy N na M, a wagi są podane w postaci listy ułamków M, w sumie 1. Dla każdego ucznia wymagane jest pomnożenie każdej oceny przez odpowiednią wagę, a następnie zsumowanie wyników M. Ostatecznym wynikiem ma być lista N ocen końcowych. Sekwencyjny algorytm wymagałby oczywiście czasu O(N × M), ponieważ dla każdej oceny należy wykonać jedno mnożenie, a każdy z N uczniów ma ocenę z każdego egzaminu M. Rysunek



przedstawia sieć skurczową do rozwiązania tego problemu (który w terminologii matematycznej nazywa się problemem mnożenia macierzy przez wektor). Sieć składa się z liniowego układu M połączonych procesorów. Szereg stopni, matryca, pod względem technicznym, jest wprowadzana do sieci od góry w pokazany sposób, po przekątnej na raz. W ten sposób sieć najpierw akceptuje ocenę ucznia 1 z egzaminu 1, następnie jednocześnie ocenę ucznia 2 z egzaminu 1 i ocenę ucznia 1 z egzaminu 2, a następnie jednocześnie ocenę ucznia 3 z egzaminu 1, ocenę 2 z egzaminu 2 i ocena 1 z egzaminu 3 i tak dalej. Ostatnim krokiem jest zaakceptowanie oceny ucznia N z egzaminu M. Z drugiej strony wektor wag jest stale dostępny wzdłuż dolnej linii. Zera są wprowadzane z lewej strony, a wektor wyjściowy jest tworzony z prawej strony, element po elemencie. (Ta liniowa konfiguracja jest czasami nazywana układem potoku, co odnosi się do sposobu, w jaki każdy procesor łączy wyjście do tego po swojej prawej stronie.) Każdy procesor z osobna jest niezwykle prosty: za każdym razem, gdy otrzymuje zestaw nowych danych wejściowych, po prostu mnoży jego górny i dolny, dodaje produkt po lewej stronie i wysyła wynik po prawej stronie. Algorytm indukowany przez tę sieć jest wyraźnie liniowy w N + M. Rysunek



przedstawia prostą symulację sieci krok po kroku dla przypadku czterech studentów i trzech egzaminów. Sieci skurczowe zostały skonstruowane w celu rozwiązania wielu problemów. Układ procesorów jest zwykle liniowy, prostokątny lub w kształcie rombu, a procesory są albo kwadratowe, co daje tak zwaną siatkę połączoną z siatką, albo sześciokątne, dające w wyniku ul, jak na rysunku



Czy paralelizm można wykorzystać do rozwiązania tego, co nierozwiązywalne?

Fakty, które omówiliśmy do tej pory, nie pozostawiają wątpliwości, że paralelizm można wykorzystać do poprawy zachowania czasowego algorytmów sekwencyjnych. Problemy, które wymagają pewnej ilości czasu na rozwiązanie sekwencyjne, można rozwiązywać szybciej, nawet w kategoriach rzędu wielkości, jeśli (rozszerzający się) równoległość jest dozwolona. Naturalne jest pytanie, czy paralelizm można wykorzystać do rozwiązywania problemów, których bez niego w ogóle nie dałoby się rozwiązać. Czy możemy opracować równoległy algorytm dla nierozstrzygalnego problemu? Odpowiedź brzmi: nie, ponieważ, jak wyjaśniono wcześniej, każdy algorytm równoległy może być symulowany sekwencyjnie przez jeden procesor, który biega i wykonuje pracę wszystkich w odpowiedniej kolejności. W tym sensie teza Churcha/Turinga odnosi się również do równoległych modeli obliczeń: klasa problemów rozwiązywalnych jest niewrażliwa nawet na dodanie rozszerzającego się paralelizmu. Następnym pytaniem, które należy zadać, jest to, czy równoległość może zmienić trudne problemy w wykonalne. Czy istnieje problem wymagający nieuzasadnionego (powiedzmy wykładniczego) czasu na sekwencyjne rozwiązanie, które można rozwiązać równolegle w rozsądnym (tj. wielomianowym) czasie? Aby móc lepiej docenić subtelność tego pytania, rozważmy najpierw problemy NP z Części 7. Jak zapewne pamiętasz, wszystkie problemy w NP mają rozsądne rozwiązania, które są niedeterministyczne; używają magicznej monety, która, jeśli zostanie rzucona, kiedy skonfrontowany z wyborem, użyje swojej magii, aby wskazać kierunek, który prowadzi do pozytywnej odpowiedzi, jeśli taki kierunek istnieje. Teraz, jeśli mamy nieograniczoną liczbę procesorów, nie potrzebujemy magicznej monety: kiedy dochodzi do „rozdroża", możemy po prostu wysłać nowe procesory, aby podążały za obiema możliwościami jednocześnie. Jeśli jeden z wysłanych procesorów kiedykolwiek wróci i powie „tak", cały proces zostanie zatrzymany i również powie „tak"; jeśli upłynął z góry określony czas wielomianu i nikt nie powiedział „tak", proces zatrzymuje się i mówi „nie". Ponieważ NP-ność problemu gwarantuje, że jeśli odpowiedź brzmi „tak", to rzeczywiście zostanie ona znaleziona przez magiczną monetę w wielomianowym okresie czasu, nasze wyczerpujące, wieloprocesorowe przemierzenie wszystkich możliwości znajdzie „tak" w tyle samo czasu. Jeśli tak nie jest, to odpowiedź musi brzmieć „nie". Konsekwencja jest jasna. Wszystkie problemy w NP, w tym te NP-zupełne, takie jak łamigłówki z małpami, komiwojażerowie i rozkłady jazdy, mają wielomianowe rozwiązania równoległe. Jednak zanim zaczniesz pospiesznie powiedzieć wszystkim, że niewykonalność jest tylko nieco uciążliwą konsekwencją konwencjonalnych jednoprocesorowych modeli obliczeń, i że można ją wyeliminować stosując obliczenia równoległe, należy poczynić trzy uwagi. Po pierwsze, liczba procesorów potrzebnych do rozwiązania problemu NP-zupełnego w rozsądnym czasie sama w sobie jest wykładnicza. Jeśli chcielibyśmy dowiedzieć się, w czasie krótszym niż miliardy lat pracy komputera, czy nasze lokalne liceum może opracować plan zajęć spełniający wszystkie ograniczenia, potrzebowalibyśmy całkowicie nierozsądnego komputera zawierającego biliony misternie połączonych procesorów. To jest coś, do czego powrócimy za chwilę. Druga uwaga jest zakorzeniona w fakcie, że problemy NP-zupełne nie są nierozwiązywalne – są jedynie przypuszczenia, że tak jest. Tak więc fakt, że możemy rozwiązywać problemy NP-zupełne równolegle w czasie wielomianowym, nie oznacza, że paralelizm może uwolnić problem od jego wrodzonej nierozwiązywalności, ponieważ nie wiemy, czy problemy NP-zupełne są faktycznie nierozwiązywalne. Wreszcie, nawet jeśli mamy algorytm równoległy, który używa tylko wielomianowej liczby instrukcji, ale wymaga wykładniczej liczby procesorów, nie jest jasne, czy naprawdę możemy uruchomić algorytm w czasie wielomianowym na rzeczywistym komputerze równoległym. W rzeczywistości istnieją wyniki, które pokazują, że przy dość liberalnych założeniach dotyczących szerokości linii komunikacyjnych i szybkości komunikacji, superwielomianowa liczba procesorów wymagałaby superwielomianowej ilości czasu rzeczywistego, aby wykonać nawet wielomianową liczbę kroków , bez względu na to, jak procesory są zapakowane razem. Wyniki te opierają się na nieodłącznych ograniczeniach przestrzeni trójwymiarowej. Tak więc

analiza złożoności obliczeń równoległych o nierozsądnych rozmiarach wydaje się wymagać czegoś więcej niż tylko oszacowania liczby kroków przetwarzania; Istotna jest na przykład ilość komunikacji. Pozostaje zatem pytanie: czy możemy użyć paralelizmu, nawet przy nadmiernie wielu procesorach, do rozwiązania w rozsądnym czasie problemu, który może okazać się nierozwiązywalny sekwencyjnie w rozsądnym czasie? I to pytanie jest nadal otwarte, jak teraz zostanie pokazane

Teza obliczeń równoległych

W Części 9 widzieliśmy, że teza Churcha/Turinga jest słuszna również w wyrafinowanym sensie, zgodnie z którym, zgodnie z bardzo naturalnymi założeniami, wszystkie sekwencyjne modele obliczeń są równoważne w obrębie wielomianu czasu. W konsekwencji trakcyjność, a nie tylko rozstrzygalność, jest niewrażliwa na wybór takiego modelu. Teza głosi, że sytuacja ta nie zmieni się wraz z proponowaniem nowych modeli. Podobne twierdzenie dotyczy równoległych modeli obliczeń. Zgodnie z naturalnymi założeniami, wszystkie dotychczas sugerowane uniwersalne modele paralelizmu rozszerzającego, w tym liczne warianty modeli pamięci współdzielonej, modele komunikacyjne, jednolite klasy sieci itp., mogą być równoważne w czasie wielomianowym. Każdy model może być symulowany przez drugi z co najwyżej wielomianową stratą czasu. Oznacza to, że podobnie jak w przypadku sekwencyjnym, klasa problemów rozwiązywanych równolegle w czasie wielomianowym jest również odporna na swój własny sposób; nie zależy to od konkretnego rodzaju wybranego modelu komputera równoległego ani od języka programowania użytego do jego programowania. Podobnie jak w przypadku sekwencyjnym, tutaj musimy być ostrożni. Modele, które pozwalają na współbieżny odczyt i zapis tej samej zmiennej, mogą być wykładniczo wydajniejsze niż te, które pozwalają na odczyt i zapis tylko przez jeden procesor na raz. Nie wchodząc w szczegóły należy stwierdzić, że aby to twierdzenie było prawdziwe, podstawowe instrukcje związane z jednoczesnym manipulowaniem elementami niskiego poziomu muszą mieć podobny charakter (tak jak podstawowe instrukcje manipulowania liczbami muszą być porównywalne w celu zachowania wielomianowej równoważności modeli sekwencyjnych). Fakt ten prowadzi do jednej części tezy o tzw. obliczeniach równoległych. Twierdzi, że i ta sytuacja nie ulegnie zmianie, ponieważ sugerowane są nowe modele. Innymi słowy, aż do różnic wielomianowych, ważne klasy złożoności dla obliczeń równoległych są również solidne i pozostaną takie nawet w miarę postępu nauki i technologii. Ta połowiczna teza jednak nie wystarcza, ponieważ mówi tylko o paralelizmie. Chcemy również wiedzieć, czy istnieje jakiś nieodłączny związek między sekwencyjnością a paralelizmem, jeśli chodzi o efektywność. Wiemy tylko, że równoległość nie pomaga w rozwiązywaniu całkowicie nierozwiązywalnych problemów; ale o ile lepiej może pomóc w rozwiązaniu problemów? Z pomocą przychodzi nam druga część pracy o obliczeniach równoległych. Twierdzi, że (ponownie, aż do różnic wielomianowych) czas równoległy jest taki sam, jak sekwencyjna przestrzeń pamięci. Jeśli możemy rozwiązać problem sekwencyjnie, używając pewnej ilości przestrzeni dla danych wejściowych o długości N, to możemy rozwiązać go równolegle w czasie, który nie jest gorszy niż wielomian w tej ilości przestrzeni. Może to być ta liczba do kwadratu, sześcianu, a może nawet podniesiona do setnej potęgi, ale nie będzie w tym wykładnicza. Odwrotna sytuacja również obowiązuje: jeśli możemy rozwiązać problem równolegle w określonym czasie dla danych wejściowych o długości N, możemy również rozwiązać go sekwencyjnie, używając przestrzeni pamięci ograniczonej wielomianem w tym czasie. Szczególnie interesujący jest szczególny przypadek tego ogólnego faktu, w którym pierwotna ilość miejsca sama w sobie jest wielomianem: każdy problem rozwiązywalny sekwencyjnie przy użyciu tylko wielomianowej ilości przestrzeni pamięci jest rozwiązywalny równolegle w wielomianowym okresie czasu i na odwrót . Symbolicznie:

Sekwencyjny-PSPACE = Równoległy-PTIME

Tak więc pytanie, czy istnieją nierozwiązywalne problemy, które stają się wykonalne wraz z wprowadzeniem paralelizmu, sprowadza się do tego, czy sekwencyjna klasa złożoności PSPACE zawiera

nierozwiązywalne problemy; to znaczy takie, co do których można udowodnić, że nie dopuszczają rozwiązań sekwencyjnych w czasie wielomianowym. To pytanie, sformułowane wyłącznie w kategoriach sekwencyjnych, jest nadal otwarte i, podobnie jak problem P vs. NP, jest prawdopodobnie również bardzo trudne.

Klasa Nicka: w kierunku rozsądnego równoległości

Ogólnie rzecz biorąc, wielomianowe algorytmy równoległe nie mogą być uważane za rozsądne, ponieważ mogą wymagać całkowicie nieuzasadnionej (tj. wykładniczej) liczby procesorów. Ponadto jednym z celów wprowadzenia równoległości jest radykalne skrócenie czasu pracy. W rzeczywistości jednym z celów jest tutaj znalezienie algorytmów podliniowych; czyli algorytmy, które są równoległe do tego stopnia, że nawet nie odczytują całego wejścia sekwencyjnie (w przeciwnym razie wymagałyby co najmniej liniowego czasu). Stąd wydaje się, że Parallel-PTIME nie jest najlepszym wyborem dla klasy problemów, które są wykonalne w obecności równoległości. Jaka jest zatem „właściwa" definicja podatności równoległej? Jedną z ciekawszych propozycji dotyczących tego zagadnienia jest klasa problemów zwana NC (klasa Nicka, od nazwiska jednego z jej pierwszych badaczy). Problem jest w NC, jeśli dopuszcza bardzo szybkie rozwiązanie równoległe, które wymaga tylko wielomianowo wielu procesorów. „Bardzo szybko" oznacza, że działa w czasie polilogarytmicznym; czyli w czasie jest to pewna stała potęga logarytmu N, jak O(log N) lub O((log N)²). Sumowanie wynagrodzeń, sortowanie, obliczanie średniej ważonej i wiele innych problemów są w NC (pierwsze dwa algorytmy opisane wcześniej, a trzeci algorytm nie opisany tutaj). Na przykład problemy NP-zupełne mogą, ale nie muszą być w NC-nie wiemy. Udało nam się jedynie wykazać, że dopuszczają rozwiązania równoległe w czasie wielomianowym; bycie w NC jest silniejszym wymogiem. Klasa NC jest solidna w tym samym sensie, co klasy takie jak P, NP i PSPACE. Pozostaje niewrażliwy na różnice między różnymi modelami obliczeń równoległych, a zatem może być używany w badaniu wrodzonej mocy równoległości. Można na przykład pokazać, że wszystkie problemy w NC są również w P (to znaczy w Sekwencyjnym-PTIME), ale nie wiadomo, czy jest odwrotnie: czy istnieje problem, który jest wykonalny w sensie sekwencyjnym (to jest w P), ale nie w tym równoległym sensie? Również tutaj, podobnie jak w pytaniu P vs. NP, większość badaczy uważa, że te dwie klasy są różne, a więc P różni się od NC. W szczególności problem znalezienia największego wspólnego dzielnika dwóch liczb całkowitych (jego gcd) występuje w P - Eukclidesowe znalazł dla niego algorytm wielomianowy już 2300 lat temu, jak wspomniano w rozdziale 1. Jednak problem gcd jest podejrzewa się, że nie jest w NC. Nikt nie wie, jak wykorzystać równoległość, aby znacząco przyspieszyć obliczanie tej najważniejszej wielkości, tak jak możemy przyspieszyć sortowanie i sumowanie wynagrodzeń. Oznacza to, że każdy krok klasycznego algorytmu gcd zależy od poprzednich kroków i nikt nie był w stanie znaleźć sposobu na usunięcie części tej zależności, aby można było z korzyścią wykorzystać równoległość. Tak więc mamy

NC ⊆ P ⊆ NP ⊆ PSPACE

a wielu informatyków uważa, że wszystkie trzy inkluzje są w rzeczywistości ścisłe. Przypominając wyżej wspomniane połączenie między PSPACE i Parallel-PTIME, ten zestaw przypuszczalnych nierówności można opisać (od prawej do lewej) mówiąc:

1. Istnieją problemy, które można rozwiązać w rozsądnej przestrzeni sekwencyjnej - to znaczy w rozsądnym czasie równoległym (ale nierozsądnym rozmiarze sprzętu) - których nie można rozwiązać w rozsądnym czasie sekwencyjnym, nawet przy magicznym niedeterminizmie.

2. Istnieją problemy, które można rozwiązać w rozsądnym czasie sekwencyjnym za pomocą magicznego niedeterminizmu, których bez niego nie da się rozwiązać w takim czasie.

3. Istnieją problemy, które można rozwiązać w rozsądnym czasie sekwencyjnym, których nie można rozwiązać w bardzo krótkim czasie równoległym przy rozsądnym rozmiarze sprzętu.

Jednak żadna z tych nierówności nie jest tak naprawdę ścisła, a zatem jest po prostu możliwe (choć bardzo mało prawdopodobne), że te klasy problemów są w rzeczywistości wszystkie równe. Należy wspomnieć, że podobnie jak w przypadku NP vs P, pytanie P vs NC również rodzi naturalne pojęcie kompletności. Tak więc, chociaż nie wiemy, czy P = NC, bardzo pomocne jest wykazanie, że problem jest P-zupełny, co oznacza, że jeśli jest w NC, to wszystkie problemy w P również są w NC.

Współbieżność rozproszona i ciągła

Byłoby miło móc zakończyć w tym miejscu dyskusję na temat jednoznacznej jednoczesności. A dlaczego nie? Omówiliśmy algorytmy równoległe i ich złożoność oraz zobaczyliśmy, jak mogą one ulepszyć sytuację i do jakiego stopnia. Przyznaliśmy nawet, że nie wiemy o paralelizmie prawie tyle, ile powinniśmy. Czego zatem brakuje? Cóż, wprowadzenie paralelizmu w celu efektywniejszego rozwiązywania konwencjonalnych problemów algorytmicznych to tylko jeden z jego aspektów. Drugi dotyczy sytuacji, w których paralelizm nie jest czymś, co wprowadzamy, aby coś poprawić, ale czymś, z czym musimy żyć, ponieważ po prostu tam jest. Wiąże się to również z innym rodzajem problemu algorytmicznego, który nie zawsze wiąże się z przekształcaniem danych wejściowych podanych na początku w pożądane wyniki, które są wytwarzane na końcu. Jest to raczej to, co można znaleźć w wielu (a właściwie większości!) reaktywnych i wbudowanych systemach, które omówimy w części 14. Problem polega na określeniu protokołów pożądanego zachowania w czasie, aby zagwarantować różne właściwości wymagane od tego zachowania trzymać. To, co szczególnie utrudnia te problemy, to fakt, że w wielu przypadkach protokół, który należy opisać jako ich rozwiązanie, w ogóle nie musi się kończyć. Musi po prostu siedzieć tam na zawsze, robiąc rzeczy, które zawsze są zgodne z tymi wymogami. Aby rozróżnienie było jasne, w tym przypadku użyjemy terminu współbieżność, a nie równoległość. W przeciwieństwie do Części 13 i 14, tutaj nie interesuje nas ogólny problem metod i języków służących do inżynierii rozwoju dużych i złożonych systemów, ale pojawiające się tam problemy algorytmiczne na małą skalę, ale zawsze tak subtelne. Rozważmy przykład. Załóżmy, że pewien hotel ma tylko jedną łazienkę na każdym piętrze. (Wiele niedrogich hoteli jest tego typu.) Załóżmy też, że ogrzewanie jest tylko w pokojach - na korytarzu jest nieprzyjemnie chłodno. (Wiele niedrogich hoteli również spełnia to założenie.) Co jakiś czas goście muszą brać prysznic. (Większość gości w takich hotelach rzeczywiście to robi). Jak powinni się zająć zaspokajaniem tych potrzeb? Nie mogą ustawić się w kolejce przed drzwiami łazienki, z powodu związanego z tym czynnika drżenia. Jeśli gość po prostu raz na jakiś czas spróbuje otworzyć drzwi łazienki, może pozostać nieczysty na zawsze, ponieważ całkiem możliwe, że za każdym razem będzie zajęty. Dzieje się tak pomimo faktu, że prysznice zajmują tylko skończoną ilość czasu; ktoś inny może zawsze wejść pierwszy. Zakładamy, że goście pochodzą z różnych krajów i wszyscy mówią różnymi językami, więc bezpośrednia komunikacja nie wchodzi w rachubę. Jedno z oczywistych rozwiązań wymaga przymocowania małej tablicy do zewnętrznej części drzwi do łazienki. Każdy gość wychodzący z łazienki kasuje zapisany na tablicy numer pokoju (będzie on jego) i zapisuje numer kolejnego pokoju w ustalonej kolejności. (Pomyśl o pokojach na piętrze jako rozmieszczonych cyklicznie, tak aby każdy pokój miał swojego niepowtarzalnego poprzednika i następcę). na tablicy i wracać do swojego pokoju, jeśli tak nie jest. Najwyraźniej każdy gość w końcu dostanie swoją kolej na prysznic. Jednak to rozwiązanie nadal ma poważne problemy. Po pierwsze, możliwe jest, że pokój numer 16 nigdy nie jest zajęty, że jego mieszkaniec nigdy nie chce wziąć prysznica, albo został właśnie uprowadzony i nigdy więcej go nie widziano. Na tablicy będzie wtedy na zawsze napisane 16 i żaden z pozostałych gości nigdy nie będzie czysty. Po drugie, nawet jeśli we wszystkich pomieszczeniach mieszkają żywi, dostępni ludzie, to rozwiązanie narzuca wszystkim gościom niemożliwą do spełnienia dyscyplinę jeden prysznic na cykl,

zmuszając tych bardziej wybrednych do brania prysznica tak rzadko, jak tych, którym nie zależy na tym. Problem ten ilustruje niektóre z głównych problemów pojawiających się podczas projektowania systemów, które są z natury rozproszone. Dystrybucja to szczególny rodzaj współbieżności, w którym współbieżne komponenty są fizycznie oddalone. W związku z tym, ludzie troszczą się nie tylko o to, co każdy komponent powinien zrobić i jak należy wykonywać inżynierię oprogramowania i zarządzanie projektami, ale także o zminimalizowanie ilości komunikacji, która ma miejsce między komponentami. Komunikacja, czy to jawna, czy niejawna (powiedzmy w postaci pamięci współdzielonej), może stać się niezwykle kosztowna w systemie rozproszonym. Tak jak poprzednio, będziemy używać terminu „procesory" dla oddzielnych podmiotów lub komponentów w systemie rozproszonym. Procesory w z natury współbieżnym lub rozproszonym systemie nie muszą osiągać jedynie relacji wejścia/wyjścia, ale raczej wykazywać określone pożądane zachowanie w czasie. W rzeczywistości system może w ogóle nie być zmuszony do zakończenia pracy, tak jak w problemie z prysznicem. Części pożądanego zachowania można określić jako ograniczenia globalne. Na przykład prysznic niesie ze sobą ograniczenie mówiące, że może pomieścić najwyżej jedną osobę na raz, a inne istotne ograniczenie mówi, że każdy potrzebuje od czasu do czasu prysznica. Tak więc prysznic jest tak naprawdę kluczowym zasobem, takim, jakiego każdy potrzebuje przez pewien ograniczony czas, po którym można z niego zrezygnować na rzecz kogoś innego. Innym ważnym ograniczeniem w problemie z prysznicem jest to, że musimy zapobiegać sytuacjom impasu, w których żaden procesor nie może zrobić postępu (na przykład w rozwiązaniu tablicy, gdy gość 16 nigdy się nie pojawia) oraz sytuacjom głodu - czasami nazywanych blokadami - w któremu co najmniej jednemu procesorowi, ale nie każdemu z nich, nie udaje się zrobić postępu (na przykład, gdy goście są poinstruowani, aby po prostu spróbowali od czasu do czasu drzwi do łazienki, a pechowcy mogą zostać na zawsze). Pojęcia te są typowe dla wielu rzeczywistych systemów, takich jak omówione w częściach 13 i 14. Na przykład większość standardowych komputerów ma kilka kluczowych zasobów, takich jak napędy taśm i dysków, drukarki i plotery oraz kanały komunikacyjne. W pewnym sensie pamięć komputera może być również postrzegana jako kluczowy zasób, ponieważ nie chcielibyśmy, aby dwa działające razem zadania jednocześnie zapisywały w tej samej lokalizacji. Innym przykładem jest system rezerwacji lotów, który jest bardzo rozpowszechniony. Może składać się z ogólnoświatowej sieci zawierającej jeden duży komputer w Nowym Jorku, drugi w Los Angeles i 2000 terminali. Byłoby żenujące, gdyby dwa z terminali przydzielały miejsce 25D na ten sam lot dwóm różnym pasażerom. Nie można również zakończyć systemu rezerwacji lotów. Musi nadal działać, stale udostępniając swoje zasoby wszystkim procesorom, którzy o to poproszą, jednocześnie zapobiegając zakleszczeniu i głodowi. To naprawdę nie są problemy algorytmiczne w zwykłym znaczeniu tego słowa. Obejmują różne rodzaje wymagań i nie są rozwiązywane przez zwykłe algorytmy. Rozwiązanie musi dyktować protokoły algorytmiczne dla zachowania każdego z procesorów, które gwarantują spełnienie wszystkich wymagań i ograniczeń określonych w zadaniu oraz przez cały (potencjalnie nieskończony) czas życia systemu.

Rozwiązywanie problemów z prysznicem w hotelu

Tablica sugerowana dotycząca problemu kąpieli pod prysznicem w niedrogich hotelach może być traktowana jako pamięć współdzielona, a prysznic jako kluczowy zasób. Zauważ, że omijaliśmy problem konfliktów w pisaniu, pozwalając na pisanie na tablicy tylko jednemu gościowi w tym samym czasie. Opiszmy teraz satysfakcjonujące rozwiązanie tego problemu. Właściwie rozwiążemy trudniejszy problem, w którym łazienki są w każdym pokoju, ale jednorazowo tylko jeden gość może brać prysznic. (Powód może być związany z ciśnieniem lub temperaturą wody, co jest kolejnym rozsądnym założeniem w przypadku niedrogich hoteli.) Ta sytuacja jest wyraźnie bardziej delikatna, ponieważ nie ma bezpośredniego sposobu, aby dowiedzieć się, czy ktoś rzeczywiście bierze prysznic. Niemniej jednak rozwiązanie musi zapewniać, że wszyscy biorą prysznic, ale dwoje gości nigdy nie bierze prysznica jednocześnie. Bardziej ogólny opis problemu jest następujący. Istnieje N procesorów, z

których każdy musi wielokrotnie wykonywać pewne „prywatne" czynności, po których następuje sekcja krytyczna:

cykl życia I procesora:

(1) wykonaj następujące czynności raz za razem:

(1.1) wykonywać prywatne czynności (np. jeść, czytać, spać);

(1.2) przeprowadzić sekcję krytyczną (na przykład wziąć prysznic).

Prywatne działania to moja własna działalność przetwórcy - nie mają nic wspólnego z nikim innym. Z drugiej strony, sekcja krytyczna to sprawa wszystkich, ponieważ żadne dwa procesory nie mogą znajdować się w swojej sekcji krytycznej jednocześnie. Problem polega na tym, aby znaleźć sposób, aby procesory N mogły żyć wiecznie, bez impasu i głodu, przy jednoczesnym poszanowaniu krytycznej natury sekcji krytycznych. Musimy poinstruować procesory, aby wykonywały określone czynności, takie jak sprawdzanie tablic i pisanie na nich, przed i/lub po wejściu w ich krytyczne sekcje, aby te wymagania zostały spełnione. Przedstawiony w tej formie problem bywa nazywany problemem wzajemnego wykluczania, ponieważ procesorom należy zagwarantować wyłączność w zakresie wchodzenia w krytyczne sekcje.

Omówimy rozwiązanie w przypadku dwóch gości, czyli procesorów, P_1 i P_2. Bardziej ogólny przypadek dla procesorów N jest przedstawiony później. Użyjemy trzech zmiennych, X_1, X_2 i Z, które w przykładzie z prysznicem są reprezentowane przez trzy małe obszary na tablicy wiszącej na korytarzu. Z może wynosić 1 lub 2, a oba procesory mogą zmieniać jego wartość. W takich przypadkach mówimy, że Z jest zmienną wspólną. Znaki X mogą być tak lub nie i mogą być odczytywane przez oba procesory, ale zmieniane tylko przez procesor z odpowiednim indeksem; to znaczy, P1 może zmienić X_1, a P_2może zmienić X_2 W takich przypadkach mówimy, że X są zmiennymi rozłożonymi. Początkowa wartość obu X to nie, a Z to 1 lub 2, to nie ma znaczenia. Oto protokoły dla dwóch procesorów:

protokół dla P_1:

(1) wykonaj następujące czynności raz za razem:

(1.1) prowadzić działalność prywatną do czasu, gdy pożądane jest wejście do sekcji krytycznej;

(1.2) $X_1 \leftarrow$ tak;

(1.3) $Z \leftarrow 1$;

(1.4) poczekaj, aż X_2 stanie się nie lub Z stanie się 2 (lub oba);

(1.5) przeprowadzić sekcję krytyczną;

(1.6) $X_1 \leftarrow$ nie.

protokół dla P_2:

(1) wykonaj następujące czynności raz za razem:

(1.1) prowadzić działalność prywatną do czasu, gdy pożądane jest wejście do sekcji krytycznej;

(1.2) $X_2 \leftarrow$ tak;

(1.3) $Z \leftarrow 2$;

(1.4) czekać, aż X_1 stanie się nie lub Z stanie się 1 (lub oba);

(1.5) przeprowadzić sekcję krytyczną;

(1.6) $X_2 \leftarrow$ nie.

Co się tutaj dzieje? Pomyśl o X_1 jako o tym, że P_1> chce wejść do jego krytycznej sekcji: X_1 oznacza, że tak oznacza, że chciałby wejść, a X_1 nie oznacza, że zakończył i chciałby wrócić do jego prywatnej działalności. X_2 odgrywa tę samą rolę dla p_2 Trzecia zmienna, Z, jest pewnego rodzaju wskaźnikiem grzecznościowym: po tym, jak P1 jasno określił, że chce wejść do swojej sekcji krytycznej, natychmiast ustawia Z na 1, wskazując w bardzo hojny sposób, że jeśli chodzi o , P_2może wejść, jeśli chce. Następnie czeka, aż albo P_2 wskaże, że opuścił własną sekcję krytyczną i wróci do swoich prywatnych działań (to znaczy, że X_2 oznacza nie), albo P_2 ustawi zmienną grzecznościową Z na 2, hojnie dając P_1 pierwszeństwo . P_2 działa w podobny sposób. (Możemy myśleć o Z jako o reprezentowaniu podpisów procesorów w dzienniku. Ostatnim, który się loguje, jest ten, który ostatnio dał drugiemu pierwszeństwo.) Oczekiwanie w klauzuli (1.4) jest czasami nazywane zajętym czekanie, ponieważ procesor nie może po prostu bezczynnie, dopóki nie zostanie poparty przez kogoś innego. Raczej aby nadal sprawdzać wartości Z i samej zmiennej X drugiego procesora, nie robiąc w międzyczasie nic więcej. Zobaczmy teraz, dlaczego to rozwiązanie jest poprawne. Cóż, przede wszystkim twierdzimy, że P_1 i P_2 nie mogą być w swoich krytycznych sekcjach jednocześnie. Załóżmy, że mogą. Oczywiście, w czasie, gdy znajdują się w swoich krytycznych sekcjach razem, zarówno X_1, jak i X_2mają wartość tak, więc ostatni procesor, który wszedł, musiał wejść na mocy części Z swojego testu oczekiwania w klauzuli (1.4), ponieważ wartość X innych była tak. Oczywiście oba procesory nie mogły przejść testów w tym samym czasie. (Dlaczego?) Powiedzmy, że P_1 zdał test jako pierwszy. Wtedy Z musiało wynosić 1, gdy P_2 później zdał test. Oznacza to, że między momentem, w którym P_1 ustawił Z na 2 w swoim punkcie (1.3) a momentem, w którym przeszedł test w punkcie (1.4), P_1musiał ustawić Z na 1 i od tego momentu P_1 nie robił nic więcej, dopóki nie wszedł jego sekcję krytyczną ze względu na fakt, że Z wynosi 1. Ale w jaki sposób P_1 zdołał wcześniej wejść do swojej sekcji krytycznej? Nie mógł wejść ze względu na fakt, że Z wynosi 2, ponieważ, jak już powiedzieliśmy, ustawił Z na 1 po tym, jak P_1 ustawił go na 2. Ponadto nie mógł wejść z tego powodu, że X_1 jest nie, ponieważ X_2 było ustawiony na tak przez P_2, zanim ustawi Z na 2. A zatem nie ma możliwości, aby P_2 wszedł w swoją sekcję krytyczną, o ile P_1 znajduje się we własnej sekcji krytycznej. Podobny argument dotyczy podwójnej możliwości, gdzie P_1 wchodzi, podczas gdy P_2 już jest. Wniosek jest taki, że rzeczywiście obowiązuje wzajemne wykluczenie. Zobaczmy, dlaczego głód i impas są niemożliwe. (W tym przypadku głód pojawia się, gdy jeden z procesorów chce wejść do swojej krytycznej sekcji, ale na zawsze nie może tego zrobić, a zakleszczenie występuje, gdy żaden z procesorów nie może poczynić postępów.) Pierwszą rzeczą, na którą należy zwrócić uwagę, jest to, że zablokowanie obu procesorów jest niemożliwe. w ich klauzulach (1.4) razem, ponieważ Z jest zawsze 1 lub 2, a zatem jeden z nich może zawsze uwolnić się od oczekiwania. Teraz załóżmy, że P_1 jest zmuszony do pozostania w swojej klauzuli (1.4), podczas gdy P_2 nie. Jeśli P_2 nigdy nie chce wejść do swojej sekcji krytycznej, to znaczy pozostaje w klauzuli (1.1), wartość X_2 będzie wynosić nie, a P_1 będzie mógł wejść, kiedy tylko zechce. Jedyną inną możliwością jest to, że P_2 kontynuuje cykle wokół swojego protokołu na zawsze. W takim przypadku prędzej czy później ustawi Z na 2, a ponieważ nigdy nie może zmienić go z powrotem na 1 (może to zrobić tylko P_1), P_1 w końcu będzie mógł przejść test i wejść (1.5). W ten sposób zapobiega się głodowi i impasowi, a rozwiązanie spełnia wszystkie wymagania.

Rzeczy są trudniejsze niż się wydaje

To dwuprocesorowe rozwiązanie wydaje się na pierwszy rzut oka dość elementarne, a argument dotyczący poprawności, choć nieco zawikłany, również nie wygląda na zbyt skomplikowany. Jednak taka prostota może być dość zwodnicza. Jako ilustrację tego, jak naprawdę delikatne są rzeczy, przestudiujmy niewielką odmianę powyższego rozwiązania. Co by się stało, gdybyśmy zamienili

kolejność klauzul (1.2) i (1.3) w protokołach? Innymi słowy, gdy na przykład P_1 chce wejść do swojej sekcji krytycznej, najpierw robi P_2 dzięki uprzejmości ustawienia Z na 1, przepuszczając go, jeśli chce, i tylko następnie ustawia X_1 na tak, aby wskazać, że chce wejść. Na pierwszy rzut oka wydaje się, że nie ma różnicy: P_1 może nadal wejść do swojej sekcji krytycznej tylko wtedy, gdy P_2 jest albo bezinteresowny (to znaczy, jeśli X_2 jest nie) lub wyraźnie daje P_1 pierwszeństwo (to znaczy, jeśli Z wynosi 2). Co może pójść źle? W tym momencie powinieneś spróbować przepracować podany nieformalny dowód poprawności, ale w przypadku poprawionej wersji, aby zobaczyć, gdzie zawodzi. To się nie udaje, a oto scenariusz, który prowadzi do tego, że oba procesory znajdują się jednocześnie w swoich krytycznych sekcjach. Początkowo, zgodnie z ustaleniami, X_1 to nie, a X_2 to nie, a oba procesory są zajęte swoimi prywatnymi czynnościami. Teraz pojawia się następująca sekwencja działań:

1. P_2 ustawia Z na 2;

2. P_1 ustawia Z na 1;

3. P_1 ustawia X1 na tak;

4. P_1 wchodzi w swoją sekcję krytyczną, ponieważ X_2 jest nie;

5. P_2 ustawia X_2 na tak;

6. P_1 wchodzi w swoją sekcję krytyczną, ponieważ Z wynosi 1.

Problem polega oczywiście na tym, że teraz P_2 może wziąć prysznic (to znaczy wejść do swojej sekcji krytycznej), mimo że P_1 sam bierze prysznic, ponieważ P_1 był ostatnim, który był uprzejmy, a zatem wartość Z wynosi 1, gdy P_2 wskazuje jego chęć wejścia. W oryginalnym protokole było to niemożliwe, ponieważ ostatnią rzeczą, jaką musiał zrobić P2 przed wejściem, było ustawienie Z na niekorzystną wartość 2. Tak więc nawet to pozornie proste rozwiązanie dla dwóch procesorów kryje w sobie pewną subtelność. Przyjrzyjmy się teraz ogólnemu rozwiązaniu dla procesorów N, które jest jeszcze delikatniejsze.

Rozwiązanie problemu wzajemnego wykluczania procesorów N

Gdy gości jest więcej niż dwóch, nie wystarczy powiedzieć „chciałbym wziąć prysznic", a następnie być uprzejmym wobec jednego z pozostałych. Nie pomoże też być uprzejmym dla wszystkich innych razem i mieć nadzieję, że wszystko się ułoży. Musimy uczynić nasze procesory bardziej wyrafinowanymi. Procesorzy zostaną poinstruowani, aby zwiększyć nacisk na wejście do sekcji krytycznej, gdy będą czekać. W ten sposób każdy procesor przejdzie przez pętlę, której wykonanie będzie odpowiadać wyższemu poziomowi nacisku. Liczba poziomów to dokładnie N, całkowita liczba procesorów, a pierwszy poziom (właściwie zero) wskazuje na brak zainteresowania wejściem do sekcji krytycznej. Każdy z poziomów nalegań ma swoją własną zmienną grzecznościową, a na każdym poziomie procesor jest uprzejmy dla wszystkich innych, wpisując w tej zmiennej własną liczbę. Gdy przetwórca zasygnalizował chęć zwiększenia poziomu nalegań i był uprzejmy dla wszystkich innych, czeka z podwyższeniem poziomu, aż albo wszyscy inni będą mniej natarczywi niż on sam, albo otrzyma uprzejmość i przystąpi do następny poziom przez kogoś innego. Sekcja krytyczna jest ostatecznie wprowadzana, gdy pojawi się zielone światło umożliwiające przekroczenie najwyższego poziomu. Po opuszczeniu sekcji krytycznej procesor rozpoczyna całą procedurę od nowa na poziomie zerowym. Oto protokół dla I procesora. W nim wektor Z jest indeksowany przez poziomy nalegań, a nie przez procesory. Zatem procesor I ustawia Z[J] na I (nie Z[I] na J, jak w przypadku wektora X), aby wskazać, że na poziomie J daje pierwszeństwo wszystkim innym. Warto zauważyć, że jeśli N = 2 to protokoły te

są dokładnie tymi właśnie przedstawionymi, więc jest to rzeczywiście bezpośrednie rozszerzenie rozwiązania dwuprocesorowego.

protokół dla I procesora, PI:

(1) wykonaj następujące czynności raz za razem:

(1.1) prowadzić działalność prywatną do czasu, gdy pożądane jest wejście do sekcji krytycznej;

(1.2) dla każdego J od 1 do N − 1 wykonaj następujące czynności:

(1.2.1) X[I] ← J ;

(1.2.2) Z[J] ← I ;

(1.2.3) czekaj, aż X[K] < J dla wszystkich K = I lub Z[J] = I ;

(1.3) przeprowadzić sekcję krytyczną:

(1.4) X[I] ← 0.

Możliwe jest dostarczenie nieformalnego dowodu poprawności dla tego ogólnego przypadku, na wzór dowodu dla przypadku dwuprocesorowego. Tutaj pokazujemy, że na każdym poziomie procesory działają tylko jeden na raz, tak że w szczególnym przypadku w danej chwili w krytycznej sekcji znajduje się najwyżej jeden procesor. Podobnie ustala się wolność od impasu i głodu.

Właściwości bezpieczeństwa i żywotności

Rozproszone systemy współbieżne są pod wieloma względami bardziej wymagające niż normalne systemy sekwencyjne. Trudności zaczynają się już od samego zadania sprecyzowania problemu i pożądanych właściwości rozwiązania. „Problemy algorytmiczne" związane z takimi systemami nie mogą być już dłużej definiowane naiwnie poprzez określenie zestawu legalnych danych wejściowych wraz z funkcją opisującą pożądane wyniki. Poprawność nie polega już po prostu na zakańczaniu i generowaniu poprawnych wyjść, a wydajności nie można już naiwnie mierzyć liczbą kroków wykonanych przed zakończeniem lub rozmiarem używanej pamięci. W związku z tym techniki opracowane w celu udowodnienia poprawności lub szacowania wydajności konwencjonalnych algorytmów są niewystarczające, jeśli chodzi o protokoły rozwiązujące problemy o charakterze współbieżnym i rozproszonym. Niektóre aspekty tych problemów omówiono w dalszej części, w częściach 13 i 14. Okazuje się, że większość wymagań dotyczących poprawności protokołów dla bieżących systemów współbieżnych można podzielić na dwie główne kategorie, właściwości bezpieczeństwa i żywotności. Właściwości bezpieczeństwa stwierdzają, że pewne „złe" rzeczy nigdy nie mogą się wydarzyć lub, równoważnie, że odpowiadające im „dobre" rzeczy zawsze będą miały miejsce, a właściwości żywotności stwierdzają że pewne „dobre" rzeczy w końcu się wydarzą. Wzajemne wykluczanie - zapobieganie jednoczesnemu przebywaniu dwóch procesorów w ich krytycznych sekcjach - jest właściwością bezpieczeństwa, ponieważ wymaga gwarancji, że zawsze będzie co najwyżej jeden procesor w krytycznej sekcji, podczas gdy zapobieganie głodowi jest właściwością żywotności, ponieważ wymaga dla gwarancji, że każdy procesor, który chce wejść do swojej krytycznej sekcji, w końcu będzie mógł to zrobić. Co ciekawe, częściowa poprawność i właściwości terminacyjne konwencjonalnych algorytmów są szczególnymi przypadkami bezpieczeństwa i żywotności. Częściowa poprawność oznacza, że program nigdy nie może kończyć się z niewłaściwymi danymi wyjściowymi – wyraźnie jest to właściwość bezpieczeństwa – a zakończenie oznacza, że algorytm musi w końcu osiągnąć swój logiczny koniec i zakończyć – wyraźnie właściwość żywotności. Aby wykazać, że dany protokół narusza właściwość bezpieczeństwa, wystarczy wykazać

skończony, zgodny z prawem ciąg czynności, który prowadzi do sytuacji zabronionej. Dokonano tego w przypadku błędnej wersji dwuprocesorowego rozwiązania problemu z prysznicem i jest to po prostu pokazanie, że algorytm narusza częściową poprawność, wyświetlając kończącą sekwencję wykonywania z błędnymi wynikami. Zupełnie inną sprawą jest pokazanie, że zostaje naruszona właściwość życia. Musimy jakoś udowodnić istnienie nieskończonej sekwencji działań, która nigdy nie prowadzi do obiecanej sytuacji. Ta różnica jest w rzeczywistości jednym ze sposobów scharakteryzowania dwóch klas właściwości. Jeśli chodzi o ciągłą współbieżność, techniki testowania zwykle nie są zbyt pomocne. Możemy być rozsądnie przekonani o poprawności zwykłego algorytmu sortującego, wypróbowując go na wielu standardowych listach elementów oraz w niektórych przypadkach „z pogranicza", takich jak: jako listy z jednym elementem lub bez, lub listy z równymi wszystkimi elementami. Z drugiej strony wiele błędów pojawiających się w sferze systemów współbieżnych nie ma nic wspólnego z nieoczekiwanymi danymi wejściowymi. Wynikają one z nieoczekiwanych interakcji między procesorami oraz nieprzewidywalnej kolejności wykonywania akcji. Udowodnienie poprawności takich systemów jest zatem bardziej śliskie i podatne na błędy niż konwencjonalne algorytmy sekwencyjne. Wiele opublikowanych „rozwiązań" współbieżnych problemów programistycznych okazało się później zawierać subtelne błędy, które wymykały się zarówno recenzentom, jak i czytelnikom, pomimo obecności pozornie ważnych argumentów za poprawnością. Potrzeba formalnej weryfikacji jest więc tutaj bardziej dotkliwa niż w przypadku sekwencyjnym i rzeczywiście zaproponowano kilka podejść. Zazwyczaj te, które zajmują się właściwościami bezpieczeństwa, są rozszerzeniem metody pośrednio-potwierdzenia do weryfikacji częściowej poprawności, a te dostosowane do właściwości związanych z żywotnością rozszerzają metodę zbieżności do udowodnienia zakończenia. Wykazanie, że np. algorytm scalania równoległego jest częściowo poprawny, wymaga m.in. formalnego dowodu, że dwa procesory przypisane do równoległego sortowania dwóch połówek listy, na dowolnym poziomie rekurencji, nie kolidują ze sobą. inny. W tym konkretnym przypadku procesory pracujące równolegle w ogóle nie wchodzą w interakcje, a taki dowód jest stosunkowo łatwy do zdobycia. Kiedy procesory wchodzą w interakcję, jak w przypadku rozwiązania problemu z prysznicem, sprawy stają się znacznie trudniejsze. Musimy w jakiś sposób zweryfikować zachowanie każdego pojedynczego procesora w izolacji, biorąc pod uwagę wszystkie możliwe interakcje z innymi, a następnie połączyć dowody w jedną całość. Asercji pośrednich nie można tutaj używać w zwykły sposób. Twierdzenie, że twierdzenie jest prawdziwe za każdym razem, gdy zostanie osiągnięty pewien punkt w protokole procesora P, nie zależy już wyłącznie od działań P. Co więcej, nawet jeśli twierdzenie jest rzeczywiście prawdziwe w tym momencie, może stać się fałszywe przed następnym działaniem P, ponieważ inny procesor był wystarczająco szybki, aby w międzyczasie zmienić wartość jakiejś zmiennej. Pewne formalne metody dowodowe przezwyciężają ten problem, wymagając oddzielnych dowodów, aby spełnić pewien rodzaj własności wolności ingerencji, która musi być udowodniona oddzielnie. Wszystko to brzmi dość skomplikowanie. W rzeczywistości tak jest. Jednym z powodów przedstawienia jedynie nieformalnego dowodu poprawności dla prostego rozwiązania problemu wzajemnego wykluczania dwóch procesorów był raczej techniczny charakter dowodu formalnego. W zasadzie jednak istnieją odpowiednie metody dowodowe i, jak w przypadku sekwencyjnym, dowody formalne istnieją zawsze, jeśli rozwiązania są rzeczywiście poprawne, chociaż ich odkrycie jest w zasadzie nieobliczalne.

Logika temporalna

W poprzednich rozdziałach wspominano logikę dynamiczną. Są to ramy formalne, które można wykorzystać do określenia i udowodnienia różnych właściwości algorytmów. Byłoby miło, gdybyśmy mogli wykorzystać taką logikę do określania i udowadniania właściwości bezpieczeństwa i żywotności systemów współbieżnych. Jak wspomniano w części 5, centralną konstrukcją logiczną używaną w logikach dynamicznych jest after(A, F), co oznacza, że F jest prawdziwe po zakończeniu A. Taka logika

opiera się zatem na paradygmacie wejścia/wyjścia - paradygmacie odniesienia sytuacji przed wykonaniem części algorytmu do sytuacji po wykonaniu. W przypadku współbieżności niezbędna wydaje się umiejętność mówienia bezpośrednio także o tym, co dzieje się podczas egzekucji. O wiele bardziej pasującym tu formalizmem jest logika temporalna, czyli TL. Jest to odmiana logiki klasycznej znanej matematykom jako logika czasów gramatycznych, specjalnie dostosowana do celów algorytmicznych. Jej formuły zawierają stwierdzenia o prawdziwości twierdzeń w miarę upływu czasu, a także mogą wyraźnie odnosić się do aktualnego miejsca kontroli w protokołach. Dwie z centralnych konstrukcji to odtąd (F) i ostatecznie (F). Pierwsza stwierdza, że F jest prawdziwe od teraz — aż do zakończenia, jeśli protokół się zakończy, i na zawsze, jeśli tak się nie stanie — a druga mówi, że F w końcu stanie się prawdą, to znaczy w pewnym momencie w przyszłości. Przypominając nasze użycie symbolu ~ do oznaczenia „nie", pierwsza z tych konstrukcji może być użyta do określenia właściwości bezpieczeństwa, pisząc odtąd (~F) (czyli F nigdy nie stanie się prawdą), a druga do określenia żywotności nieruchomości. Jako przykład rozważ rozwiązanie problemu prysznica dwuprocesorowego i następujące formuły TL:3

P1-jest -na (1.4) → ostatecznie (P1-jest na (1.5))

P2-jest-na(1.4) → ostatecznie (P2-w-(1.5))

∿ (P1-jest przy-(1.5) & P2-jest-(1.5))

Pierwsze dwie formuły stwierdzają, że jeśli procesor czeka na wejście do sekcji krytycznej, w końcu wejdzie, a trzecia stwierdza, że oba procesory nigdy nie znajdą się w swojej sekcji krytycznej jednocześnie. Wzięte razem i stwierdzone, że są prawdziwe we wszystkich punktach obliczeń (używane odtąd), formuły te zapewniają, że rozwiązanie jest poprawne. Formalnym dowodem na to jest logiczna manipulacja, która trzyma się ścisłych aksjomatycznych reguł logiki temporalnej, o których tutaj nie będziemy się rozwodzić. Jednak, aby dokonać porównania z metodami dowodowymi z rozdziału 5, okazuje się że czyste właściwości bezpieczeństwa (takie jak w trzecim wzorze powyżej) mogą być udowodnione w sposób podobny do metody niezmiennej asercji dla częściowej poprawności. Ewentualności (takie jak te w pierwszych dwóch wzorach) można udowodnić, najpierw analizując każdy możliwy pojedynczy krok protokołów, a następnie stosując indukcję matematyczną, aby wydedukować ewentualności, które stają się prawdziwe w ciągu pewnej liczby kroków od tych, które stają się prawdziwe w mniej. Jak wspomniano, jeśli protokoły są rzeczywiście poprawne, istnieje dowód poprawności, a jeśli zostanie znaleziony, jego części można sprawdzić algorytmicznie. W rzeczywistości w wielu zastosowaniach logiki temporalnej każda zmienna ma tylko skończoną liczbę możliwych wartości (omówiony wcześniej protokół wzajemnego wykluczania wykorzystuje trzy zmienne, każda z tylko dwiema możliwymi wartościami). Te protokoły skończonych stanów, jak się je czasami nazywa, można zweryfikować algorytmicznie. Oznacza to, że istnieją algorytmy, które akceptują jako dane wejściowe pewne rodzaje protokołów i formułę logiki temporalnej zapewniającą poprawność i weryfikują te pierwsze z tymi drugimi. Tak więc, o ile znalezienie dowodów na poprawność transformacyjnych lub obliczeniowych części współbieżnych systemów nie jest na ogół możliwe, możliwe jest skuteczne znalezienie dowodów na części kontrolne, takie jak mechanizmy harmonogramowania procesorów i zapobiegania głodowaniu i zakleszczeniu, jak zazwyczaj dotyczą one tylko zmiennych o skończonym zakresie. Jednak te automatyczne weryfikatory nie zawsze są tak wydajne, jak byśmy chcieli, ponieważ między innymi liczba kombinacji wartości rośnie wykładniczo wraz z liczbą procesorów. Dlatego nawet krótkie i niewinnie wyglądające protokoły mogą być dość trudne do automatycznej weryfikacji, a w ręcznie tworzonych dowodach oczywiście subtelne błędy są regułą, a nie wyjątkiem. W ciągu ostatnich kilku lat opracowano potężne metody weryfikacji systemów współbieżnych względem formuł logiki temporalnej. Korzystając na przykład ze sprawdzania modelu, można konstruować wspomagane komputerowo dowody, że system spełnia formuły logiki

temporalnej, w tym właściwości bezpieczeństwa i żywotności. Może to stanowić pewną niespodziankę, biorąc pod uwagę nierozstrzygalność weryfikacji, o której była mowa w części 8. Co więcej, nawet jeśli ograniczymy się do systemów skończonych, np. zabraniając zmiennym przyjmowania wartości powyżej pewnej skończonej granicy, a poprzez ograniczenie a priori liczby elementów w systemie, odpowiednie problemy weryfikacyjne są nie do rozwiązania. Mimo to istnieją sposoby radzenia sobie z tymi problemami, które działają wyjątkowo dobrze w wielu przypadkach pojawiających się w praktyce. Rzeczywiście istnieje duża nadzieja na weryfikację, nawet w śliskiej sferze współbieżności.

Uczciwość i systemy czasu rzeczywistego

W tym miejscu warto wspomnieć o dwóch dodatkowych kwestiach, choć żadnej z nich nie będziemy szczegółowo omawiać. Pierwsza dotyczy globalnego założenia, które jest zwykle przyjmowane w przypadku bieżącej współbieżności. Zazwyczaj nie przyjmujemy żadnych założeń dotyczących względnej szybkości procesorów biorących udział w rozwiązywaniu problemów współbieżności, ale zakładamy, że wszystkie odpowiadają i robią postęp w skończonym, choć prawdopodobnie długim czasie. W naszych rozwiązaniach problemu wzajemnego wykluczania użyliśmy oczywiście tego założenia, ponieważ w przeciwnym razie P_1 mógłby osiągnąć klauzulę (1.3) i nigdy nie przejść do ustawienia Z na 1 i kontynuować. Założenie to jest czasami nazywane uczciwością, ponieważ jeśli myślisz o współbieżności jako implementowanej przez centralny procesor, czasami nazywany harmonogramem, który daje każdemu z procesorów współbieżnych kolejkę w wykonywaniu kilku instrukcji, to mówisz, że każdy procesor w końcu wykonuje postęp jest jak powiedzenie, że symulujący procesor jest sprawiedliwy wobec wszystkich, ostatecznie dając każdemu swoją kolej. Rozważ następujące protokoły dla dwóch procesorów P i Q.

protokół dla P:

(1) X ← 0;

(2) wykonaj następujące czynności raz za razem:

(2.1) jeśli Z = 0 to zatrzymaj się;

(2.2) X ← X + 1.

protokół dla Q:

(1) Z 0;

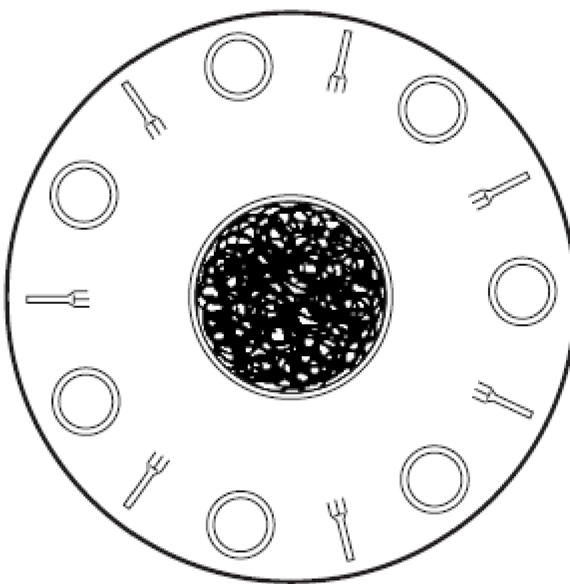
(2) zatrzymaj się.

Niech wartość początkowa Z wynosi 1. Jeśli nie przyjmiemy uczciwości, P może stale mieć prawo do kontynuacji, a słabe Q pozostaje na zawsze wykluczone z gry. To oczywiście prowadzi do nieskończonej pętli. Jedynym sposobem na zapewnienie zakończenia protokołów jest umożliwienie Q postępu. Z drugiej strony, przy założeniu słuszności, ten współbieżny algorytm kończy działanie, ustawiając X na jakąś nieznaną, prawdopodobnie bardzo dużą, nieujemną liczbę całkowitą. Sposób, w jaki to robi, jest następujący. Program planujący może dowolnie zaplanować kolejność P i Q, a w szczególności może pozwolić P na wiele razy, zanim wpuści Q po raz pierwszy. Musi jednak pozwolić Q na zmianę w pewnym momencie, zgodnie z założeniem uczciwości. W ten sposób P zwiększy X od 0 nieznaną liczbę razy, ale gdy Q dostanie, wykonanie zielonego światła zakończy się. Oczywiście ostateczna wartość X może być dowolną liczbą od zera w górę, ale proces w końcu się zakończy w wyniku sprawiedliwości. Oczywiście, że protokoły te mogą generować bardzo duże liczby, opiera się na założeniu, że niektóre procesory mogą być dowolnie wolniejsze niż inne. Czasami przydatne są silniejsze pojęcia uczciwości, zwłaszcza te, które ograniczają opóźnienie, które może wystąpić między akcjami. Możemy powiedzieć,

że Q ma wolniejszą reakcję niż P, ale nie jest gorsza niż dwa razy wolniej. Takie twierdzenia o uczciwości mają charakter ilościowy i wprowadzają do gry problemy z wyczuciem czasu. Dodatkowa komplikacja pojawia się w systemach, dla których ograniczenia czasowe mają kluczowe znaczenie, zwłaszcza w systemach czasu rzeczywistego. Są one zobowiązane do natychmiastowej reakcji na określone zdarzenia, a przynajmniej w niezauważalnym i znikomym czasie. Przykłady obejmują kontrolę lotu, naprowadzanie pocisków i systemy szybkiej komunikacji. Tutaj również może pomóc logika temporalna, wykorzystująca specjalny operator następnego kroku, który mówi o tym, jaka będzie prawdziwa jednorazowa jednostka czasu od chwili obecnej. Możliwe jest na przykład napisanie formuły TL, która stwierdza, że jeśli jeden dany fakt jest prawdziwy, inny stanie się prawdziwy, powiedzmy, 10 kroków później. To nadal nie rozwiązuje problemów programowania takich systemów, które zwykle są nie tylko krytyczne czasowo, ale także duże i złożone.

Problem jedzących filozofów

Jednym z najpopularniejszych przykładów ciągłej współbieżności, który ilustruje wiele zagadnień synchronizacji i współpracy, jest następujący. Mamy stół, wokół którego zasiada N-filozofów. Pośrodku duży talerz zawierający nieograniczoną ilość spaghetti. W połowie drogi między każdą parą sąsiednich filozofów znajduje się pojedyncze rozwidlenie



Teraz, skoro nikt, nawet filozof, nie może jeść spaghetti jednym widelcem, pojawia się problem. Pożądany cykl życia filozofa polega na wykonywaniu jego prywatnych czynności (np. myślenie, a następnie zapisywanie wyników do publikacji), odczuwanie głodu i próby jedzenia, jedzenia, a następnie powrót do prywatnych zajęć, ad infinitum. (Podobny, ale nieco trudniejszy do opanowania problem dotyczy chińskich filozofów, w których spaghetti i widelce zastępują ryż i pałeczki). Jak filozofowie powinni odprawiać swoje rytuały bez głodowania? Możemy poinstruować ich, aby po prostu podnieśli widelce po obu stronach, gdy są głodni, i jedzą, a po zakończeniu ponownie odkładają widelce. To rozwiązanie nie zadziała, ponieważ jeden lub oba widelce mogą zostać przejęte przez sąsiednich filozofów w tym czasie. Ponadto dwóch sąsiednich filozofów może próbować podnieść ten sam widelec w tym samym czasie. Używanie widelców, które są poza zasięgiem filozofa, jest zabronione. Tutaj jedzenie można uznać za część krytyczną, ponieważ dwóch sąsiednich filozofów nie

może jeść jednocześnie, a widelce są swego rodzaju kluczowymi zasobami, ponieważ nie mogą być używane przez dwóch filozofów jednocześnie. Ten problem jest również typowy dla wielu rzeczywistych sytuacji, takich jak na przykład komputerowe systemy operacyjne, w których wiele procesorów konkuruje o pewne współdzielone zasoby. Schemat połączeń takich systemów jest zwykle dość rzadki (nie wszystkie zasoby są dostępne dla każdego procesora), a liczba zasobów jest zbyt mała, aby wszyscy byli razem szczęśliwi. Jednym z rozwiązań tego problemu jest wprowadzenie do gry nowego gracza, portiera z jadalni. Filozofom instruuje się, aby wychodzili z pokoju, gdy nie są zainteresowani jedzeniem, i próbują wejść ponownie, gdy są. Portier ma obowiązek liczyć filozofów aktualnie przebywających w pokoju, ograniczając ich liczbę do N-1. Oznacza to, że jeśli w pokoju znajduje się wszystkich filozofów oprócz jednego, ten ostatni będzie czekał przy drzwiach, aż ktoś wyjdzie. Otóż, jeśli w danym momencie przy stole siedzi co najwyżej N - 1 filozofów, to jest co najmniej dwóch filozofów, którzy nie mają co najmniej jednego sąsiada, a zatem co najmniej jeden filozof może jeść. (Dlaczego?) W przypadku odpowiedniego sformalizowania rozwiązanie to może okazać się satysfakcjonujące. Wykorzystuje jednak dodatkową osobę (której miesięczna pensja będzie prawdopodobnie większa niż suma potrzebna na zakup dodatkowego zestawu widelców, przynajmniej za rozsądnie małe N). Czy problem kulinarnych filozofów można rozwiązać bez odźwiernego i bez uciekania się do pamięci wspólnej lub jej odpowiedników? Innymi słowy, czy istnieje w pełni rozproszone, w pełni symetryczne rozwiązanie problemu, które nie wymaga żadnych dodatkowych procesorów? W tym przypadku „w pełni rozproszony" oznacza, że nie ma centralnej pamięci współdzielonej, a protokoły mogą wykorzystywać tylko zmienne rozproszone (współdzielone, powiedzmy, tylko przez dwóch sąsiednich filozofów). „W pełni symetryczny" oznacza, że protokoły dla wszystkich filozofów są zasadniczo identyczne — nie pozwalamy różnym filozofom działać inaczej i nie pozwalamy im zaczynać od różnych wartości w swoich zmiennych. Te warunki mogą wydawać się zbyt restrykcyjne. Jeśli jednak filozofowie mają różne programy lub różne wartości początkowe, to tak, jakby powiedzieć, że wykorzystują swoją osobistą wiedzę i talenty w przyziemnym poszukiwaniu pożywienia. Chcielibyśmy to wszystko zachować na myślenie i publikowanie; jedzenie powinno być procedurą standardową i wspólną dla wszystkich. Odpowiedź na te pytania brzmi: nie. Do rozwiązania problemu potrzebujemy czegoś więcej, np. pamięci współdzielonej, bezpośredniej komunikacji między procesorami, specjalnego scentralizowane sterowanie procesorami lub wykorzystanie różnych informacji dla różnych filozofów.

Dlaczego? Argument jest naprawdę prosty. Pomyśl o prawidłowym rozwiązaniu jako takim, które gwarantuje wszystkie pożądane właściwości, nawet w przypadku najbardziej złośliwego programu planującego (to znaczy nawet w przypadku programu planującego, który stara się tak bardzo, jak tylko może, spowodować zakleszczenie lub zagłodzenie). Innymi słowy, aby pokazać, że nie ma w pełni rozłożonego, w pełni symetrycznego rozwiązania, wystarczy wykazać jedną konkretną kolejność, w jakiej filozofowie są zaplanowani do wykonywania działań kandydata na rozwiązanie, a następnie pokazanie, że coś musi pójść nie tak. Załóżmy zatem, że mamy jakieś takie kandydujące rozwiązanie. Załóżmy również, że ponumerowaliśmy filozofów 1, 2, . . . , N. (Sami filozofowie nie są świadomi nawet swoich liczb; nie mogą mieć żadnych prywatnych informacji, które mogą okazać się przydatne). Harmonogram, który przyjmujemy, jest następujący. Na każdym etapie każdy filozof w kolejności 1, 2, . . . , N może wykonać jedną podstawową akcję. Akcja może być zajętym testem oczekiwania, w takim przypadku testowanie i decydowanie, czy czekać, czy kontynuować, są traktowane razem jako jedno niepodzielne działanie podstawowe. Można wykazać, że na każdym etapie dokładnie to samo działanie wykonuje każdy procesor. Wynika to z faktów, że zarówno sytuacja wyjściowa a protokoły są w pełni symetryczne, nie ma procesorów innych niż sami filozofowie, a tabela i jej zawartość są symetrycznie cykliczne. W konsekwencji sytuacja na końcu każdego etapu będzie nadal w pełni symetryczna; to znaczy, że wartości zmiennych będą takie same dla wszystkich filozofów, podobnie jak ich lokalizacje

w protokołach. Czy jakikolwiek filozof mógł jeść na jednym lub kilku etapach? Nie, ponieważ etapy obejmują tylko podstawowe czynności, a pod koniec etapu można wykryć fakt, że filozof je lub jadł. Ale niemożliwe jest, aby wszyscy filozofowie jedli na raz, a proces jedzenia dwóch filozofów, jeden po drugim, nie może być przeprowadzony w jednym etapie. Z niezbędnej symetrii wynika zatem, że na końcu każdego etapu żaden filozof nie będzie jadł. Dlatego nikt nigdy nie będzie. W części 11 zobaczymy, że problem filozofów jedzenia można rozwiązać w sposób w pełni rozproszony i w pełni symetryczny, ale z bardziej liberalnym pojęciem poprawności.

Semafory

Istnieje wiele języków programowania, które w taki czy inny sposób obsługują współbieżność, w tym języki zorientowane obiektowo i formalizmy wizualne do rozwoju systemu, z rodzaju omówionego później. Nie będziemy tutaj opisywać żadnego z tych języków, ale pokrótce rozważymy jedną z głównych konstrukcji wymyślonych specjalnie do radzenia sobie ze współbieżnością. Stanowi on podstawę części sposobu implementacji współbieżności w niektórych z tych języków i może być używany jawnie w innych. Wiemy już, że pamięć współdzielona i komunikacja bezpośrednia reprezentują dwa główne podejścia do opisywania współpracy, jaka ma zachodzić między równolegle realizowanymi procesami. Jeśli używa się tego pierwszego, musi istnieć mechanizm rozwiązywania konfliktów pisarskich, które wywołuje pamięć współdzielona. Jednym z najpopularniejszych z nich jest semafor.

Semafor to specjalny rodzaj elementu programistycznego, który wykorzystuje dwie operacje do kontrolowania użycia sekcji krytycznych (takich jak te, które wiążą się z zapisem w części pamięci współdzielonej). Próba wejścia do takiej sekcji jest reprezentowana przez operację żądania, a zwolnienie oznacza wyjście. Semafor S może właściwie być postrzegany jako zmienna o wartości całkowitej. Wykonywanie żądania(S) to niepodzielna akcja, która próbuje zmniejszyć S o 1, robiąc to bez przerwy, jeśli jego wartość jest dodatnia i czeka, aż stanie się dodatnia w przeciwnym razie. Skutkiem zwolnienia (S) jest po prostu zwiększenie S o 1. Ważne jest to, że z samej definicji semafor S poddaje się tylko jednemu żądaniu lub operacji zwolnienia na raz. W typowym użyciu semaforów w celu osiągnięcia wzajemnego wykluczenia, S otrzymuje wartość początkową 1, a sekcja krytyczna każdego procesora jest zawarta w następujący sposób:

. . .

upraszanie (S);

przeprowadzić sekcję krytyczną;

zwolnienie(S);

. . .

Powoduje to dopuszczenie tylko jednego procesora na raz do jego krytycznej sekcji. Pierwszy, który próbuje wejść, udaje się zmniejszyć S do 0 i wchodzi. Pozostali muszą poczekać, aż pierwszy z nich wyjdzie, zwiększając w tym procesie S do 1. Semafor, który zaczyna się od 1, a zatem przyjmuje tylko wartości 1 i 0, nazywany jest semaforem binarnym. Prostym sposobem użycia semaforów dla sekcji krytycznych, które mogą obsłużyć do K procesorów na raz, jest użycie niebinarnego semafora o początkowej wartości K. (Dlaczego ma to pożądany efekt?) Na przykład w problemie dotyczącym jadalni , portier może być modelowany przez semafor o wartości początkowej N - 1, kontrolujący sekcję krytyczną, która obejmuje czynności związane z próbą jedzenia, jedzeniem i opuszczeniem pokoju. Użycie każdego widelca może być modelowane przez semafor binarny. Warto zauważyć, że ta prosta definicja semaforów nie zakłada, że wszystkie procesory oczekujące na zablokowane operacje żądania

ostatecznie otrzymają prawo do wejścia, gdy S staje się niezerowe. Jest całkiem możliwe, że złośliwa implementacja semaforów zawsze daje pierwszeństwo najnowszemu procesorowi, blokując inne na zawsze. To jeden z przypadków, w których pewna uczciwość założenia wydaje się konieczne, dzięki czemu, powiedzmy, każdy oczekujący procesor ma zagwarantowany ewentualny postęp. Semafory można zatem opisać jako bardzo prosty typ danych, którego operacje (inkrementacja, test-i-dekrementacja) są chronione przed konfliktami zapisu przez wbudowany mechanizm wzajemnego wykluczania. Wzajemne wykluczanie dla bardziej skomplikowanych typów danych, które wykorzystują wiele operacji, można osiągnąć przez otoczenie każdego wystąpienia a operacja pisania z odpowiednimi operacjami na semaforach. Jednak w pewnym sensie semafory są jak wypowiedzi goto; zbyt wiele operacji żądań i wydawania rozrzuconych po całym długim programie może stać się niejasne i podatne na błędy. Semafory mogą być używane do rozwiązywania standardowych rodzajów problemów w programowaniu współbieżności, ale stanowią niestrukturalną konstrukcję programistyczną.

Badania nad równoległością i współbieżnością

Jeśli w poprzednich rozdziałach stwierdziliśmy, że prowadzone są intensywne badania nad wieloma omawianymi tematami, to jest to prawdziwsze niż kiedykolwiek w dziedzinie paralelizmu i współbieżności. Badacze starają się pogodzić praktycznie ze wszystkimi aspektami współpracy algorytmicznej i nie będzie przesadą stwierdzenie, że badania większości informatyków mają jakiś związek z poruszanymi w tej części tematami. Prowadzone są intensywne badania nad znajdowaniem szybkich algorytmów równoległych dla różnych problemów algorytmicznych, a rozwiązania wykorzystują szerokie spektrum wyrafinowanych struktur danych i mechanizmów współbieżności. Wiele problemów (takich jak obliczenia gcd) oparło się próbom użytecznego wykorzystania wszelkiego rodzaju równoległości. Obwody Boole'a i sieci skurczowe są również przedmiotem wielu aktualnych badań i odkryto interesujące powiązania między tymi podejściami a konwencjonalną algorytmiką sekwencyjną. Abstrakcyjna teoria złożoności paralelizmu stawia wiele ważnych i najwyraźniej bardzo trudnych, nierozwiązanych pytań dotyczących klas takich jak NC i PSPACE, z których niektóre zostały opisane wcześniej. Podczas gdy, jak widzieliśmy, sekwencyjność stwarza już wiele nierozwiązanych problemów, równoległość niewątpliwie rodzi o wiele więcej. W rzeczywistości wydaje się jasne, że rozumiemy podstawy algorytmów sekwencyjnych znacznie lepiej niż algorytmów równoległych, a przed nami długa i trudna droga. Inne tematy aktualnych badań obejmują równoległe projektowanie komputerów, techniki sprawdzania i analizy w celu wnioskowania o współbieżnych procesach oraz tworzenie użytecznych i wydajnych języków programowania współbieżnego. Jak ilustruje historia budowy domu na początku rozdziału, współbieżność jest faktem i im lepiej ją rozumiemy, tym bardziej możemy ją wykorzystać na naszą korzyść. W pewnym sensie najnowsze postępy naukowe i technologiczne w zakresie współbieżności wyprzedzają się. Wiele z najlepszych znanych algorytmów równoległych nie może zostać zaimplementowanych, ponieważ istniejące komputery równoległe są w jakiś sposób nieodpowiednie. Z drugiej strony wciąż nie wiemy wystarczająco dużo o projektowaniu współbieżnych programów i systemów, aby w pełni wykorzystać funkcje oferowane przez te same komputery. Prace są jednak kontynuowane i ciągle osiągane są znaczące wyniki, chociaż głębokie kwestie związane z prawdziwą złożonością paralelizmu pozostają nieuchwytne. Przejdziemy teraz do dwóch nowszych podejść do równoległości, które atakują go z zupełnie innych punktów widzenia.

Obliczenia kwantowe

Więc co to za modne nowe rzeczy do obliczeń kwantowych? Cóż, jest to głęboki i skomplikowany temat, oparty na złożonym materiale matematycznym i fizycznym, a przez to bardzo trudny do opisania w sposób wyjaśniający tej książki. Obliczenia kwantowe opierają się na mechanice kwantowej, niezwykłym temacie we współczesnej fizyce, który niestety jest śliski i trudny do uchwycenia i często

jest sprzeczny z intuicją. Naiwna próba wykorzystania ziemskiego zdrowego rozsądku, aby to zrozumieć, może łatwo stać się przeszkodą w zrozumieniu, a nie pomocą. Kolejne sekcje będą więc traktować ten temat wyjątkowo powierzchownie, nawet stosując standardy tej techniczności, unikając książki. Przepraszamy za to. Notatki bibliograficzne zawierają jednak kilka wskazówek do badań w literaturze dla bardziej dociekliwych, biegłych matematycznie czytelników. Po jaśniejszej stronie jest szansa - że komputery kwantowe mogą przynieść dobre wieści. Jak, dlaczego i kiedy są pytania, którymi postaramy się odpowiedzieć, bardzo krótko, w dalszej części pracy. Jedną z głównych zalet fizyki kwantowej jest jej zdolność do zrozumienia pewnych zjawisk eksperymentalnych na poziomie cząstek, których fizyka klasyczna wydawała się nie być w stanie. Dwie z głównych ciekawostek świata kwantowego, o których mówi się bardzo nieformalnie, to fakt, że cząstki nie można już uważać za znajdujące się w jednym miejscu w przestrzeni w określonym czasie oraz że jej sytuacja (w tym lokalizacje) może się zmienić w wyniku po prostu to obserwując. Pierwsza z nich wydaje się dobrą wiadomością dla komputerów: czy nie bylibyśmy w stanie wykorzystać właściwości przebywania w wielu miejscach razem, aby przeprowadzić masową równoległość obliczeń? Druga jednak wydaje się złą wiadomością: próba „zobaczenia” lub „dotknięcia” wartości podczas obliczeń, powiedzmy, aby przeprowadzić porównanie lub aktualizację, może w nieprzewidywalny sposób zmienić tę wartość! Obliczenia kwantowe to bardzo nowy pomysł. Wczesne prace były motywowane twierdzeniem, że gdyby można było zbudować komputer działający zgodnie z prawami fizyki kwantowej, a nie klasycznej, można by uzyskać wykładnicze przyspieszenie niektórych obliczeń. Komputer kwantowy, podobnie jak klasyczny, ma być oparty na jakimś elemencie skończonym, analogicznym do klasycznego bitu dwustanowego. Kwantowy analog bitu, zwany kubitem i wymawiany jako „bit kolejki”, można wyobrazić sobie fizycznie na wiele sposobów: zgodnie z kierunkiem polaryzacji fotonu (pozioma lub pionowa), przez spin jądrowy (specjalny dwuwartościowy kwant obserwowalne) lub przez poziom energii atomu (ziemnego lub wzbudzonego). Dwa tak zwane stany bazowe kubitu, analogiczne do 0 i 1 zwykłego bitu, są oznaczone odpowiednio przez $|0>$ i $|1>$. To, czego nie mamy w systemie kwantowym, to proste, deterministyczne pojęcie, że kubit znajduje się w takim czy innym stanie bazowym. Jego pojęcie bycia lub nie bycia jest raczej nieokreślone: wszystko, co możemy powiedzieć o statusie kubitu, to to, że znajduje się on w obu stanach jednocześnie, z których każdy ma pewne „prawdopodobieństwo”. Ale jakby celowo uczynić rzeczy jeszcze mniej zrozumiałymi dla śmiertelników, nie są to zwykłe, dodatnie prawdopodobieństwa, jak bycie w stanie $|0>$ z prawdopodobieństwem 1/4 i w $|1>$ z prawdopodobieństwem 3/4 . Te „prawdopodobieństwa” mogą być ujemne, a nawet urojone (tj. liczby zespolone, które zawierają pierwiastki kwadratowe z liczb ujemnych), a wynikowy stan kombinacji nazywany jest superpozycją. Gdy już „przyjrzymy się” kubitowi, czyli dokonamy pomiaru, nagle decyduje, gdzie być, widzimy go w jednym lub drugim stanie bazowym, prawdopodobieństwa znikają, a superpozycja zostaje zapomniana. Ten rodzaj „wymuszonej dyskrecji” prowadzi do przymiotnika „kwant”. To tyle, jeśli chodzi o pojedynczy kubit. Co dzieje się z wieloma kubitami razem, obok siebie, których potrzebujemy jako podstawy do prawdziwych obliczeń kwantowych? Jak połączone są stany kilku kubitów, aby uzyskać złożony stan całego urządzenia obliczeniowego? W klasycznym przypadku każdy zbiór N bitów, z których każdy może być w dwóch stanach 0 lub 1, powoduje powstanie 2^N stanów złożonych. W kwantowym świecie kubitów również zaczynamy od 2^N stanów złożonych zbudowanych ze stanów bazowych N kubitów (na przykład w przypadku dwóch kubitów cztery stany złożone są oznaczone $|00>$, $|01>$, $|10>$ i $|11>$). Następnie stosujemy do nich złożone kombinacje, tak jak w przypadku pojedynczego kubitu. Jednak w tym przypadku sposób zdefiniowania kombinacji powoduje dodatkowy kluczowy skręt zwany odpowiednio splątaniem: niektóre stany złożone są czystymi kompozytami, które można uzyskać – za pomocą operacji zwanej „iloczynem tensorowym” – ze stanów oryginalne kubity, ale niektóre nie; są uwikłane. Splątane kubity, termin, który pojawia się w precyzyjnym matematycznym ujęciu, stanowią z natury nierozłączny „misz-masz” oryginalnych kubitów. Mają dziwną właściwość natychmiastowej

komunikacji: obserwowanie jednego, a tym samym naprawianie jego stanu, powoduje, że drugi jednocześnie blokuje się w podwójnym stanie, bez względu na to, jak daleko się od siebie znajdują. Splątanie okazuje się być podstawowym i niezbędnym pojęciem w obliczeniach kwantowych, ale niestety dalsze omówienie ich technicznych aspektów i sposobu, w jaki jest ono wykorzystywane w samych obliczeniach, wykracza poza zakres tej książki.

Algorytmy kwantowe

Co ludzie byli w stanie zrobić z obliczeniami kwantowymi? Z góry należy podać kilka faktów. Po pierwsze, pełne obliczenia kwantowe ogólnego przeznaczenia obejmują obliczenia klasyczne. Oznacza to, że jeśli i kiedy zostanie zbudowany, komputer kwantowy będzie w stanie emulować klasyczne obliczenia bez znaczącej straty czasu. Po drugie, choć pozornie słabszy, klasyczny komputer może symulować dowolne obliczenia kwantowe, ale może to pociągać za sobą wykładniczą stratę czasu. Fakt, że taka symulacja jest możliwa, oznacza, że obliczenia kwantowe nie mogą zniszczyć tezy Churcha/Turinga: obliczalność pozostaje nienaruszona również w świecie obliczeń kwantowych. Jeśli i kiedy zostaną zbudowane rzeczywiste komputery kwantowe, nie będą w stanie rozwiązać problemów, których bez nich nie da się rozwiązać. To powiedziawszy, głównym pytaniem jest, czy wykładnicza strata czasu w drugim stwierdzeniu jest rzeczywiście nie do pokonania. Podobnie jak w przypadku równoległości, pytamy, czy w świecie kwantowym istnieją nierozwiązywalne problemy, które stają się wykonalne. To znaczy, czy istnieje problem z dolną granicą wykładniczą czasu wykładniczego w klasycznych modelach obliczeń, które mają algorytm kwantowy w czasie wielomianowym? I tutaj, jeśli używamy QP do oznaczenia quantum-PTIME, mamy:

PTIME ⊆ QP ⊆ PSPACE (= równolegle-PTIME)

Zatem rozsądny, tj. wielomianowy, czas kwantowy leży w tym samym miejscu co NP, tj. między rozsądnym czasem deterministycznym a rozsądną przestrzenią pamięci. Niestety, tak jak poprzednio, nie wiemy, czy któraś z tych inkluzji jest ścisła. Pomijając złożoność obliczeniową i niezależnie od technologicznego problemu faktycznego zbudowania komputera kwantowego, dokonano już kilku niezwykle ekscytujących postępów w algorytmice kwantowej. Oto niektóre z najważniejszych wydarzeń. Rzeczywiście osiągnięto równoległość kwantową, w której superpozycja wejść jest wykorzystywana do wytworzenia superpozycji wyjść. Co ciekawe, chociaż wydaje się, że rzeczywiście wykonuje się wiele rzeczy równolegle, wyników nie można naiwnie oddzielić i odczytać z ich superpozycji; każda próba odczytu lub pomiaru da tylko jeden wynik, a reszta zostanie po prostu utracona. Potrzebny jest algorytm, aby sprytnie obliczyć wspólne właściwości wspólne dla wszystkich wyników i poradzić sobie z nimi. Przykłady mogą obejmować pewne zagregowane wartości arytmetyczne wyników liczbowych lub „i" i „lub" logicznych wyników tak/nie . Później odkryto dość zaskakujący algorytm kwantowy do wyszukiwania na nieuporządkowanej liście, powiedzmy w dużej bazie danych. Zamiast około N operacji, element można znaleźć tylko z √N operacjami (pierwiastek kwadratowy z N). Jest to sprzeczne z intuicją, prawie paradoksalne, ponieważ wydaje się konieczne przynajmniej spojrzenie na wszystkie wejścia N, aby dowiedzieć się, czy rzeczywiście tam jest to, czego szukasz.

Jednak wielką niespodzianką, a nawet szczytem dotychczasowych algorytmów kwantowych, jest algorytm faktoryzacji Shora. Wielokrotnie wspominaliśmy o faktoryzacji w książce, a jego znaczenie jako centralnego problemu algorytmicznego jest niepodważalne. Jak widzieliśmy, faktoring nie okazał się jeszcze wykonalny w zwykłym sensie – nie wiadomo, czy jest w PTIME (co, jak teraz wiemy, nie dotyczy testowania pierwszości) i sam fakt, że się pojawia trudność obliczeniowa odgrywa kluczową rolę w kryptografii, jak zobaczymy w części 12. Do tego stopnia, że znaczna część ścian, które podtrzymują współczesną kryptografię, zawaliłaby się, gdyby dostępny był wydajny algorytm

faktoryzacji . Na tym tle należy spojrzeć na znaczenie tej pracy, która dostarcza wielomianowego algorytmu kwantowego dla problemu. Aby docenić subtelność faktoryzacji kwantowej, rozważ naiwny algorytm, który próbuje znaleźć czynniki liczby N metodą prób i błędów, przechodząc przez wszystkie pary potencjalnych czynników i mnożąc je, aby sprawdzić, czy ich iloczyn jest dokładnie N. Dlaczego nie możemy to zrobić za pomocą równoległości kwantowej na wielką skalę? Moglibyśmy użyć zmiennych kwantowych do zachowania superpozycji wszystkich czynników kandydujących (powiedzmy, wszystkich liczb od 0 do N − 1), a następnie obliczyć równolegle, w najlepszym duchu kwantowym, wszystkie iloczyny wszystkich możliwych par tych liczb. mógł następnie spróbować sprawdzić, czy istnieje para, która wykonała zadanie. Niestety, to by nie zadziałało, ponieważ przyjrzenie się – to znaczy wykonanie pomiaru – tego ogromnego nałożonego wyniku nie powiedziałoby wiele. Może się zdarzyć, że trafimy na faktoryzację, ale możemy też trafić na dowolny inny spośród wielu produktów, które różnią się od N. Jak już wspomnieliśmy, gdy zmierzysz, to właśnie zobaczysz, a reszta jest stracona. Tak więc samo mieszanie wielu informacji nie wystarczy. Okazuje się, że trzeba wszystko tak zaaranżować, żeby była ingerencja. Jest to pojęcie kwantowe, zgodnie z którym możliwe rozwiązania „walczą" ze sobą o dominację w subtelny sposób. Te, które okażą się nie dobrymi rozwiązaniami (w naszym przypadku pary liczb, których iloczyn nie jest N) będą kolidować destrukcyjnie w superpozycji, a te, które są dobrymi rozwiązaniami (ich iloczynem jest N) będą przeszkadzać konstruktywnie. Wyniki tej walki pokażą się jako zmienne amplitudy na wyjściu, tak więc pomiar superpozycji wyjścia da dobrym rozwiązaniom znacznie lepszą szansę na pojawienie się. Należy zauważyć, że to liczby ujemne w definicji superpozycji umożliwiają tego rodzaju interferencję w algorytmie kwantowym. Łatwiej to powiedzieć niż zrobić, i właśnie tutaj matematyka obliczeń kwantowych staje się skomplikowana i wykracza poza zakres i poziom naszej prezentacji. Ale możemy powiedzieć, że osiągnięto właściwy rodzaj splątania dla faktoringu. Sam algorytm jest dość niezwykły, zarówno pod względem techniki, jak i, jak zobaczymy później, w jego rozgałęzieniach. Jego wydajność czasowa jest z grubsza sześcienna, czyli niewiele większa niż M^3, gdzie M jest liczbą cyfr w numerze wejściowym N. Dla bardziej zainteresowanego technicznie czytelnika algorytm wykorzystuje wydajną metodę kwantową do obliczania kolejności liczby Y modulo N, to znaczy znaleźć najmniejszą liczbę całkowitą a taką, że $Y^a = 1$ (mod N). Wiadomo, że jest to wystarczające, aby umożliwić szybkie faktoring, a resztę pracy wykonuje się przy użyciu konwencjonalnych algorytmów. Algorytm ten nie zamienił jeszcze trudnego do udowodnienia problemu w wykonalny z dwóch powodów, z których jeden wielokrotnie wspominaliśmy, a jeden o którym wspomnieliśmy, ale wkrótce zajmiemy się bardziej szczegółowo. Po pierwsze, faktoring nie jest trudny do wyegzekwowania; po prostu nie byliśmy w stanie znaleźć dla niego algorytmu wielomianowego. Przypuszcza się, że jest to trudne, ale nie jesteśmy pewni. Po drugie, praktyczne trudności związane ze zbudowaniem komputera kwantowego są naprawdę ogromne.

Czy może istnieć komputer kwantowy?

Omawiając wcześniej równoległość, zauważyliśmy, że istnieje pewna niezgodność między istniejącymi algorytmami równoległymi a równoległymi komputerami, które zostały zbudowane do ich obsługi. Aby mogły zostać skutecznie zaimplementowane, wiele znanych algorytmów wymaga funkcji sprzętowych, które nie są jeszcze dostępne, i, z drugiej strony, teoria algorytmów równoległych musi jeszcze dogonić to, co jest w stanie zrobić dostępny sprzęt. W dziedzinie obliczeń kwantowych sytuacja jest mniej symetryczna. Mamy do dyspozycji kilka naprawdę fajnych algorytmów kwantowych, ale żadnych maszyn, na których można by je uruchamiać. Czemu? Znowu sprawa ta kręci się wokół głębokich szczegółów technicznych, ale tym razem barierą uniemożliwiającą szczegółowe przedstawienie nie jest matematyka, ale fizyka. Tak więc, ponownie, przedstawimy tylko bardzo krótką relację, a zainteresowany czytelnik będzie musiał szukać więcej informacji gdzie indziej. Jaki jest problem? Dlaczego nie możemy zwiększyć skali? Pomimo faktu, że same algorytmy kwantowe, a w szczególności ten faktoring, są zaprojektowane do pracy zgodnie z rygorystycznymi i powszechnie akceptowanymi

zasadami fizyki kwantowej, istnieją poważne problemy techniczne związane z samą konstrukcją komputera kwantowego. Po pierwsze, fizykom eksperymentalnym nie udało się połączyć nawet niewielkiej liczby kubitów (powiedzmy 10) i kontrolować ich w jakiś rozsądny sposób. Trudności wydają się wykraczać poza dzisiejsze techniki laboratoryjne. Szczególnie niepokojącym problemem jest dekoherencja: nawet jeśli udałoby się zebrać dużą liczbę kubitów i sprawić, by same zachowywały się ładnie, rzeczy, które znajdują się w pobliżu układu kwantowego, mają natarczywy zwyczaj wpływania na niego. Zachowanie kwantowe wszystkiego, co otacza komputer kwantowy - obudowa, ściany, ludzie, klawiatura, cokolwiek! - potrafi zepsuć delikatną konfigurację konstruktywnej i destrukcyjnej ingerencji w obliczeniach kwantowych. Nawet pojedynczy niegrzeczny elektron może wpłynąć na wzór interferencji, który jest tak istotny dla prawidłowego wykonania algorytmu, poprzez splątanie z kubitami biorącymi udział w tym wykonaniu, w wyniku czego pożądana superpozycja może się nie powieść. W związku z tym komputer musi być bezwzględnie odizolowany od otoczenia. Ale musi również odczytywać dane wejściowe i generować dane wyjściowe, a jego proces obliczeniowy może być kontrolowany przez niektóre elementy zewnętrzne. W jakiś sposób te sprzeczne wymagania trzeba pogodzić. Jakich rozmiarów naprawdę potrzebujemy? Niektóre protokoły kodowania kwantowego na małą skalę wymagają tylko około 10 kubitów, a nawet algorytm faktoryzacji kwantowej potrzebuje tylko kilku tysięcy kubitów, aby można go było zastosować w rzeczywistych sytuacjach. Ale ponieważ fizyka eksperymentalna radzi sobie obecnie tylko z siedmioma kubitami, a nawet to jest niezwykle trudne, wiele osób jest pesymistami. Prawdziwy przełom nie jest spodziewany w najbliższym czasie. Z drugiej strony, podekscytowanie związane z tym tematem już powoduje lawinę pomysłów i propozycji, którym towarzyszą złożone eksperymenty laboratoryjne, tak że z biegiem czasu z pewnością zobaczymy interesujące postępy. Podsumowując, wielomianowy algorytm faktoringu kwantowego Shora stanowi duży postęp w badaniach obliczeniowych pod każdą miarą. Jednak w tej chwili musi zostać zdegradowany do statusu półek i prawdopodobnie pozostanie takim przez jakiś czas. Nieusuwalność nie została jeszcze pokonana.

Obliczenia molekularne

Aby zakończyć naszą dyskusję na temat modeli obliczeniowych mających na celu złagodzenie niektórych złych wiadomości, wspominamy jeszcze jedną: obliczenia molekularne, czasami nazywane obliczeniami DNA. Główne podejście tutaj polega na tym, aby obliczenia odbywały się zasadniczo same, w starannie skomponowanej „zupie” cząsteczek, które bawią się ze sobą, dzieląc, łącząc i scalając. W ten sposób otrzymujesz miliardy lub biliony molekuł do rozwiązania trudnego problemu za pomocą brutalnej siły, sprytnie konfigurując rzeczy, aby zwycięskie przypadki można później wyizolować i zidentyfikować. W eksperymencie z 1994 roku stworzono molekuły w celu rozwiązania małego przykładu problemu ścieżki Hamiltona, który, jak wyjaśniono w rozdziale 7, jest w rzeczywistości rodzajem jednostki długości wersji problemu komiwojażera. Później inne problemy — w zasadzie wszystkie problemy w NP - okazały się podatne na podobne techniki. To, że natura może być dostrojona do rozwiązywania rzeczywistych problemów algorytmicznych, zasadniczo sama i w skali molekularnej, jest raczej zdumiewające. Podczas gdy oryginalny eksperyment dla instancji z siedmiu miast trwał kilka dni w laboratorium, problem został później rozwiązany przez innych w mniej brutalny sposób i dla znacznie większych instancji (50–60 miast). Poświęcenie laboratoriów biologii molekularnej do tego rodzaju prac może skutkować znacznym przyspieszeniem procesu i rzeczywiście trwa wiele prac, aby spróbować zwiększyć skalę technik. Z purystycznego punktu widzenia rzeczy przypominają konwencjonalne algorytmy równoległe: chociaż w zasadzie złożoność czasowa takich algorytmów molekularnych jest wielomianowa ze względu na wysoki stopień paralelizmu, który zachodzi w zupie molekularnej, liczba cząsteczek biorących udział w procesie rośnie wykładniczo. Ale z drugiej strony, jedną z głównych zalet używania DNA jest jego niesamowita gęstość informacji. Niektóre wyniki pokazują, że obliczenia DNA mogą zużywać miliard razy mniej energii niż komputer

elektroniczny wykonujący te same czynności i mogą przechowywać dane w bilionach razy mniejszej przestrzeni. W każdym razie informatyka molekularna jest zdecydowanie kolejnym ekscytującym obszarem badań, przyciągającym wyobraźnię i energię wielu utalentowanych informatyków i biologów. W przyszłości czeka nas wiele ekscytujących prac w tej dziedzinie, a niektóre specyficzne, trudne problemy mogą stać się wykonalne przy rozsądnych nakładach. Musimy jednak pamiętać, że zdecydowanie nie może wyeliminować nieobliczalności, ani nie oczekuje się, że zlikwiduje fatalne skutki nierozwiązywalności.

Algorytmy probabilistyczne

W poprzedniej części podjęliśmy kroki, które wyprowadziły nas poza standardowe ramy problemów algorytmicznych i ich rozwiązań. Zezwoliliśmy na algorytmy równoległe, takie, które wykorzystują kilka małych Runaroundów zamiast jednego. Odejście to wymagało niewielkiego uzasadnienia, ponieważ można łatwo zauważyć, że poprawia wydajność. Dyskutowaliśmy również o wykorzystaniu kuszącej mocy mechaniki kwantowej lub sił rządzących interakcjami molekularnymi w naszych poszukiwaniach masywnej równoległości. W sumie skoncentrowaliśmy się na robieniu wielu rzeczy na raz – nie tylko jednej. W tym rozdziale zrobimy bardziej radykalny krok, wyrzekając się jednego z najświętszych wymagań w całej algorytmice, a mianowicie tego, że rozwiązanie problemu algorytmicznego musi rozwiązać ten problem poprawnie, dla wszystkich możliwych danych wejściowych. Nie możemy całkowicie zrezygnować z potrzeby poprawności, ponieważ jeśli to zrobimy, każdy algorytm „rozwiąże" każdy problem. Nie możemy też sobie pozwolić na polecanie ludziom rozwiązywania problemów algorytmicznych za pomocą algorytmów, które mają nadzieję zadziałać, ale których działanie mogą tylko obserwować, a nie analizować. Interesują nas algorytmy, które mogą nie zawsze być poprawne, ale których ewentualną niepoprawność można bezpiecznie zignorować. Ponadto nalegamy, aby fakt ten był uzasadniony rygorystycznymi podstawami matematycznymi. Jeśli założymy, że obracanie lufy rewolweru jest naprawdę przypadkowym sposobem wyboru jednej z sześciu pozycji pocisku, to niektórzy mogą uznać, że szanse na śmierć w jednej próbie rosyjskiej ruletki są mało prawdopodobne. Większość ludzi nie. Załóżmy teraz, że rewolwer ma 2200 pozycji pocisków lub (równoważnie), że spust w zwykłym rewolwerze sześciopociskowym jest faktycznie pociągany tylko wtedy, gdy pojedynczy pocisk zawsze trafiał do pozycji strzeleckiej w 77 kolejnych obrotach. W takim przypadku szanse na śmierć w jednym (77 spinów) zagraniu są o wiele rzędów wielkości mniejsze niż szanse na osiągnięcie tego samego efektu przez wypicie szklanki wody, dojazd do pracy lub wzięcie głębokiego oddechu. powietrze. Oczywiście w takim przypadku szanse można spokojnie zignorować; prawdopodobieństwo katastrofy jest niewyobrażalnie znikome. Ta część dotyczy jednego ze sposobów wykorzystania teorii prawdopodobieństwa w projektowaniu algorytmicznym. Rozważymy algorytmy, które w trakcie swojego działania mogą rzucać uczciwymi monetami, dając naprawdę losowe wyniki. Konsekwencje dodania tego nowego obiektu okazują się dość zaskakujące. Zamiast stanowić krok wstecz, prowadzący do algorytmów dających nieprzewidywalne wyniki, nowa zdolność okaże się niezwykle użyteczna i zdolna do uzyskania szybkich probabilistycznych rozwiązań problemów, które mają tylko bardzo nieefektywne rozwiązania konwencjonalne. Ceną za to zapłaconą jest możliwość pomyłki, ale podobnie jak w 77-rundowej wersji rosyjskiej ruletki, tę możliwość można spokojnie zignorować. Co ciekawe, w Części 12 zobaczymy, że probabilizm lub randomizacja w algorytmice jest najkorzystniejsza, gdy używa się go razem z negatywnymi wynikami dotyczącymi problemów, dla których nie są znane żadne dobre rozwiązania, nawet te probabilistyczne.

Więcej o Jadalni Filozofów

W Części 10 zapoznaliśmy się z filozofami kulinarnymi i zobaczyliśmy, że problem nie daje rozwiązań pozbawionych impasu, jeśli nalegamy na całkowitą symetrię i nie używamy scentralizowanych zmiennych. Reguły określały, że kandydujące rozwiązanie musi działać poprawnie nawet w przypadku najbardziej złośliwego programu planującego; na przykład nawet w przypadku, gdy wszyscy stają się głodni i starają się podnieść widły dokładnie w tym samym czasie. Teraz pokazujemy, że problem można rozwiązać, jeśli pozwolimy filozofom rzucać monetami. Podstawową ideą jest wykorzystanie rzucania monetami do złamania symetrii na dłuższą metę. Konkretnie, rozważ następujące potencjalne rozwiązanie:

protokół dla każdego filozofa:

(1) wykonaj następujące czynności raz za razem:

(1.1) prowadzić prywatną działalność aż do głodu;

(1.2) rzucać monetą, aby wybrać losowo kierunek, w lewo lub w prawo;

(1.3) poczekaj, aż widelce leżące w wybranym kierunku będą dostępne, a następnie podnieś je;

(1.4) jeśli inny widelec nie jest dostępny, wykonaj następujące czynności:

(1.4.1) odłóż widelce, które zostały podniesione;

(1.4.2) przejdź do (1.2);

(1.5) w przeciwnym razie (tj. dostępny jest inny widelec) podnieść inny widłelec;

(1.6) sekcja krytyczna: jedz do syta;

(1.7) odłóż oba widelce (i wróć do (1.1)).

Można wykazać, że to rozwiązanie jest wolne od zakleszczeń z prawdopodobieństwem 1. Co to oznacza? Cóż, oznacza to, że w nieskończonym czasie szanse wystąpienia impasu są zerowe. Nie tylko mały lub znikomy, ale zero! To stwierdzenie wymaga dalszych wyjaśnień. Możliwe, że system filozofów znajdzie się w martwym punkcie, ale szanse, że nastąpi impas, są zerowe w stosunku do szans, że tak się nie stanie. Rozważmy program planujący, który powoduje, że wszyscy filozofowie stają się jednocześnie głodni. Powiedzmy, że wszyscy filozofowie znajdują się razem w punkcie (1.2). Jednym ze sposobów na zakleszczenie systemu jest to, że wszyscy filozofowie jednocześnie wybierają ten sam kierunek w (1.2), powiedzmy w prawo, podnoszą prawe rozwidlenie, odkrywają, że drugie rozwidlenie nie jest dostępne (w (1.4)), włóż w dół prawego widelca (w (1.4.1)), wróć do (1.2) i ponownie, wszyscy wybierają ten sam kierunek, być może tym razem w lewo, ponownie podnieś odpowiednie widły, a następnie odłóż je, a następnie wybierz znowu w tym samym kierunku i tak dalej, w nieskończoność. To wyraźnie sytuacja impasowa, ponieważ żaden filozof nie może jeść. To, że wszyscy filozofowie są doskonale zsynchronizowani, osiągając tę samą instrukcję w tym samym czasie, jest przebiegłą pracą tego specjalnego planisty, a planowanie nie jest czymś, co jest wybierane przypadkowo; nasze rozwiązania muszą działać nawet przeciwko najgorszym z harmonogramów. Rzucanie monetą może być używane tylko w ramach protokołów, aby wpłynąć na zachowanie poszczególnych procesorów, ale nie w celu wpływania na kolejność lub czas, w jakim mają wykonać swoje instrukcje. Wynika z tego, że złośliwe planowanie nie może być powodem, dla którego konkretny scenariusz, taki jak ten, ma zerową szansę na wystąpienie; planista może po prostu zachowywać się dokładnie w ten sposób. Prawdziwy powód ma związek z prawdopodobieństwem, że w wyniku działań związanych z rzucaniem monetą pojawią się pewne wyniki. Aby osiągnąć zakleszczenie w ramach tego harmonogramu, wszystkie N monet muszą wskazywać ten sam kierunek za każdym razem, gdy są rzucane. Można wykazać, że jest to zdarzenie o zerowym prawdopodobieństwie, używając terminologii teorii prawdopodobieństwa. Nie ma realnej szansy, że tak się stanie w procesie rzucania N monet jednocześnie i nieskończenie często. (Czy to prawda dla wszystkich N? Co się dzieje, gdy N wynosi 1?) Innymi słowy, w nieskończonym wykonaniu tych protokołów dla N filozofów pozorna symetria zostanie złamana z prawdopodobieństwem 1, czyli na pewno przez jakiś nierówny zbiór wyników w jednym z etapów rzucania monetą. Łatwo zauważyć, że takie złamanie symetrii skutkuje przynajmniej jednym filozofem jedzenia, dzięki czemu unika się impasu. Ważne jest, aby zdać sobie sprawę, że powyższy argument nie stanowi dowodu na to, że protokół jest wolny od zakleszczeń z prawdopodobieństwem 1. Omówiliśmy tylko jeden konkretny harmonogram i jeden konkretny zestaw rzutów monetą, który skutkuje zakleszczeniem, pokazując, że jest zerowy prawdopodobieństwo. A co, jeśli moneta wskazuje na lewo

dla filozofów o numerach parzystych, a na prawo dla nieparzystych, a planista wielokrotnie daje pierwszym filozofom jeden obrót, a drugim jeden obrót? A jeśli jest siedmiu filozofów, którzy zmieniają się cyklicznie, pomijając za każdym razem dwóch? Uogólnienie argumentu do wstrzymania dla wszystkich harmonogramów nie jest proste, ale można to zrobić, dzięki czemu przedstawiony właśnie protokół jest rzeczywiście wolny od zakleszczeń w dowolnym harmonogramie. Jednak to rozwiązanie nadal nie jest satysfakcjonujące, ponieważ dopuszcza lokauty lub głód. Oto program planujący, który z pewnością spowoduje, że wszyscy oprócz jednego filozofa zostaną zablokowani; innymi słowy, wszyscy zostaną pozbawieni pożywienia z prawdopodobieństwem 1. Planista najpierw aranżuje sytuację, w której wszyscy filozofowie znajdują się w punkcie (1.2). Teraz będzie działać w taki sposób, aby zagwarantować, że z prawdopodobieństwem 1 jeden z filozofów, powiedzmy Platon, będzie jadł nieskończenie często, podczas gdy każdy z pozostałych w końcu zje ostatnią kolację, a następnie będzie głodował na zawsze. Pomysł opiera się na fakcie, że jeśli filozofowi da się możliwość wielokrotnego jedzenia, aż jego rzut monetą (1.2) przyniesie jakiś konkretny pożądany kierunek, to z prawdopodobieństwem 1 w końcu ten kierunek się pojawi. Wykorzystując ten fakt, planista będzie teraz ignorował wszystkich oprócz Platona, pozwalając mu znaleźć swoje widelce i jeść wielokrotnie, o ile losuje dokładnie w rzucie monetą (1.2). Platon zostanie zatrzymany, gdy po raz pierwszy zremisuje i, jak wspomniano, z prawdopodobieństwem 1 rzeczywiście tak się stanie. Platon jest teraz pozostawiony w stanie zawieszenia po tym ostatnim wykonaniu (1.2), ale przed instrukcją check-and-lift (1.3), a jego sąsiad po prawej stronie otrzymuje zielone światło. Ona również może jeść wielokrotnie, dopóki nie wyciągnie w lewo, w którym to czasie zostaje również pozostawiony po ukończeniu (1.2) i wykonaniu (1.3). To dzieje się wokół stołu w cyklicznej kolejności przeciwnej do ruchu wskazówek zegara, dając każdemu filozofowi podłogę (i spaghetti), dopóki nie zacznie rysować w lewo. Kiedy to się skończy, wszyscy filozofowie są gotowi do wykonania punktu (1.3), przy czym lewy jest ich wybranym kierunkiem. Ponieważ prawdopodobieństwo wynosi 1, że szał jedzenia każdego filozofa w końcu zakończy się z monetą pokazującą lewą stronę, prawdopodobieństwo osiągnięcia tej wspólnej sytuacji wynosi również 1. Teraz wszyscy filozofowie mogą zmaterializować swój ostatni wybór lewicy przez planistę, i aby podnieść lewy widelec. Żaden jeszcze się nie rozpoczął (1.4). Od tego momentu nasz planista pozwoli tylko Platonowi jeść; wszyscy inni będą głodować na zawsze. Sposób na osiągnięcie tego jest następujący. Prawy sąsiad Platona może kontynuować. Spogląda w prawo, widzi, że widelec jest niedostępny (ma go jej sąsiad po prawej stronie), odkłada widelec po lewej stronie i ponownie rzuca monetą. Trwa to bez jedzenia, dopóki nie zacznie prawidłowo rysować. Następnie zostaje ona pozostawiona w otchłani przez planistę, przed (1.3), tak że nawet nie ustaliła, że jej prawy widelec jest niedostępny, a kontrola przechodzi do jej prawego sąsiada (drugi od Platona). Filozof ten ma podobnie możliwość kontynuowania (podobnie bez udanego jedzenia), aż on również odwróci się w prawo i zostanie zatrzymany w tym samym punkcie, tuż przed (1.3). Ta procedura jest przeprowadzana dla wszystkich filozofów, w kolejności przeciwnej do ruchu wskazówek zegara wokół stołu, aż do samego Platona, ale z wyłączeniem. Sytuacja jest teraz taka, że wszystkie widelce są na stole, z wyjątkiem lewej ręki Platona, którą trzyma, a wszyscy inni filozofowie są gotowi spojrzeć w prawo w (1.3). Teraz Platon może kontynuować, co robi, podnosząc prawy widelec i jedząc. W rzeczywistości może on wielokrotnie przejść przez cały protokół, o ile wybierze pozostawienie w (1.2), za każdym razem dobrze się odżywiając i zostaje zatrzymany przez planistę, gdy po raz pierwszy rysuje prawidłowo. Teraz sytuacja jest dokładnie taka, jak po początkowym szale jedzenia Platona, ale z tym, że prawo jest jego ostatnim wyborem, a nie lewą. Harmonogram działa teraz dokładnie tak, jak poprzednio, ale z odwróconymi kierunkami, przechodząc od sąsiada do sąsiada po lewej stronie w kolejności zgodnej z ruchem wskazówek zegara, czekając, aż wszyscy narysują w lewo, aż do osiągnięcia Platona. Znowu je wielokrotnie, dopóki nie wyciągnie w lewo i cała procedura się powtarza. Z wyjątkiem Platona, wszyscy filozofowie wyraźnie głodują po zakończeniu pierwszej rundy jedzenia. Z drugiej strony Platon je nieskończenie często. Co więcej, wszystkie ewentualności, które zostały

przywołane przy opisie tego najbardziej złośliwego programu planującego, w rzeczywistości występują z prawdopodobieństwem 1. W konsekwencji wszyscy filozofowie, z wyjątkiem Platona, będą głodni z prawdopodobieństwem 1. Tak więc, jak powiedzieliśmy, przedstawiony wcześniej protokół unika impasu, ale niestety przyznaje się do głodu. Istnieje rozszerzenie protokołu, które daje również zerowe prawdopodobieństwo zagłodzenia. Ta wersja nie będzie tutaj prezentowana, poza zaznaczeniem, że wykorzystuje ten sam mechanizm rzucania monetą do wybierania kierunków, a także zmienne, które są nieco podobne do tych, które zastosowano w rozwiązaniu problemu prysznica. Wśród nich są dwie zmienne dla każdego filozofa - jedna informuje dwóch sąsiadów, że chce jeść, a druga (wspólna dla filozofa i jego sąsiadów) wskazuje, który z nich jadł jako ostatni. Żadna ze zmiennych nie jest scentralizowana, ponieważ każda jest wspólna dla co najwyżej dwóch lokalnie sąsiadujących ze sobą filozofów. Tak więc w rzeczywistości istnieje w pełni symetryczne, w pełni rozproszone rozwiązanie problemu filozofów jedzenia i jest ono całkiem satysfakcjonujące, pomimo dowodu w Części 10, że takie rozwiązanie nie istnieje.1 Istnieje tylko w obecności bardziej liberalnego pojęcia poprawności użytego tutaj: nie absolutna poprawność, ale raczej poprawność z prawdopodobieństwem 1.

Algorytmy probabilistyczne dla konwencjonalnych problemów algorytmicznych

Widzieliśmy, że problem filozofów jedzenia, z wymogami symetrii i rozdzielności, nie może być rozwiązany bez pomocy randomizacji. Jest jednak coś niepokojącego w omawianych właśnie rozwiązaniach probabilistycznych. Wydają się polegać na swoim sukcesie na nieskończonym, wiecznym charakterze protokołów, ponieważ w takich rozwiązaniach to, co powoduje, że pewne zdarzenia zachodzą z prawdopodobieństwem 1, to fakt, że myślimy o nieskończonym przedziale czasowym, a rzeczy mogą się dziać dowolnie daleko przyszłość. Rzeczywiście, mamy wrażenie, że gdyby przedstawić jakąś wersję problemu, która obejmuje tylko skończone okresy czasu, cały zbudowany powyżej probabilistyczny budynek zawaliłby się i nie byłoby wiele do powiedzenia. W jakiś sposób wydaje się, że naprawdę interesującym problemem jest to: czy probabilizm lub randomizacja może poprawić sytuację, jeśli chodzi o zwykłe, konwencjonalne problemy algorytmiczne, takie, które akceptują dane wejściowe i muszą się zatrzymać z pożądanymi wynikami? Odpowiedź na to również brzmi: tak. Zanim podamy konkretne przykłady, wyobraźmy sobie następującą sytuację, która nie różni się od historii rosyjskiej ruletki, z tym wyjątkiem, że wolimy mówić o pieniądzach ludzi, a nie o ich życiu. Załóżmy, że z jakiegoś niewyjaśnionego powodu wszystkie nasze pieniądze były związane z problemem małpiej układanki z części 7 w następujący sposób. Dostajemy jeden duży przypadek problemu (powiedzmy 225 kart z małpami) i powiedziano nam, że nasze pieniądze zostaną podwojone, jeśli udzielimy właściwej odpowiedzi tak/nie na pytanie, czy karty można ułożyć w legalną 15 15 kwadratowych. Mówi się nam również, że wiele stracimy, jeśli udzielimy złej odpowiedzi. Co więcej, nasze pieniądze są niedostępne, dopóki nie udzielimy jakiejś odpowiedzi. Ponieważ problem małpiej łamigłówki jest NP-zupełny, mamy własny problem. Co powinniśmy zrobić? Moglibyśmy uruchomić nasz ulubiony algorytm z czasem wykładniczym na kartach wejściowych, mając nadzieję, że ten konkretny zestaw jest łatwy, tak że algorytm będzie w stanie poradzić sobie z nim dość szybko, lub możemy ustawić się na podłodze i zacząć próbować. na własną rękę. Biorąc pod uwagę dyskusje w rozdziale 7, te możliwości mają pewne oczywiste wady. Alternatywnie, zdając sobie sprawę z beznadziejności sytuacji, możemy po prostu rzucić monetą, powiedzieć losowo tak lub nie i mieć nadzieję na najlepsze. Czy jest lepszy sposób? Pozostając w wyimaginowanym trybie myślenia, załóżmy, że zaoferowano nam algorytm, który rozwiązał problem małpiej łamigłówki, ale z niewielkim prawdopodobieństwem błędu. Powiedzmy, że mieliśmy gwarancję, że losowo, raz na 2200 egzekucji, podało to złą odpowiedź. Byłby to doskonały sposób na rozwiązanie tego dylematu. Bez wątpienia po prostu uruchomilibyśmy algorytm na kartach wejściowych i przedstawilibyśmy odpowiedź naszemu oprawcy. Szanse na utratę naszych pieniędzy byłyby, tak jak w 77-rundowej rosyjskiej ruletce, znacznie mniejsze niż szanse na przejechanie podczas przekraczania drogi do centrum komputerowego i znacznie, znacznie mniejsze

niż szanse, że podczas egzekucji w komputerze, na którym zaimplementowany jest algorytm, wystąpi błąd sprzętowy. Faktem jest, że w przypadku wielu problemów algorytmicznych, w tym niektórych, które wydają się być nierozwiązywalne, takie algorytmy istnieją (nie, o ile nam wiadomo, dla problemu łamigłówki małp, ale dla wielu podobnych). Algorytmy te mają charakter probabilistyczny, ponieważ wykorzystują losowe rzucanie monetą, i dlatego są czasami nazywane algorytmami probabilistycznymi lub randomizowanymi. Ze wszystkich praktycznych celów, które przychodzą mi na myśl, takie algorytmy są całkowicie zadowalające, niezależnie od tego, czy chodzi o pieniądze lub życie jednostki, przyszłość finansową firmy, czy bezpieczeństwo lub dobrobyt całego kraju. Spójrzmy teraz na przykład niezwykłego probabilistycznego rozwiązania problemu, który przez bardzo długi czas uważano za nierozwiązywalny.

Generowanie dużych liczb pierwszych

Część 7 wspomniała o liczbach pierwszych lub po prostu liczbach pierwszych w skrócie. Liczba pierwsza to dodatnia liczba całkowita, której jedynymi czynnikami (to znaczy liczbami, które dzielą ją dokładnie, bez reszty) są 1 i sama liczba. Inne liczby są określane jako złożone; są tylko wielokrotnościami liczb pierwszych. Pierwsze kilka liczb pierwszych to 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, . . . Liczb pierwszych jest nieskończenie wiele i są one rozłożone na całe spektrum dodatnich liczb całkowitych w następujący sposób. Liczba liczb pierwszych mniejszych niż podana liczba N jest rzędu N/log N. Jako konkretne przykłady, jest 168 liczb pierwszych mniejszych niż 1000, około 78 500 liczb pierwszych mniejszych niż milion i około 51 milionów liczb pierwszych mniejszych niż miliard. Spośród wszystkich 100-cyfrowych liczb mniej więcej jedna na 300 jest liczbą pierwszą, a w przypadku 200-cyfrowych jest to mniej więcej jedna na 600. Liczby pierwsze bez wątpienia stanowią najciekawszą klasę liczb, jaka kiedykolwiek przyciągnęła uwagę matematyków. Odgrywają kluczową rolę w gałęzi matematyki znanej jako teoria liczb i mają wiele niezwykłych właściwości. Ich badania doprowadziły do jednych z najpiękniejszych wyników w całej matematyce. Co więcej, jak zobaczymy w rozdziale 12, liczby pierwsze szybko stają się niezbędne w kilku ekscytujących zastosowaniach algorytmiki, w których ważna jest możliwość dość szybkiego generowania dużych liczb pierwszych. Załóżmy teraz, że jesteśmy zainteresowani wygenerowaniem nowej dużej liczby pierwszej, powiedzmy o długości 150 cyfr. Gdybyśmy tylko mieli dobry sposób testowania pierwszości dużych liczb, moglibyśmy wykorzystać sposób, w jaki liczby pierwsze są rozłożone między wszystkie dodatnie liczby całkowite, aby w umiarkowanym czasie znaleźć 150-cyfrową liczbę pierwszą. Aby to zrobić, po prostu wielokrotnie generowalibyśmy losowo nieparzyste 150-cyfrowe liczby (poprzez rzucanie monetami, aby wybrać cyfry) i testowalibyśmy każdą z nich pod kątem pierwszości, aż trafimy na tę, która jest pierwsza. Istnieje bardzo duże prawdopodobieństwo (ponad 90%), że znajdziemy go w ciągu pierwszych 1000 prób, a duża szansa, że znajdziemy go po znacznie krótszym czasie. W każdym razie, jeśli będziemy uważać, aby nie wybrać dwa razy tej samej liczby, na pewno niedługo ją znajdziemy. Problem wydajnego generowania dużych liczb pierwszych sprowadza się zatem do testowania pierwszości dużych liczb. Ale jak testujemy liczbę pod kątem pierwszości? Oto jeden prosty sposób. Mając liczbę N, podziel ją przez wszystkie liczby od 2 aż do N − 1. Jeśli okaże się, że którakolwiek z tych wartości dzieli N równo, zatrzymaj się i powiedz nie, ponieważ właśnie znalazłeś czynnik N inny niż 1 i N samo. Jeśli okaże się, że wszystkie dzielenia N − 2 pozostawiają resztki, zatrzymaj się i powiedz tak, N jest liczbą pierwszą. Ten algorytm jest w porządku, z wyjątkiem tego, że jest nadmiernie nieefektywny. W rzeczywistości, jak wyjaśniono w Części 7, przebiega wykładniczo czas, w którym jako wielkość wejścia przyjmuje się liczbę cyfr w N.2 Testowanie w ten sposób liczb 10- lub 15-cyfrowych jest wykonalne, ale testowanie liczb 150-cyfrowych nie jest możliwe, ponieważ zajęłoby to miliardy lat najszybsze komputery (patrz rozdział 7). Oczywiście ten algorytm można ulepszyć, dzieląc liczbę kandydującą N tylko przez 2 i przez liczby nieparzyste od 3 do pierwiastka kwadratowego z N lub, co ważniejsze, ostrożnie pomijając wszystkie wielokrotności już uwzględnionych dzielników. Jednak żadne takie naiwne ulepszenia nie eliminują

superwielomianowego czasu potrzebnego na rozwiązanie, a zatem ta brutalna siła jest całkowicie bezużyteczna. Do niedawna nie wiedziano, że pierwszorzędność jest w P. Jak wspomniano w rozdziale 7, od prawie 30 lat było wiadomo, że jest w NP, a więc ma krótkie certyfikaty, ale nikt nie wiedział o jakimkolwiek algorytmie wielomianowym do testowania prymat i zawsze istniała możliwość, że problem okaże się nie do rozwiązania. Wobec braku rozwiązania problemu opartego na czasie wielomianowym, na przestrzeni lat opracowano kilka pomysłowych podejść do obejścia. Obejmowały one algorytm wielomianowy do testowania pierwszości, który opierał się pod względem poprawności na głębokiej, ale nieudowodnionej hipotezie matematycznej, zwanej rozszerzoną hipotezą Riemanna. Gdyby ta hipoteza kiedykolwiek okazała się prawdziwa, problem pierwszości automatycznie stałby się członkiem P. Co więcej, nawet bez polegania na hipotezie Riemanna, ludzie byli w stanie wymyślić algorytmy testowania pierwszości, które działają w „prawie" czas wielomianowy. Ten, który przez długi czas był najlepszy pod względem wydajności rzędu wielkości, przebiegał w czasie O(NO(log log N)), który można uznać za bardzo bliski czasowi wielomianowemu, ponieważ log log N rośnie bardzo wolno: jeśli podstawa logarytmu wynosi 2, pierwsze N, dla którego log log N osiąga 5, wynosi więcej niż cztery miliardy. Bardzo niedawno wykazano, że pierwszość jest w P. Nie będziemy tutaj opisywać algorytmu, z wyjątkiem dwóch uwag. Po pierwsze, jest to niezwykle ważny wynik, zarówno ze względu na klasyczną naturę liczb pierwszych, jak i ich znaczenie w algorytmice , a także dlatego, że kładzie kres jednemu z najbardziej znanych otwartych problemów w algorytmice. Po drugie, wykładnik wielomianu w tym tak zwanym algorytmie AKS jest nadal dość wysoki (12 w pierwotnym algorytmie, który później został sprowadzony do 8), a jego działanie jest nadal boleśnie powolne. Pod tym względem nie różni się on od algorytmu wielomianowego dla programowania liniowego, wspomnianego pod koniec części 7. Mimo to, biorąc pod uwagę, że algorytm wielomianowy jest tak nowy, wiele osób oczekuje, że zostanie znacznie udoskonalony i ulepszony, więc że w końcu stanie się praktyczny. Niemniej jednak istnieją niezwykle szybkie iw pełni praktyczne algorytmy wielomianowe do testowania pierwszości, które w rzeczywistości były dostępne na długo przed odkryciem algorytmu AKS. Są probabilistyczne.

Algorytmy probabilistyczne do testowania pierwszości

W połowie lat 70. odkryto dwa bardzo eleganckie algorytmy probabilistyczne do testowania pierwszości. Były to jedne z pierwszych probabilistycznych rozwiązań trudnych problemów algorytmicznych, które zostały znalezione i zapoczątkowały szeroko zakrojone badania, które doprowadziły do randomizacji algorytmów dla wielu innych problemów. Oba algorytmy działają w czasie, który jest wielomianem (niskiego rzędu) liczby cyfr w liczbie wejściowej N, i oba mogą testować pierwszorzędność liczby 150-cyfrowej z znikomym prawdopodobieństwem błędu w ciągu kilku sekund na nośniku komputer wielkości! Algorytmy opierają się na losowym wyszukiwaniu określonych rodzajów świadectw lub świadków złożoności N. Jeśli taki świadek zostanie znaleziony, algorytm może bezpiecznie zatrzymać się i powiedzieć „nie, N nie jest liczbą pierwszą", ponieważ uzyskał niepodważalne dowody, że N jest złożone. Jednak wyszukiwanie musi być skonstruowane w taki sposób, aby w dość wczesnym momencie algorytm mógł przerwać wyszukiwanie i zadeklarować, że N jest liczbą pierwszą, z bardzo małym prawdopodobieństwem popełnienia błędu. Zauważ, oczywiście, że nie możemy po prostu zdefiniować świadka jako liczby od 2 do N − 1, która dokładnie dzieli N, chociaż oczywiście takie odkrycie stanowi niekwestionowany dowód, że N jest złożone. Powodem jest to, że istnieje wykładniczo wiele liczb między 2 a N − 1 (czyli wykładniczo wiele w stosunku do liczby cyfr w N), a jeśli chcemy się poddać i zadeklarować, że N jest liczbą pierwszą z małą szansą bycia źle, musielibyśmy sprawdzić prawie wszystkie z nich, co jest nierozsądne. Pomysł polega zatem na znalezieniu innej definicji świadka, takiej, która jest również szybko testowalna, ale z tą właściwością, że jeśli N jest rzeczywiście złożone, ponad połowa liczb między 1 a N − 1 jest świadkami N złożoność. W ten sposób, jeśli wybierzemy losowo pojedynczą liczbę K z zakresu od 1 do N − 1, a N jest rzeczywiście złożone, prawdopodobieństwo, że K posłuży do przekonania nas o tym fakcie, jest większe niż 12 .

Teraz możemy lepiej zrozumieć, dlaczego naiwny pomysł uczynienia K świadkiem, jeśli dzieli N, nie zadziała: ogólnie rzecz biorąc, znacznie mniej niż połowa liczb między 1 a N − 1 dokładnie dzieli liczbę złożoną N. Definicja świadka musi być bardziej subtelna. Zanim zobaczymy satysfakcjonujące podejście do definicji świadka, warto zrozumieć, dlaczego ta właściwość „ponad pół-świadkami" jest tak ważna. Sekret tkwi w idei wielokrotnego testowania wielu różnych potencjalnych świadków. Jeśli wybierzemy losowo tylko jedną K i powiemy „tak, N jest liczbą pierwszą", jeśli okaże się, że K nie jest świadkiem złożoności N, mamy szansę na mniej niż 1/2 pomyłek, ponieważ co najmniej połowa liczb z którego wybraliśmy, spowodowałoby to, że powiedzielibyśmy „nie", jeśli odpowiedź rzeczywiście brzmi „nie". Teraz, jeśli wybierzemy losowo dwa takie Ks, niezależnie, i powiemy tak, jeśli żaden z nich nie zostanie uznany za świadka, prawdopodobieństwo, że jesteśmy w błędzie jest zredukowany do 1/4 , ponieważ jest jedna szansa na cztery, że chociaż N jest naprawdę złożone, trafiliśmy dwukrotnie na nie-świadka, podczas gdy ponownie, co najmniej połowa możliwości dla każdego wyboru doprowadziłaby nas do właściwej odpowiedzi . Jeśli wybierzemy trzy Ks, prawdopodobieństwo błędu wynosi 1/8 i tak dalej. Fakt ten przekłada się od razu na algorytm. Wybierz, powiedzmy, 200 losowych liczb od 1 do N − 1 i przetestuj każdą pod kątem bycia świadkiem złożoności N. Zatrzymaj się i powiedz „nie", jeśli (i kiedy) któryś z nich zostanie uznany za świadka, i zatrzymaj się i powiedz „tak", jeśli wszyscy pomyślnie przejdą procedurę przesłuchania świadków. Ponieważ wybory Ks są od siebie niezależne, obowiązuje następujące stwierdzenie. Za każdym razem, gdy ten algorytm jest uruchamiany na liczbie pierwszej, odpowiedź z pewnością będzie twierdząca. Kiedy jest uruchamiany na liczbie złożonej, odpowiedź prawie na pewno będzie brzmieć nie, a prawdopodobieństwo, że będzie tak (jeśli nie powinno) wynosi mniej niż 1 do 2²⁰⁰. Wydaje się, że nie trzeba tutaj powtarzać historii rosyjskiej ruletki, lub fakty dotyczące oddychania, picia lub przechodzenia przez drogę, ale bez wątpienia zgodzisz się, że to działanie jest całkowicie zadowalające dla każdego praktycznego celu, w tym przypadków, w których czyjeś pieniądze lub życie zależą od udzielenia prawidłowej odpowiedzi. Jeśli użytkownik nie jest usatysfakcjonowany tym niewiarygodnym prawdopodobieństwem sukcesu, może poinstruować algorytm, aby wypróbował 201 Ks zamiast 200, a tym samym zmniejszyć o połowę prawdopodobieństwo błędu lub, powiedzmy, 500 Ks, czyniąc go niewiarygodnie małym 1/ 2⁵⁰⁰ . W praktyce, możemy dodać, uruchomienie takich algorytmów na zaledwie 50 Ks okazało się całkiem wystarczające. Pozostaje nam przedstawić wykonalną definicję świadka złożoności N i właśnie tutaj oba algorytmy się różnią. Oto krótki opis jednego z nich. Będziemy potrzebować kilku pojęć z elementarnej teorii liczb. Dla dwóch dodatnich liczb całkowitych K i N mówimy, że K jest kwadratową resztą modulo N, jeśli istnieje pewna liczba X, taka, że X2 i K dają tę samą resztę po podzieleniu przez N. Jest to oznaczone przez:

$$X^2 \equiv K \pmod N$$

Jeśli K i N nie mają wspólnych dzielników, a N jest nieparzyste, z parą K, N przypisujemy specjalną liczbę (czyli zawsze 1 lub −1). Liczba ta jest znana jako symbol Jacobiego K i N i oznaczymy ją tutaj przez Js(K,N). Jeśli N jest liczbą pierwszą, zdefiniuj Js(K, N) a 1, jeśli K jest kwadratową resztą modulo N, a -1 w przeciwnym razie. Jeśli N nie jest liczbą pierwszą, to Js(K, N) definiuje się jako iloczyn wielki symboli Jacobiego K i każdego z czynników pierwszych N, gdzie każdy czynnik pojawia się w produkcie tyle razy, ile pojawia się w Unikalny rozkład N na czynniki pierwsze. Na przykład, ponieważ 261 = 3 × 3 × 29, mamy:

Js(35, 261) = Js(35, 3) × Js(35, 3) × Js(35, 29)

Teraz można pokazać, że Js(35,3) i Js(35,29) mają odpowiednio -1 i 1 (to drugie, ponieważ 82 = 64, a 64 i 35 dają tę samą resztę modulo 29). Stąd ostateczna wartość Js(35, 261) to:

-1×-1 × 1 = 1

Jeśli K jest liczbą wybraną losowo spośród 1, 2, … , N − 1, najpierw określ, czy K i N, liczba, której pierwszorzędność próbujemy przetestować, mają jakieś wspólne czynniki inne niż 1. Jeśli tak, to oczywiście N nie jest liczbą pierwszą i test się skończył. Załóż więc, że nie. Teraz oblicz:

$$X \leftarrow K^{(N-1)/2}(\mathrm{mod}\ N)$$

to znaczy ustaw X na resztę uzyskaną z podzielenia K(N−1)/2 przez N. Oblicz także:

$$Y \leftarrow Js(K, N)$$

Jeśli X = Y mówimy, że K jest świadkiem złożoności N, a cała procedura zatrzymuje się i mówi „nie, N nie jest liczbą pierwszą". Jeśli natomiast X = Y , K nie jest świadkiem. Możliwe jest wykazanie, używając argumentów teorii liczb, że dla każdej liczby pierwszej N wynikowe X i Y są równe, tak że jeśli rzeczywiście wybrane K daje początek X = Y, mamy pełne prawo do wniosku, że N nie jest liczbą pierwszą. Z drugiej strony X = Y nie gwarantuje, że N jest liczbą pierwszą. Zdarza się jednak (chociaż wcale nie jest to łatwe do udowodnienia), że jeśli N naprawdę nie jest liczbą pierwszą, to równość X = Y obowiązuje dla co najwyżej połowy liczb N − 2 między 1 a N − 1. Tak więc, zgodnie z wymaganiami prawdopodobieństwo błędnego udzielenia wskazanych odpowiedzi tak/nie dla losowo wybranego K jest nie większe niż 1 2 , tak że powtórzenie procedury dla, powiedzmy, 200 takich Ks daje pożądany algorytm. Aby dokończyć tę historię, musimy być w stanie sprawnie stwierdzić, czy K i N mają jakieś wspólne czynniki, oraz szybko obliczyć X i Y. To pierwsze można osiągnąć, stosując szybki algorytm gcd Euklidesa, a w przypadku drugiego istnieją również szybkie algorytmy, o których tutaj nie będziemy się rozwodzić. To kończy krótkie omówienie szybkiego algorytmu probabilistycznego do testowania pierwszości. Warto zauważyć, że drugie rozwiązanie daje jeszcze mniejsze prawdopodobieństwa błędów, w tym sensie, że ułamek świadków złożoności N wśród liczb od 1 do N − 1 wynosi co najmniej 3/4 , a nie 1/2 . Tak więc, ponieważ więcej Ks jest wybieranych losowo, prawdopodobieństwo błędu zmniejsza się dwukrotnie szybciej. Efekt 1 na 2²⁰⁰ można zatem osiągnąć za pomocą 100 losowych Ks, a nie 200. Mamy więc niezwykle szybkie algorytmy probabilistyczne do testowania pierwszości, które są znacznie lepsze w praktyce - przynajmniej dla obecnie - niż jakiekolwiek inne, w tym najnowszy nieprobabilistyczny algorytm AKS z czasem wielomianowym. W przeciwieństwie do tego, problem stwierdzenia, że czynniki liczby, nawet znanej z wyprzedzeniem, są złożone, nie wydaje się dopuszczać nawet rozwiązania probabilistycznego, które przebiega w czasie wielomianowym. Tak więc, podczas gdy testowanie pierwszości stało się wykonalne zarówno w zasadzie (AKS), jak iw praktyce (algorytmy probabilistyczne), faktoring nie stał się taki w żadnym z nich. Zgodnie z obietnicą, w następnym rozdziale zobaczymy zaskakujące zastosowania szybkiego testowania pierwszości i, co ciekawe, opierają się one dokładnie na tej różnicy między testowaniem pierwszości a faktoringiem.

Szybkie dopasowywanie wzorców probabilistycznych

Testowanie pierwszości jest uderzającym przykładem trudnego problemu, który można rozwiązać, gdy dozwolone jest rzucanie monetą. Mówiąc bardziej skromnie, istnieje wiele przykładów problemów, które mają rozsądne rozwiązania, ale dla których randomizacja może jednak znacznie poprawić sytuację. Rozważmy przykład. Załóżmy, że chcemy ustalić, czy dany wzorzec symboli występuje w długim tekście3. Załóżmy, że wzorzec ma długość M, a tekst o długości N. Oczywiście każdy algorytm, który rozwiązuje problem, musi w najgorszym przypadku uwzględniać wszystkie pozycji w całym tekście, tak aby O(N) było wyraźnie dolnym ograniczeniem złożoności czasowej problemu. (Czy możesz w tym celu sformułować precyzyjny argument?) Naiwny algorytm wymaga przeszukania tekstu i sprawdzenia na każdej pozycji, czy kolejne M symboli tekstu idealnie pasuje do M symboli wzorca.

Może to prowadzić do najgorszego przypadku zachowania czasowego O(N × M), co jest nieodpowiednie, chyba że wzór jest bardzo krótki. Jeśli na przykład szukamy krótkiego słowa w Encyklopedii Britannica, ta procedura może być wykonalna, mimo że N wynosi około miliarda. Jeśli jednak Britannica ma być przeszukiwana pod kątem ciągu znaków o długości 1000 lub miliona, naiwny algorytm jest beznadziejnie powolny. Chociaż istnieje wiele sprytnych algorytmów czasu liniowego dla tego problemu, są one nieco skomplikowane, a ich stałe są zbyt wysokie, aby były pomocne, gdy M jest również duże. Większość z nich wymaga również sporej ilości pamięci przestrzeni. Wprowadź rzucanie monetą. Poniższy algorytm wykorzystuje pomysł zwany odciskiem palca. Zamiast porównywać wzorzec, symbol po symbolu, z każdym możliwym blokiem M sąsiednich symboli tekstu, używamy funkcji odcisku palca, która wiąże liczbę z każdym łańcuchem symboli o długości M. Następnie, gdy algorytm przechodzi przez tekst, biorąc pod uwagę każdy możliwy blok M z kolei, to te reprezentatywne liczby, a nie same ciągi, są porównywane. Brzmi to całkiem prosto, ale nie wydaje się stanowić ulepszenie w stosunku do górnej granicy O(M × N), ponieważ nadal musimy brać pod uwagę każdą z liter M w każdej z możliwych N lub mniej potencjalnych lokalizacji wzorca. Możemy jednak poprawić sytuację, jeśli zostaną spełnione następujące dwa wymagania: (1) numer odcisku palca jest sam w sobie znacznie krótszy niż M, najlepiej o długości log N lub log M, oraz (2) odcisk palca bloku z symbolem M jest obliczalna w czasie mniejszym niż O(M), najlepiej w czasie stałym. Wymóg (1) nie jest spełniony przez proste tłumaczenie bloku M symboli na cyfry (jak to zrobiono w części 9 w przypadku redukcji między maszynami Turinga a programami licznikowymi). Funkcja odcisków palców musi być bardziej subtelna. Oto, co zrobimy. Liczba pierwsza K składająca się z około log N cyfr binarnych jest wybierana losowo, a odcisk palca bloku B o symbolu M jest uważany za resztę uzyskaną po podzieleniu zdigitalizowanej wersji B przez K. Tak więc dla Britanniki potrzebujesz liczby pierwszej o długości około 30 cyfr binarnych lub około 10 cyfr dziesiętnych.

Ta definicja spełnia warunek (1), ponieważ reszty z dzieleń przez K mieszczą się w zakresie od 0 do K − 1, a zatem nie są dłuższe niż log N, który jest rzędu długości samego K. Wymóg (2) również jest spełniony. Aby zobaczyć, jak to zrobić, załóżmy (jak w części 9), że tekst jest zbudowany z alfabetu składającego się z 10 symboli, dzięki czemu można go traktować po prostu jako długą liczbę dziesiętną. W ten sposób bloki z symbolem M stają się po prostu liczbami M-cyfrowymi, które, dla ułatwienia, są rozumiane w odwrotnej kolejności, przy czym najmniej znacząca cyfra znajduje się po lewej stronie. Ogólnie rzecz biorąc, obliczenie odcisku palca liczby M-cyfrowej (tj. jej reszty modulo K) zajmuje niestałą ilość czasu, w zależności od M. Jednak tutaj podróżujemy wzdłuż długiego ciągu cyfr i możemy użyć odcisku palca jednej M-cyfrowej liczby, aby szybko obliczyć następną. Rozważmy na przykład tekst

9 8 3 3 4 1 1 5 86 4 4 9 32 2 9 1 6 1 5 . . .

Niech M = 6 i załóżmy, jak pokazano na rysunku, że osiągnęliśmy dziewiątą pozycję (tj. drugą 8 od lewej), po obliczeniu odcisku palca J jako reszty liczby 394.468 modulo wybranej liczby pierwszej K. nowy odcisk palca, nazwijmy go J , to pozostałość z 239 446 modulo K. Korzystając z faktu, że drugą z tych sześciocyfrowych liczb otrzymuje się od pierwszej przez odjęcie 8, podzielenie przez 10 (pozostawiając 39 446), a następnie dodanie 200 000, J może otrzymać z J za pomocą trzech prostych operacji arytmetycznych, przeprowadzonych modulo K, tj. mając na uwadze reszty modulo K. Każda z tych operacji zajmuje tylko stały czas, ponieważ na przykład istnieje tylko 10 możliwości dla liczby, która ma zostać dodana, w tym przypadku 200 000, a resztę tych modulo K można obliczyć z góry i przechowywać w tabeli . W związku z tym cały algorytm (który przechodzi przez cały tekst, porównując każdy symbol M ze wzorcem, dopóki nie znajdzie pasujących odcisków palców) można łatwo zauważyć, że działa w czasie liniowym niższego rzędu. Dokładniej, zajmuje czas O(N + M), z bardzo małą stałą i pomijalną przestrzenią pamięci.

To dobra wiadomość. Zła wiadomość jest taka, że algorytm w obecnej postaci może się pomylić. Oczywiście, jeśli odciski palców okażą się nierówne, rzeczywiste ciągi (wzór i blok symbolu M, względem którego jest sprawdzany) również muszą być nierówne. Jednak odwrotność niekoniecznie jest prawdziwa. Dwa różne bloki symbolu M mogą mieć ten sam odcisk palca, ponieważ muszą one dać tę samą resztę tylko po podzieleniu przez K, właściwość wspólną dla wielu różnych liczb. Algorytm może zatem znaleźć nieprawidłowe „dopasowania". Jednak, jak każda dobra historia, ta również ma szczęśliwe zakończenie. Można wykazać, że w naszym przypadku prawdopodobieństwo, że jeden z bloków M-symbolu w tekście będzie miał taką samą resztę po podzieleniu przez losową liczbę pierwszą K, jak ma to miejsce we wzorcu M-symbolu, chociaż nie są one równe łańcuchom symboli, wynosi około 1 na N. Innymi słowy, nawet jeśli złośliwy przeciwnik próbuje ustawić wiadomość i wzorca, który wygeneruje wiele różnych bloków z tym samym odciskiem palca, fakt, że K jest wybierane losowo po podaniu tekstu i wzorca, gwarantuje, że, z prawdopodobieństwa, niezgodność zostanie błędnie zadeklarowana jako dopasowanie tylko mniej więcej raz w ciągu całego przebieg algorytmu. Aby upewnić się, że nawet to zdalne zdarzenie nie doprowadzi do błędnej odpowiedzi, możemy zmodyfikować algorytm tak, aby po znalezieniu dopasowania odcisku palca algorytm faktycznie sprawdzał rzekomo pasujące bloki, symbol po symbolu, przed zatrzymaniem i zadeklarowaniem dopasowania. Jeśli tak się stanie, że jest to jedyny przypadek, w którym porównywanie odcisków palców jest błędne, algorytm będzie kontynuował wyszukiwanie innych dopasowań. Jak wspomniano, istnieje bardzo mała szansa, że wiele z tych kosztownych podwójnych kontroli będzie koniecznych. Zwykle w każdym przebiegu algorytmu będzie mniej więcej jeden, co nie zmieni jego wydajności O(M + N). Wyłania się tu wybór dwóch wersji tego algorytmu. Pierwszy, w którym nie przeprowadza się podwójnej kontroli, gdy odciski palców są zgodne, gwarantuje przebieg liniowy, ale (z małym prawdopodobieństwem) może się pomylić, a drugi, w którym dokonuje się podwójnej kontroli, gwarantuje, że nie błądzić, ale (z małym prawdopodobieństwem) może działać dłużej niż czas liniowy. Istnieją nazwy dla tych różnych rodzajów algorytmów probabilistycznych. Te, które zawsze są szybkie i prawdopodobnie prawidłowe, nazywane są Monte Carlo, a te, które zawsze są prawidłowe i prawdopodobnie szybkie, nazywane są Las Vegas. Algorytmy probabilistyczne testowania pierwszości są więc typu Monte Carlo, a do dopasowywania wzorców mamy jeden z nich.

Probabilistyczne klasy złożoności

Algorytmy probabilistyczne można sformalizować za pomocą probabilistycznych maszyn Turinga. Są to niedeterministyczne maszyny Turinga, w których wyborów dokonuje się poprzez rzucanie normalnymi, bezstronnymi monetami, a nie magicznymi. Klasa RP (oznaczająca losowy czas wielomianowy) jest zdefiniowana jako klasa problemów decyzyjnych, dla których istnieje probabilistyczna maszyna Turinga z wielomianowym czasem wielomianowym o następującej właściwości. Jeśli poprawną odpowiedzią dla wejścia X jest nie, maszyna mówi nie z prawdopodobieństwem 1, a jeśli poprawna odpowiedź brzmi tak, maszyna mówi tak z prawdopodobieństwem większym niż 12 . Oczywiście zainteresowanie problemami RP wynika z faktu, że dla dowolnego X te potencjalnie błędne algorytmy można wielokrotnie powtarzać, uzyskując malejące prawdopodobieństwo błędu, jak wyjaśniono szczegółowo wcześniej. Klasa dopełniacza, co-RP, zawiera te problemy, których dopełnienia są w RP. Problem co-RP dopuszcza zatem probabilistyczną wielomianową maszynę Turinga, która z prawdopodobieństwem 1 mówi tak dla danych wejściowych tak, a z prawdopodobieństwem większym niż 12 mówi nie dla danych wejściowych. Klasa RP leży pomiędzy P i NP. Każdy wykonalny problem, to znaczy taki, który można rozwiązać w konwencjonalnym czasie wielomianu, jest w RP (dlaczego?), a każdy problem RP jest w NP, a zatem można go rozwiązać przez magiczny niedeterminizm w czasie wielomianu. (Dlaczego?) Problemy co-RP są podobne między P i co-NP. Również tutaj niektórzy badacze uważają, że inkluzje w sekwencji:

$P \subseteq RP \subseteq NP$

są surowe, ale nikt nie wie na pewno. Na przykład, tak jak pierwszorzędność okazała się być w P, tak może się stać z wszystkimi problemami NP, powodującymi upadek tej trójdrożnej hierarchii. Interesujące jest więc to, że w rozsądnym czasie nie wiemy, czy rzucanie monetą zapewnia jakąkolwiek realną dodatkową moc, czy też magiczne rzucanie monetą daje jeszcze więcej. Jeśli chodzi o uniwersalną moc algorytmiczną, teza Kościoła/Turinga rozciąga się również na algorytmy probabilistyczne. Randomizacja, podobnie jak współbieżność, nie może być stosowana do rozwiązywania tego, co nieobliczalne lub nierozstrzygalne, przynajmniej nie w ramach tych definicji: możliwe jest symulowanie każdej probabilistycznej maszyny Turinga za pomocą konwencjonalnej maszyny Turinga, a ponadto można to zrobić co najwyżej wykładnicza strata czasu. Rzucanie monetą nie rozszerza więc naszych absolutnych możliwości algorytmicznych, ale pozwala lepiej je wykorzystać. Być może w końcu uda się wykorzystać rzucanie monetą do rozwiązywania w praktyce nierozwiązywalnych problemów. Jednak jak na razie nie znamy żadnego szybkiego algorytmu probabilistycznego dla jakiegokolwiek nierozwiązywalnego problemu. Przypuszcza się, że niektóre problemy, które przypuszcza się, że są nierozwiązywalne, pozostają takie nawet w obliczu prawdopodobieństwa. Jednym z przykładów jest faktoring liczb, do którego będziemy mieli okazję wrócić później.

Badania nad algorytmami probabilistycznymi

Choć na razie musi to brzmieć powtarzalnie, tematy poruszane w tej części są również przedmiotem intensywnych badań. Randomizacja w algorytmice to niezwykle ekscytujący pomysł. Odkryto wiele nowych zastosowań i wiele wyników zostało ustalonych od czasu, gdy ten obszar badań rozpoczął się na dobre około 25 lat temu. Podobnie jak w przypadku równoległości i współbieżności, naukowcy pracują zasadniczo w dwóch różnych kierunkach. Pierwszym z nich jest poszukiwanie wydajnych algorytmów zrandomizowanych dla konwencjonalnych problemów algorytmicznych zorientowanych na wejście/wyjście (jest to analogiczne do poszukiwania wydajnych algorytmów zrównoleglonych dla takich problemów), a drugim jest znalezienie sposobów wykorzystania rzucania monetą w rozwiązywaniu problemów z natury wieczysty, ciągły charakter, który zazwyczaj obejmuje przetwarzanie rozproszone. Probabilistyczne testowanie pierwszości jest przykładem pierwszego, a protokół rzucania monetą dla filozofów jedzenia jest przykładem drugiego. Badaczy interesują oba rodzaje algorytmów probabilistycznych, Monte Carlo i Las Vegas, a także inne, a także relacje między nimi. Niektóre z najtrudniejszych problemów pojawiają się, gdy połączy się współbieżność i randomizację. Jeśli w części 10 stwierdziliśmy, że współbieżność sprawia, że specyfikacja i weryfikacja są subtelne i śliskie, to dodanie rzucania monetą czyni je podwójnie. Formalne sprawdzenie, czy protokół probabilistycznych filozofów jedzenia jest wolny od impasu z prawdopodobieństwem 1, jest bardzo delikatnym i żmudnym obowiązkiem. Badacze są zainteresowani znalezieniem zadowalających metod dowodowych dla protokołów i algorytmów probabilistycznych, a także konstruowaniem probabilistycznych logik dynamicznych, które umożliwiają określenie i udowodnienie wielu różnych właściwości takich algorytmów w rygorystyczny sposób matematyczny. Innym ciekawym tematem badawczym jest klasyfikacja problemów algorytmicznych na probabilistyczne klasy złożoności. Klasy takie jak RP, co-RP i ich przecięcie (czasami nazywane ZPP), a także szereg klas dodatkowych, są badane zarówno z konkretnego, jak i bardziej abstrakcyjnego punktu widzenia. W ujęciu konkretnym celem jest próba znalezienia interesujących problemów, które tkwią w tych klasach i udowodnienie, że inni nie. W podejściu abstrakcyjnym celem jest poszukiwanie immanentnych właściwości tych klas oraz ich związku z innymi klasami złożoności, zarówno probabilistycznymi, jak i nieprobabilistycznymi.

Jako przykład przywołajmy sieci logiczne z Części 10. Twierdziliśmy tam, że każdy efektywnie obliczalny problem algorytmiczny może być rozwiązany przez jednolity zbiór obwodów binarnych. Oznacza to, że

dla każdego takiego problemu istnieje skuteczny sposób generowania obwodu, który rozwiązuje problem dla danych wejściowych o danym rozmiarze N. Łatwo zauważyć, że jeśli pierwotny problem jest w P, to znaczy, że można go rozwiązać za pomocą wielomianu czas, wtedy te obwody mogą być wykonane jako wielomianowe o rozmiarze N. Wykazano, że problemy w RP, czyli te, które można rozwiązać w losowym czasie wielomianowym, również mają obwody o rozmiarze wielomianowym, i jest to prawdą, nawet jeśli problem pod ręką nie wiadomo, czy jest w P. Jednak nie wiemy, czy te obwody mogą być jednorodnie skonstruowane w czasie wielomianowym. Gdybyśmy to zrobili, mielibyśmy dowód, że każdy problem w RP jest w rzeczywistości w P. (Jak?) Jak zobaczymy w części 12, nowy, niezwykle interesujący aspekt randomizacji i obliczeń probabilistycznych wynika z jego użycia w protokołach interaktywnych. Połączenie to ma daleko idące konsekwencje dla wielu zagadnień w teorii złożoności i trudności problemów algorytmicznych i zostało wspomniane w rozdziale 7 w związku z trudnością aproksymacji problemów NP-zupełnych. Idee te są przedmiotem jednych z najbardziej intensywnych badań w teorii informatyki w ostatniej dekadzie. Szczególnie interesującym kierunkiem badań jest powiązanie algorytmów probabilistycznych z analizą probabilistyczną algorytmów konwencjonalnych. W Rozdziale 6 krótko omówiliśmy zachowanie algorytmów sekwencyjnych w przypadku przeciętnych przypadków, a w Części 7 wspomnieliśmy o pewnych podejściach aproksymacyjnych do problemów NP-zupełnych i innych. W obu przypadkach w grę wchodzi rozumowanie probabilistyczne, ponieważ należy poczynić pewne założenia dotyczące prawdopodobieństwa wystąpienia określonych danych wejściowych. Analiza odpowiada na pytania dotyczące zachowania algorytmu średnio lub na „prawie wszystkich” danych wejściowych. W ten sposób prawdopodobieństwo dyktuje formę, jaką przyjmą dane wejściowe, podczas gdy sam algorytm jest całkowicie deterministyczny. Tutaj sprawy mają się na odwrót. Zamiast martwić się o probabilistyczny rozkład danych wejściowych, sam algorytm, rzucając monetami, generuje prawdopodobieństwa, o których później myślimy. Okazuje się, że w pewnych celach te dwa podejścia są w rzeczywistości równoważne. W pewnym sensie technicznym wyniki rzutu monetą można traktować jako dodatkowe nakłady, podane na początku, i odwrotnie, generowanie rozkładu probabilistycznego na rzeczywistych nakładach można przesunąć do samego algorytmu. Zatem probabilistyczną złożoność konwencjonalnych algorytmów i konwencjonalną złożoność algorytmów probabilistycznych można postrzegać, że tak powiem, jako dwie strony tego samego medalu. Jedną z kwestii, która w ogóle nie została tutaj poruszona, jest sposób, w jaki komputery mogą rzucać uczciwymi, bezstronnymi monetami. W prezentowanych algorytmach wielokrotnie czyniono aluzję do tej umiejętności, ale dorozumiane założenie, że można to zrobić, jest nieuzasadnione, ponieważ prawdziwy komputer cyfrowy jest bytem całkowicie deterministycznym, a zatem w zasadzie wszystkie jego działania można przewidzieć w osiągnięcie. Dlatego komputer nie może generować prawdziwie losowych liczb, a zatem nie może symulować prawdziwie losowego rzucania uczciwymi monetami. Istnieje wiele sposobów na przezwyciężenie tego problemu. Jednym z nich jest odwołanie się do fizycznego źródła. Np. nasz komputer mógłby być przymocowany do małej dłoni robota, która aby wybrać losowo 0 lub 1, zgarnia garść piasku z dużego pojemnika, liczy zawarte w nim ziarenka piasku, decyduje 0 jeśli liczba jest parzysta i 1 w przeciwnym razie, a następnie wrzuca piasek z powrotem do pojemnika. Takie podejście ma kilka oczywistych wad. Istnieją jednak bardziej praktyczne fizyczne metody uzyskiwania naprawdę losowych liczb, wykorzystywane przez karty inteligentne i podobne urządzenia. Inne podejście obejmuje tak zwane liczby pseudolosowe. Jednym słowem ciąg pseudolosowy to taki, którego nie można odróżnić od ciągu naprawdę losowego w czasie wielomianowym. W następnym rozdziale omówimy funkcje jednokierunkowe, które są obliczalne w czasie wielomianowym, ale których odwrotności są niemożliwe do obliczenia. Można wykazać, że jeśli istnieją funkcje jednokierunkowe, które można udowodnić (to znaczy, gdybyśmy mogli udowodnić, że twardy kierunek tych funkcji jest naprawdę niewykonalny), to istnieją również generatory liczb pseudolosowych. Generator otrzymuje jako dane wejściowe pojedynczą liczbę, ziarno, a następnie

generuje liczby pseudolosowe w nieskończoność. Co ciekawe, sama zdolność generowania liczb losowych potrzebnych w algorytmach probabilistycznych zależy również od przypuszczeń dotyczących nierozwiązywalności innych problemów. Jeśli problem P jest naprawdę trudny, problem Q można uprościć, odwołując się do algorytmu probabilistycznego, który wykorzystuje twardość P do generowania rzutów monetą, których nie można odróżnić od losowych rzutów. Tym samym nie powiedziano ostatniego słowa nawet o najbardziej fundamentalnej kwestii leżącej u podstaw zastosowania randomizacji do algorytmiki: możliwości algorytmicznego symulowania prawdziwego lub prawie prawdziwego losowego wyboru.

Twierdzenia, które są prawie prawdziwe?

Na końcu Części 5 omówiliśmy twierdzenie o czterech kolorach, którego dowód został częściowo przeprowadzony za pomocą programu komputerowego. W pewnym sensie nie można twierdzić, że twierdzenie to zostało w pełni udowodnione, ponieważ nikt nie udowodnił poprawności programu, kompilatora lub systemu operacyjnego. Z drugiej strony, jeśli naprawdę chcemy mieć pewność, że twierdzenie jest prawdziwe, zawsze możemy spróbować udowodnić poprawność tych fragmentów oprogramowania. Dowód twierdzenia podlega zatem formalnej weryfikacji, przynajmniej co do zasady. Interesujące jest wyobrazić sobie inną sytuację, która, o ile nam wiadomo, jeszcze się nie wydarzyła. Co by się stało, gdyby jakiś ważny problem w matematyce został rozwiązany za pomocą algorytmu probabilistycznego, powiedzmy typu Monte Carlo, z malejącym prawdopodobieństwem błędu? Czy moglibyśmy wtedy umieścić „Q.E.D." na końcu dowodu? Trudność polega na tym, że nawet po formalnej weryfikacji programu, kompilatora i systemu operacyjnego, ustalimy rygorystycznie, że proces dopuszcza błąd z, powiedzmy, prawdopodobieństwem 1/2200. Co powinniśmy zrobić? Czy mamy domagać się twierdzenia prawie, czy twierdzenia o bardzo wysokim prawdopodobieństwie? Czy matematycy będą mieli wystarczające zaufanie do takiego wyniku, aby oprzeć na nim dalszy rozwój? Prawdopodobnie będziemy musieli poczekać i zobaczyć. Niektórzy ludzie odrzucają cały problem, wskazując, że wszystkie dowody matematyczne mają pewne szanse na błędność, ponieważ są przeprowadzane przez ludzi podatnych na błędy. I rzeczywiście, wiele dowodów, nawet opublikowanych, okazało się wadliwych. W rzeczywistości stało się to z samym twierdzeniem o czterech kolorach przy wielu poprzednich okazjach. Tutaj sytuacja jest inna, ponieważ mamy rygorystycznie zweryfikowany dowód twierdzenia. Jednak jeden z jego elementów, algorytm probabilistyczny, jest poprawny do udowodnienia tylko z bardzo dużym prawdopodobieństwem, które to prawdopodobieństwo może być tak wysokie, jak pożądane, przez nieco dłuższe uruchomienie algorytmu. Tak długo, jak używamy algorytmów probabilistycznych tylko do drobnych, przyziemnych spraw, takich jak bogactwo, zdrowie i przetrwanie, możemy łatwo zadowolić się bardzo prawdopodobnymi poprawnymi odpowiedziami na nasze pytania. Wydaje się, że tego samego nie można powiedzieć o naszych poszukiwaniach absolutnej prawdy matematycznej.

Kryptografia i niezawodna interakcja

Przejdźmy teraz do nowego i ekscytującego obszaru zastosowań algorytmiki. Jej nowatorską cechą jest to, że metody stosowane do rozwiązywania problemów w tym obszarze wykorzystują trudność rozwiązywania innych problemów. To samo w sobie jest dość zaskakujące, ponieważ spodziewalibyśmy się, że negatywne wyniki, które ustalają dolne granice rozwiązywania problemów algorytmicznych, nie będą miały żadnej praktycznej wartości, z wyjątkiem zapobiegania próbom poprawy tych granic. Bynajmniej. Kluczowe są tutaj problemy, dla których nie są znane żadne dobre algorytmy. Ogólnie rzecz biorąc, obszar ten dotyczy kryptografii i dotyczy potrzeby komunikowania się w bezpieczny, prywatny i niezawodny sposób. Kryptografia ma wiele różnorodnych zastosowań w kręgach wojskowych, dyplomatycznych, finansowych i przemysłowych. Zapotrzebowanie na dobre protokoły kryptograficzne znacznie wzrasta dzięki szybkiemu rozprzestrzenianiu się komputerowych systemów komunikacyjnych, w szczególności oczywiście Internetu. Coraz częściej komputery stają się odpowiedzialne za przechowywanie, manipulowanie i przesyłanie wszystkiego, od kontraktów, poleceń strategicznych i transakcji biznesowych po zwykłe poufne informacje, takie jak dane wojskowe, medyczne i osobiste. Ta sytuacja z kolei sprawia, że problemy z podsłuchiwaniem i manipulacjami są jeszcze bardziej dotkliwe. Jednym z podstawowych problemów w kryptografii jest szyfrowanie i deszyfrowanie danych. Jak zakodować ważną wiadomość tak, aby odbiorca mógł ją rozszyfrować, a nie podsłuchiwacz? Co więcej, czy wiadomość może być „podpisana" przez nadawcę, aby (1) odbiorca miał pewność, że mógł ją wysłać sam, (2) nadawca nie może później zaprzeczyć, że ją wysłał, oraz (3) odbiorca , po otrzymaniu podpisanej wiadomości, nie może podpisać w imieniu nadawcy żadnej wiadomości, nawet dodatkowych wersji samej wiadomości, która właśnie została odebrana? Kwestia podpisu dotyczy wielu aplikacji, takich jak przekazy pieniężne i umowy elektroniczne. Moglibyśmy długo kontynuować z takimi pytaniami i ich motywującymi przykładami, bo jest ich wiele, a każde rodzi nowe wyzwania. Dla niektórych z nich znaleziono eleganckie i użyteczne rozwiązania, dla innych nie ma. Zaczniemy od skoncentrowania się na problemach z szyfrowaniem i podpisem. Konwencjonalne kryptosystemy oparte są na kluczach. Są one używane do przetłumaczenia wiadomości M na jej zaszyfrowaną formę, zaszyfrowanego tekstu H, a następnie do odszyfrowania jej z powrotem do jej oryginalnej postaci. Jeśli mamy na myśli ogólną procedurę szyfrowania związaną z kluczem przez Encr i odpowiednią procedurę deszyfrowania przez Decr, możemy napisać:

H = Encr (M) i M = Decr (H)

Innymi słowy, zaszyfrowaną wersję H uzyskuje się przez zastosowanie procedury Encr do wiadomości M, a oryginalne M można pobrać z H przez zastosowanie procedury Decr do H. Prosty przykład, którego wszyscy używaliśmy w dzieciństwie, wymaga klucz K jest liczbą z zakresu od 1 do 25, aby Encr była procedurą, która zastępuje każdą literę tą, która znajduje się w K w dalszych pozycjach alfabetu, a Decr ma zamieniać każdą literę na tę znajdującą się wcześniej w K pozycji. W ten sposób Encr i Decr są wzajemnie podwójne; Decr jest odwrotnością Encr. (Dla celów liczenia liter alfabet jest uważany za cykliczny; następuje z.) To standardowe podejście można zilustrować za pomocą metafory zamkniętego pudełka. Aby wymieniać gryps z przyjacielem należy najpierw przygotować skrzynkę z bezpiecznym zatrzaskiem. Następnie kupmy kłódkę z dwoma kluczami, jeden dla nas, a drugi dla naszego przyjaciela. Następnie wysłanie wiadomości polega na włożeniu jej do pudełka, zablokowaniu pudełka za pomocą klucza i wysłaniu pudełka do miejsca przeznaczenia. Nikt nie może odczytać wiadomości po drodze, jeśli nie ma klucza, a ponieważ są tylko dwa klucze, przechowywane przez nadawcę i zamierzonego odbiorcę, system jest dość bezpieczny. Takie podejście ma kilka wad. Po pierwsze, nie odnosi się do kwestii podpisu. Odbiorcy mogą samodzielnie wymyślać fałszywe wiadomości i twierdzić, że zostały wysłane przez nadawcę, a nadawca z kolei może zaprzeczyć, że wysłał wiadomości autentyczne. Kolejna poważna wada dotyczy konieczności współpracy w doborze i

bezpiecznej dystrybucji kluczy. Ogólnie rzecz biorąc, w sieć komunikacyjną zaangażowane są więcej niż dwie strony, a aby zapewnić prywatność między dowolnymi dwoma, musi istnieć jakiś bezpieczny sposób dystrybucji par kluczy, po jednym dla każdej pary stron. Biorąc pod uwagę, że główne zastosowania współczesnej kryptografii znajdują się w środowiskach skomputeryzowanych, klucze cyfrowe nie mogą być dystrybuowane tymi samymi (niebezpiecznymi) kanałami komunikacyjnymi, co zaszyfrowane wiadomości. Konieczne byłoby zatem skorzystanie z innych, znacznie droższych metod, takich jak osobiste doręczenie przez zaufanego kuriera. Jest to oczywiście niewykonalne w aplikacjach obejmujących wiele stron.

Kryptografia klucza publicznego

W 1976 roku zaproponowano nowatorskie podejście do problemów z szyfrowaniem, deszyfrowaniem i podpisem, kryptosystem klucza publicznego. Być może najlepiej tłumaczy to wariant metafory zamkniętego pudełka. Pomysł polega na użyciu innego rodzaju kłódki, takiej, którą można zamknąć bez klucza, po prostu zatrzaśniętej. Otwarcie takiego zamka wymaga jednak klucza. Aby skonfigurować mechanizm wymiany tajnych informacji, każdy potencjalny użytkownik systemu wychodzi na własną rękę i kupuje taką kłódkę i klucz. Następnie zapisuje swoje imię na kłódce i kładzie ją na stole, w miejscu publicznym. Klucz pozostaje jednak u nabywcy. Załóżmy teraz, że strona B (powiedzmy Bob) chce wysłać wiadomość do strony A (powiedzmy Alice). Bob wkłada wiadomość do pudełka, podchodzi do stołu, podnosi kłódkę Alicji i zamyka nią pudełko. Do tego nie jest potrzebny klucz. Pudełko jest następnie wysyłane do Alice, która używa swojego klucza do otwarcia zamka i przeczytania wiadomości. Nikt poza Alicją nie ma klucza, dzięki czemu wiadomość jest bezpieczna. Zauważ, że nie jest wymagana żadna wcześniejsza komunikacja ani współpraca między Alicją i Bobem. Gdy jakakolwiek strona zdecyduje się dołączyć do gry, kupi kłódkę i upubliczni ją, ta strona może zacząć otrzymywać wiadomości. Aby zrozumieć, w jaki sposób systemy z kluczem publicznym mogą być używane w cyfrowych, skomputeryzowanych środowiskach, załóżmy, że wiadomości są (być może długimi) sekwencjami cyfr. Tak więc pewna bezpośrednia i prosta metoda tłumaczenia symboli na cyfry został już zastosowany. Kłódka Alicji to po prostu funkcja szyfrowania $Encr_A$, która przekształca liczby na inne liczby, a klucz Alicji jest sekretnym sposobem obliczania funkcji deszyfrowania $Decr_A$. W ten sposób każda ze stron upublicznia swoją procedurę szyfrowania, ale zachowuje swoją procedurę deszyfrowania jako prywatną. Aby wysłać wiadomość do Alicji, Bob używa publicznej procedury szyfrowania Alicji $Encr_A$ i wysyła Alicji numer $Encr_A$ M). Teraz Alicja może to rozszyfrować, korzystając z prywatnej procedury $Decr_A$. Aby metoda działała, obie funkcje muszą być łatwe do obliczenia, a równanie dualności $Decr_A$ ($Encr_A$ (M)) = M musi być przechowywany dla każdej wiadomości M. Co jednak najważniejsze, nie powinno być możliwe wywnioskowanie metody obliczania funkcji deszyfrującej $Decr_A$ na podstawie publicznie znanej funkcji szyfrowania $Encr_A$. Tutaj „niemożliwe" naprawdę oznacza „niewykonalne obliczeniowo", tak więc to, czego naprawdę potrzebujemy, to odpowiedni rodzaj jednokierunkowej funkcji zapadni; to znaczy funkcję Encr dla każdego użytkownika, którą łatwo obliczyć, powiedzmy w czasie wielomianu niższego rzędu, ale której funkcji odwrotnej Decr nie można obliczyć w czasie wielomianu, chyba że tajny klucz tego użytkownika jest znany. Analogia do zapadni jest oczywista: zapadnia nie może zostać aktywowana, dopóki nie jest znane istnienie lub położenie tajnej dźwigni lub przycisku. Później omówimy takie funkcje. Jeśli chodzi o podpisy, to oczywiste jest, że w przeciwieństwie do podpisu odręcznego, podpis cyfrowy, który ma być użyty w skomputeryzowanym kryptosystemie, musi być funkcją nie tylko strony podpisującej, ale także podpisanej wiadomości. W przeciwnym razie odbiorca może wprowadzić zmiany w podpisanej wiadomości przed pokazaniem jej neutralnemu sędziemu, a nawet dołączyć podpis do zupełnie innej wiadomości. Jeśli wiadomość jest poleceniem przelewu pieniężnego, odbiorca może po prostu dodać kilka kluczowych zer do sumy i stwierdzić, że nowa podpisana wiadomość jest autentyczna. Dlatego podpisy muszą być różne dla różnych wiadomości. Aby użyć jednokierunkowych funkcji pułapek do

podpisywania wiadomości, wymagamy, aby funkcje Encr i Decr były przemienne, to znaczy, że odszyfrowanie jakiejkolwiek zaszyfrowanej wiadomości powinno dać tę wiadomość w jej oryginalnej formie, ale także szyfrowanie odszyfrowanej wiadomości musi dać oryginalną wiadomość. W związku z tym wymagamy dla każdej ze stron A zarówno:

$Decr_A$ ($Encr_A$ (M)) = M i $Encr_A$ ($Decr_A$ (M)) = M

Ponieważ komunikat jest tylko liczbą, a zarówno $Encr_A$ jak i $Decr_A$ są funkcjami na liczbach, ma sens, przynajmniej matematycznie, zastosowanie $Decr_A$ do komunikatu M. Ale jaki to ma sens praktyczny? Dlaczego ktokolwiek miałby być zainteresowany zastosowaniem funkcji deszyfrowania do niezaszyfrowanej wiadomości? Odpowiedź jest prosta. Aby to podpisać! Oto jak to działa

<img src=”images/121.PNG”>

Jeśli Bob chce wysłać Alicji podpisaną wiadomość M, Bob najpierw oblicza swoją specjalną sygnaturę zależną od wiadomości S, stosując własną prywatną funkcję deszyfrującą $Decr_B$ do M. W ten sposób:

S = $Decr_B$(M)

Następnie szyfruje podpis S w zwykły sposób z kluczem publicznym, używając publicznej funkcji szyfrowania Alicji $Encr_A$ i wysyła wynik, a mianowicie $Encr_A$ ($Decr_B$ (M)), do Alicji. Po otrzymaniu tego dziwnie wyglądającego numeru, Alice najpierw odszyfrowuje go za pomocą swojej prywatnej funkcji deszyfrującej $Decr_A$ Wynikiem jest $Decr_A$ ($Encr_A$ S)), co w rzeczywistości jest $Decr_A$ ($Encr_A$($Decr_B$ (M))). Jednak ponieważ $Decr_A$ cofa wszystko, co $Encr_A$ związał, wynikiem tego będzie po prostu S lub $Decr_B$((M). (Zauważ, że Alicja nie może jeszcze odczytać wiadomości M, ani nie jest w żaden sposób przekonana, że Bob był naprawdę nadawcą.) Wreszcie, Alicja stosuje publiczną funkcję szyfrowania Boba $Encr_B$ do S, otrzymując $Encr_B$ (S) = $Encr_B$ ($Decr_B$ (M)) = M. W ten sposób Alicja widzi wiadomość M i może być całkiem pewna, że tylko Bob mógł ją wysłać. Wynika to z faktu, że funkcje są takie, że żadna liczba nie da w wyniku M po poddaniu $Encr_B$, chyba że liczba ta była dokładnie $Decr_B$ (M), i nikt poza Bobem nie mógł wytworzyć $Decr_B$(M), ponieważ funkcja deszyfrująca $Decr_B$ jest Ściśle strzeżony sekret Boba. Co więcej, Alicja nie może podpisać żadnej innej wiadomości w imieniu Boba, ponieważ podpisywanie pociąga za sobą zastosowanie tajnej funkcji Boba $Decr_B$ do nowej wiadomości. Jednak nadal istnieje możliwość, że Alicja będzie mogła wysłać tę samą wiadomość M do kogoś innego, powiedzmy Carol, ale z podpisem Boba. Powodem jest to, że podczas tego procesu Alicja jest w posiadaniu $Decr_B$ (M), który może następnie zaszyfrować za pomocą $Encr_C$, wysyłając wynik do Carol, która pomyśli, że pochodzi od Boba. Może to mieć kluczowe znaczenie w przypadku, gdy komunikat M brzmi „Ja, generał Bob, niniejszym rozkazuję ci wyruszyć na następującą niebezpieczną misję: . . ”. Aby zapobiec takiej sytuacji, imię i nazwisko odbiorcy wiadomości (i ewentualnie także data) powinny być zawsze podane, tak jak w „Ja, generał Bob, niniejszym rozkazuję ci, major Alice, abyś wyruszył z następującą niebezpieczną misją: . . ”. Takie psoty w imieniu Alice (i na koszt Carol) stałyby się wtedy niemożliwe. Oczywiście oznacza to, że Alicja nie może obliczyć funkcji $Decr_B$ bez odpowiedniego klucza, nawet dla nieco zmodyfikowanej wiadomości M, która jest bardzo zbliżona do wiadomości M, dla której ma dostęp do $Decr_B$(M). Koncepcja kryptografii klucza publicznego brzmi zatem bardzo obiecująco. Jednak aby to zadziałało, musimy znaleźć odpowiednie definicje kluczy i odpowiadające im procedury Encr i Decr, które cieszą się wszystkimi przyjemnymi właściwościami, które omówiliśmy. Innymi słowy, interesują nas jednokierunkowe funkcje zapadni, a jeśli chcemy użyć funkcji sygnatury, muszą one również spełniać właściwość wzajemnej odwrotności. Nie jest wcale jasne, czy takie funkcje istnieją. W rzeczywistości można by argumentować, że wymagania są paradoksalne, niemal wewnętrznie sprzeczne. Gdzie znajdziemy funkcję jednokierunkową z naprawdę trudną do obliczenia odwrotnością? Na przykład wzięcie pierwiastka kwadratowego nie jest o wiele trudniejsze niż jego odwrotność, podniesienie do kwadratu, a poruszanie się wstecz w alfabecie jest tak proste, jak

poruszanie się do przodu. Ponadto trudny kierunek musi stać się łatwy do obliczenia, jeśli znany jest tajny klucz. Czy są takie funkcje? Zobaczymy teraz, że są, ale trudność obliczenia odwrotności bez klucza będzie polegać na domniemanej, a nie udowodnionej niewykonalności. Pytanie, czy istnieją takie funkcje z niewykonalnymi do udowodnienia odwrotnościami, jest nadal otwarte.

Kryptosystem RSA

Około rok po pojawieniu się koncepcji kryptosystemów z kluczem publicznym, znaleziono pierwszą metodę jej wdrożenia. Powstały system, nazwany Kryposystem RSA, od inicjałów jego wynalazców, jest opisany tutaj. Od tego czasu zaproponowano kilka innych definicji, z których niektóre w konsekwencji okazały się nie być bezpieczne. Podejście RSA pozostaje jednak jednym z najciekawszych z nich wszystkich i, jak wyjaśniono później, istnieją powody, by sądzić, że jest naprawdę nie do złamania. Ważne jest, aby zrozumieć, co to znaczy, że kryptosystem klucza publicznego został złamany lub scrackowany. Ponieważ integralność systemu klucza publicznego zależy od trudności obliczenia funkcji Decr bez odpowiednich kluczy, złamanie takiego systemu wiąże się ze znalezieniem szybkiego algorytmu obliczania funkcji Decr przy znajomości odpowiedniej funkcji Encr. Jeśli używane są podpisy, można posiadać również wiedzę o kilku przykładowych wiadomościach M i ich szyfrogramach Decr (M). Tak więc, chociaż złamanie pewnych konwencjonalnych metod kryptograficznych może wymagać szczęśliwych zgadnięć lub wyrafinowanych sposobów na znalezienie jakiegoś tajnego kodu lub liczby, złamanie kryptosystemów z kluczem publicznym jest tak naprawdę równoznaczne ze znalezieniem sprytnych algorytmów czasu wielomianowego dla pewnych problemów, które uważa się za możliwe. nieodłączne zachowanie superwielomianu w czasie. I to jest praca algorytmiczna par excellence. System RSA opiera się na kontraście między testowaniem pierwszości a faktoringiem. To pierwsze, jak widzieliśmy, można przeprowadzić bardzo szybko, przy użyciu algorytmu probabilistyki , a być może w przyszłości również przy użyciu szybkiej wersji nowego nieprobabilistycznego algorytmu wielomianowego. Jednak dla tych ostatnich nie ma znanych szybkich metod, nawet probabilistycznych, a faktoring jest faktycznie przypuszczalny, że nie jest nawet w probabilistycznej/randomizowanej klasie RP. Każda ze stron, powiedzmy Alice, potajemnie i losowo, wybiera dwie duże liczby pierwsze P i Q, o długości powiedzmy około 300 cyfr, i mnoży je, w wyniku czego iloczyn N = P × Q. Alicja zachowuje w tajemnicy liczby pierwsze, ale upublicznia ich iloczyn (oraz inną ilość, jak wyjaśniono później). czynniki pierwsze w rozsądnym czasie. Oto szczegóły.

Przed wybraniem dwóch liczb pierwszych Alicja musi wybrać inną liczbę, G, zwaną wykładnikiem publicznym. Powinna to być liczba nieparzysta, najlepiej pierwsza i nie musi być zbyt duża. Ta liczba może być taka sama dla wszystkich uczestników; ulubionym wyborem jest pierwszy 2¹⁶ + 1 = 65 537. Wybierając liczby pierwsze P i Q, Alicja musi upewnić się, że ani P − 1, ani Q − 1 nie mają żadnych wspólnych czynników z G, z wyjątkiem oczywiście trywialnego czynnika 1. Na koniec Alicja oblicza swój prywatny wykładnik K, być multiplikatywną odwrotnością G modulo (P - 1) × (Q - 1), co oznacza, że K × G daje resztę 1 po podzieleniu przez (P - 1) × (Q - 1). Symbolicznie:

$K \times G \equiv 1 \pmod{(P-1) \times (Q-1)}$

To kończy proces wyjścia Alice i kupowania kłódki i klucza. Kłódką jest para G, N , która jest upubliczniana, a tajnym kluczem jest K. Czynniki pierwsze N, a mianowicie P i Q, również są utrzymywane w tajemnicy. Aby być całkiem precyzyjnym, powinniśmy wskazać, że są to wszystkie liczby Alicji, zapisując je jako P_A, Q_A, N_A, K_A i GA. Inne strony wybierają własne numery P_B, P_C , Q_B, Q_C itd. Jak wyglądają procedury szyfrowania i deszyfrowania? Załóżmy, że Bob chce wysłać wiadomość M do Alicji. Aby go zaszyfrować, używa publicznej pary Alice (G_A, N_A) . Bob najpierw dzieli M na bloki liczb, każdy z przedziału od 0 do $N_A - 1$. Dalej możemy założyć, że istnieje tylko jedna taka liczba M, ponieważ

cały proces jest przeprowadzany dla każdej z nich. Aby uzyskać tekst zaszyfrowany H, Bob podnosi M do potęgi G_A , modulo N_A:

$H = Encr_A(M) = M^{Ga} \ (mod\ N_A)$

Oznacza to, że H jest resztą uzyskaną po podzieleniu M^{Ga} przez N_A. To kończy definicję procedury szyfrowania. Zauważ, że ponieważ cała arytmetyka jest wykonywana modulo N_A ,wszystkie zaangażowane liczby mieszczą się w zakresie od 0 do $N_A - 1$, więc zarówno wiadomość, jak i jej zaszyfrowany tekst znajdują się w tym samym zakresie liczb. Deszyfrowanie jest bardzo podobne: Alicja po otrzymaniu zaszyfrowanego tekstu H podnosi go do mocy swojego tajnego klucza K_A, również modulo N_A. Zatem:

$Decr_A\ (H) = H^{Ka} \ (mod\ N_A)$

Pochodzenie terminów wykładnik publiczny i wykładnik prywatny powinno być teraz jasne. Teraz łatwo zauważyć, że:

$Decr_z(Encr_A(M)) = Encr_A\ (Decr_z(M)) \equiv M^{Ga} \times K_A \ (mod\ N_A)$

Nie jest to takie łatwe do zauważenia, ale mimo wszystko prawdą jest, że w szczególny sposób K_A zostało wyprowadzone z G_A, P i Q, ta ostatnia liczba to po prostu M (mod K_A N_A), więc odszyfrowanie zaszyfrowanej wiadomości daje wiadomość w swojej pierwotnej formie. Fakt, że nie tylko $Decr_A$ ($Encr_A$ (M)) daje M, ale także $Encr_A$ ($Decr_A$ (M)) i że jest to prawdą dla każdego M, sprawia, że metoda RSA pasuje również do sygnatur. Oczywiście musimy pokazać, w jaki sposób wszystkie obliczenia mogą być rzeczywiście wykonane efektywnie i że $Decr_A$ (M) nie można obliczyć bez znajomości klucza K_A Alicji. Cały proces konfiguracji zaczyna się od tego, że każda ze stron wybiera dwie duże liczby pierwsze, co można osiągnąć bezboleśnie przy użyciu szybkiego algorytmu testowania pierwszości, wielokrotnie dla losowych 300-cyfrowych liczb, jak wyjaśniono wcześniej. Oczywiście szansa, że dwie strony wymyślą dokładnie te same liczby pierwsze, jest znikoma. Ostatnim krokiem przygotowania jest obliczenie K z G, P i Q (pomijamy tutaj indeks A dla jasności). To, jak również rzeczywiste obliczenia MG i HK modulo N, można przeprowadzić dość szybko przy użyciu stosunkowo prostych procedur dla potęgowania modulo N i dla pewnej wersji algorytmu największego wspólnego dzielnika (gcd). Pominięto szczegóły, ponieważ mają one nieco techniczny charakter. Jeśli chodzi o bezpieczeństwo systemu RSA, można wykazać, że jeśli rozłożymy duże liczby w rozsądnym czasie, system zostanie natychmiast zepsuty, ponieważ wtedy przeciwnik może wziąć produkt publiczny N, znaleźć jego czynniki (tj. dwie tajne liczby pierwsze P i Q) i użyj ich, wraz z publicznym wykładnikiem G, do obliczenia prywatnego wykładnika K. Podwójnie, wszystkie podejścia sugerowane do tej pory w celu złamania systemu RSA wykazały, że aby zadziałać, muszą dać szybkie rozwiązania problemu faktoringowego. Innymi słowy, na chwilę obecną dla każdego podejścia sugerowanego jako atak na bezpieczeństwo systemu RSA wykazano, że albo to nie zadziała, albo że jeśli w zasadzie zadziała, to skutkuje szybkim algorytmem faktoring. Ponieważ jednak faktoring jest mocno przypuszczany, że nie ma szybkiego algorytmu, nawet probabilistycznego, a żaden z proponowanych ataków na system RSA nie dał takiego algorytmu, ludzie mocno wierzą, że RSA jest bezpieczny. Istnieje nieco inna wersja systemu RSA, której bezpieczeństwo jest dowodem na to, że szybki faktoring. Innymi słowy, wykazano, że każda metoda złamania tego konkretnego kryptosystemu da szybki algorytm faktoryzacji. Oczywiście, ponieważ nie znamy dokładnego statusu problemu faktoringu, nie jest jasne, czy ten kryptosystem jest lepszy czy gorszy od oryginalnego RSA. Warto ponownie podkreślić fundamentalne aspekty algorytmiki, które są zaangażowane w takie idee, jak kryptosystem RSA. Obejmują one konwencjonalne algorytmy sekwencyjne, pozornie nierozwiązywalne problemy algorytmiczne, algorytmy probabilistyczne, a jeśli metoda ma działać w miarę szybko na dużych liczbach, to albo bardzo wydajne programowanie, albo projektowanie sprzętu specjalnego przeznaczenia.

Gra w pokera przez telefon

Funkcje kryptograficzne klucza publicznego mogą być używane w wielu pozornie niepowiązanych aplikacjach. Rozważ dwie osoby, Alice i Bob, którzy chcą grać w pokera przez telefon. Można założyć, że każdy gracz ma dostęp do komputera na wypadek, gdyby w trakcie gry wykonywał obliczenia, a informacje cyfrowe mogą być przesyłane linią telefoniczną. Nie ma neutralnego sędziego, który rozda karty lub który ma ogólną wiedzę na temat rąk graczy i/lub kart pozostających w talii; wszystko musi być wykonane przez samych dwóch graczy. Nietrudno wyobrazić sobie mniej zabawne sytuacje o podobnym charakterze, takie jak cyfrowe negocjacje kontraktowe. Oczywiście każdy gracz musi mieć pewne informacje, które mogą być utrzymywane w tajemnicy podczas gry, takie jak jego ręka z kartami. Teraz, w zwykłej grze twarzą w twarz, gracz nie może twierdzić, że ma asa, chyba że jest w stanie to zademonstrować, odsłaniając asa jako jedną z posiadanych kart. Elektronicznym odpowiednikiem tego jest czekanie, aż gra się skończy, a następnie umożliwienie każdemu graczowi sprawdzenie całej sekwencji ruchów drugiego, w tym jego prywatnych działań. Zapobiegnie to zwykłym oszustwom. Oszukiwanie to jednak nie jedyny problem. Trochę namysłu pokazuje, że rozdawanie kart stanowi prawdziwe wyzwanie. Musimy zaprojektować protokół, który zaczyna się od 52 (reprezentowanych cyfrowo) kart i powoduje, że każdy z dwóch graczy ma losową rękę składającą się z pięciu kart, a pozostałe 42 pozostają w stosie do wykorzystania w przyszłości. Tym, co sprawia, że problem nie jest trywialny, jest to, że musimy upewnić się, że żaden z graczy nie zna ręki drugiego gracza. Na pierwszy rzut oka wydaje się to niemożliwe. Możemy zacząć od zaszyfrowania i przetasowania kart. Ponieważ jednak przynajmniej jeden z graczy musi być w stanie odszyfrować używane szyfrowanie, wydaje się, że jeden z nich będzie wiedział, która z kart została rozdana drugiemu lub że jeden z nich będzie w stanie zaaranżować żeby mieć szczególnie dobrą rękę. Czy możemy zalecić uczciwą i losową ofertę? Odpowiedź brzmi tak. Pomysł polega na użyciu jednokierunkowych funkcji pułapek, jak w kryptografii klucza publicznego, ale w tym przypadku wymagamy, aby były przemienne; to znaczy dla każdej wiadomości M musimy mieć:

$Encr_B(Encr_A (M)) = Encr_A (Encr_B (M))$

Gwarantuje to, że jeśli wiadomość jest zaszyfrowana przez Alicję, a następnie przez Boba, może zostać odszyfrowana najpierw przez Alicję, a następnie przez Boba, ponieważ podwójne szyfrowanie da $Encr_B (Encr_A(M))$, co jest takie samo jak $Encr_B$ $Encr_B(M))$, przy czym odszyfrowanie Alicji przyniesie $Decr_A (Encr_A (Encr_B(M)))$, co jest po prostu $Encr_B (M)$, a na koniec odszyfrowanie Boba da $Decr_B (Encr_B (M))$, który jest po prostu oryginalnym M. Można wykazać, że funkcje RSA spełniają ten dodatkowy wymóg przemienności, a zatem wydają się być odpowiednie również dla niniejszego zastosowania. Jeśli chodzi o zamknięte pudełka, przemienność oznacza, że zatrzask ma wystarczająco dużo miejsca na dwie kłódki, które można zatrzasnąć obok siebie w dowolnej kolejności, a nie jedna nad drugą. W ten sposób pierwsza kłódka do założenia nie musi być usuwana jako ostatnia. Tak jak poprzednio, każdy gracz zaczyna od wybrania własnych funkcji Encr i Decr. Jednak, w przeciwieństwie do kryptografii z kluczem publicznym, żadna informacja nie jest upubliczniana przed zakończeniem gry, nawet funkcje szyfrowania. Oto jak nasi gracze Alice i Bob rozdają sobie pięć kart z 52 talii. Alice używa własnej funkcji szyfrowania Encr do zaszyfrowania opisów 52 kart. Przypomnij sobie, że cała sekwencja gry jest później sprawdzana, gdy funkcje obu graczy są ujawnione, aby Alicja nie mogła zaszyfrować nielegalnego zestawu kart, mając, powiedzmy, 20 asów. Używając rzucania monetą lub w jakikolwiek inny sposób, Alicja tasuje zaszyfrowaną talię i wysyła wynikową listę do Boba. Mimo że dokładnie wie, jak wyglądają 52 zaszyfrowane wiadomości po odszyfrowaniu, nie ma możliwości dowiedzenia się, która karta jest która, ponieważ docierają one w nieznanej mu kolejności, a $Encr_A$ i $Decr_A$ są ściśle strzeżonymi tajemnicami Alicji. Teraz czas na samo rozdanie. Bob wybiera 10 zaszyfrowanych kart, z których pięć szyfruje po raz drugi za pomocą własnej funkcji $Encr_B$. Następnie wysyła Alice wszystkie 10. Tak więc

Bob ma teraz 42 pudełka zamknięte na kłódkę Alice, a Alice ma 10 - pięć zamkniętych na własną kłódkę i pięć zamkniętych na obie kłódki. Alice odblokowuje teraz własne 10 zamków; to znaczy, Alice odszyfrowuje 10 wiadomości otrzymanych od Boba za pomocą jej funkcji deszyfrującej Decr. Pierwsze pięć z nich staje się teraz niezaszyfrowanymi opisami kart i odtąd Alicja uważa je za swoją rękę. Pozostałe pięć jest nadal zablokowanych za pomocą $Encr_B$ (tutaj jest potrzebna przemienność - czytelnik powinien to sprawdzić), a Alicja odsyła je z powrotem do Boba. Po otrzymaniu, Bob odblokowuje je za pomocą $Decr_B$, odsłaniając w ten sposób własną rękę z pięcioma kartami. Ręce są oczywiście rozłączne i różnią się od pozostałych 42 kart. Co więcej, żaden z graczy nie wie, jaka jest ręka drugiego gracza. Bob nie wie nic o ręce Alicji, chociaż Bob był tym, który faktycznie rozdał karty, ponieważ wybrał 10 kart z potasowanej talii, która została zaszyfrowana za pomocą funkcji, której nie potrafi odszyfrować. Podobnie Alicja nie wie nic o ręce Boba, chociaż Alicja wie, jak odszyfrować oryginalne szyfrowanie, ponieważ pięć kart Boba zostało zaszyfrowanych po raz drugi (przez Boba) przy użyciu funkcji, której Alicja nie ma możliwości odszyfrowania. W ten sposób transakcja wydaje się być ważna, uczciwa i tak samo bezpieczna, jak każda metoda handlowania. Teraz nie powinieneś mieć trudności z ustaleniem, jak toczy się gra. Jedyną nietrywialną częścią jest sytuacja, w której gracz musi wziąć nową kartę z pozostałego stosu, procedura, którą można przeprowadzić przy użyciu dokładnie tego samego pomysłu, co przy rozdawaniu pierwszych rąk. Pomimo tych faktów okazuje się, że istnieją dość poważne problemy, jeśli chodzi o wdrożenie tego protokołu handlu. Wykazano na przykład, że funkcje szyfrowania RSA są tutaj niewystarczające. Chociaż wiadomości zaszyfrowanych przy ich użyciu nie można, o ile wiemy, faktycznie odszyfrować w rozsądnym czasie, pewne częściowe informacje można z nich wydobyć. W ten sposób można manipulować zaszyfrowaną wersją karty i ustalić pewne matematyczne właściwości jej oryginalnej (cyfrowej) wersji, które wynikają ze szczególnego sposobu definiowania funkcji RSA. Na przykład może być możliwe sprawdzenie, czy liczba kodująca kartę ma taką samą resztę po podzieleniu przez jakąś specjalną liczbę pierwszą, jak inna liczba kwadratowa. To trochę tak, jakby powiedzieć, że jeden gracz może określić kolor (czerwony lub czarny) kart drugiego – jest to oczywiście niedopuszczalny kompromis w większości gier karcianych. Jednym z doraźnych sposobów przezwyciężenia takich problemów jest opisanie przez Alicję kart przed zaszyfrowaniem w swoim własnym, nieformalnym języku, dokładna forma, jaką przybiera opis, jest nieznana Bobowi z wyprzedzeniem lub wstawienie do nich losowych liter i cyfr; podobny pomysł zostałby wykorzystany przez Boba. Wydaje się, że w ten sposób jeden gracz nie mógłby nic zyskać znając takie arytmetyczne właściwości oryginalnych, nieprzewidywalnych opisów kart drugiego gracza. Powinniśmy jednak zdać sobie sprawę, że takie podejście może nie być naprawdę bezpieczne, ponieważ nie możemy udowodnić że nie wyciekają żadne istotne informacje o kartach. Rzeczywiście odkryto inne, bezpieczniejsze protokoły rozdawania kart. Same są probabilistyczne, ale są znacznie bardziej skomplikowane niż opisana tutaj prosta i elegancka.

Interaktywne dowody

Wróćmy na chwilę do klasy NP. Widzieliśmy, że problemy w NP charakteryzują się możliwością rozwiązania w czasie wielomianowym za pomocą magicznej monety. Równoważnie problemy w NP to te, które dopuszczają „tak"-certyfikat wielkości wielomianu. Tę drugą charakterystykę można przeformułować w kategoriach swoistej gry między udowadniający a weryfikatorem. Alicja, dowodząca, jest wszechmocna i próbuje przekonać Boba, weryfikatora, który ma tylko deterministyczną moc wielomianu w czasie, że dane wejściowe X do problemu to dane wejściowe „tak". Dla konkretności weźmy konkretny problem w NP, powiedzmy trójkolorowalność grafu. Alicja chce przekonać Boba, że wykres G można pokolorować trzema kolorami. (Przypomnij sobie, że reguła jest taka, że żadne dwa sąsiednie węzły nie mogą być monochromatyczne.) Ponieważ Bob ma tylko potęgę wielomianu, nie może sam zweryfikować tego faktu. W związku z tym Alicja, która ma nieograniczoną moc, po prostu wysyła Bobowi trójkolorowe G. Jasne jest, że Bob, nawet ze swoją

ograniczoną mocą, może zweryfikować, czy kolorowanie jest zgodne z prawem i w ten sposób nabierze przekonania, że G rzeczywiście jest trójkolorowe. Oczywiście nie ma sposobu, aby Alicja przekonała Boba, że wykres jest trójkolorowy, jeśli tak nie jest. Zatem możemy powiedzieć, że problem decyzyjny P jest w NP, jeśli dla każdego wejścia X, ilekroć X jest wejściem „tak", Alicja może przekonać Boba o tym fakcie w czasie wielomianowym, ale jeśli X jest wejściem „nie", to ani Alicja, ani żaden inny dowódca nie może przekonać Boba, że jest inaczej. Ta mała gra jest dość prosta i wymaga tylko jednej rundy: Alicja wysyła certyfikat wielkości wielomianu do Boba, który natychmiast weryfikuje, czy rzeczywiście jest to certyfikat. Ta konfiguracja została uogólniona na kilka sposobów, prowadząc do silniejszych koncepcji dowodzenia i weryfikacji. Podstawową ideą jest przekształcenie procesu w interaktywny, z wieloma rundami, oraz umożliwienie weryfikatorowi rzucania monetami i zadawania pytań udowadniających, wszystko w czasie wielomianowym. Tak więc Alicja pozostaje wszechmocna, ale Bob ma teraz moc probabilistycznej maszyny czasu wielomianowego. Co więcej, w dobrym duchu poprzednie części, czynimy również podstawowe pojęcie dowodzenia probabilistycznego. W szczególności wymagamy tylko, aby Alicja mogła przekonać Boba o „tak" danych wejściowych do P z przytłaczająco wysokim prawdopodobieństwem, podczas gdy dowodzący może (błędnie) przekonać Boba, że dane wejściowe „nie" są w rzeczywistości tylko danymi wejściowymi „tak". z pomijalnie małym prawdopodobieństwem. To rozszerzenie prowadzi do klasy problemów znanych jako IP, oznaczające interaktywny czas wielomianowy. Warto się zatrzymać, aby ocenić znaczenie tego pojęcia. Konwencjonalna gra związana z NP jest bardzo podobna do standardowego sposobu udowadniania komuś oświadczenia na piśmie, powiedzmy w ramach publikacji matematycznej: dostarczasz to, co twierdzisz, że jest kompletnym dowodem, wykorzystując całą pomysłowość, na jaką możesz się zdobyć, i Następnie sprawdzam, czy w to wierzę, czy nie. (Wrócimy do tej wersji dowodu nieco później.) Nowe pojęcie jest potężnym, ale bardzo naturalnym rozszerzeniem, bardziej podobnym do sposobu, w jaki ludzie udowadniają sobie nawzajem oświadczenia ustnie: Dostarczacie pewnych informacji; Zadaję pytania, być może dotyczące przypadkowych elementów, o których nie wiedziałeś, że wybiorę; następnie udzielasz odpowiedzi i więcej informacji; Nadal zadręczam cię pytaniami; itd. Trwa to do czasu, aż przekonam się (w probabilistycznym sensie tego słowa, to znaczy z bardzo dużym prawdopodobieństwem), że masz rację. I oczywiście wymagamy, aby cała procedura trwała tylko rozsądnie, a mianowicie. wielomian, ilość czasu. Jaka jest dokładna siła tej kombinacji interakcji i rzucania monetą? Jakie problemy decyzyjne można rozwiązać za pomocą nowej procedury? Innymi słowy, chcielibyśmy dokładnie wiedzieć, jakie problemy występują w IP. Z poprzedniej dyskusji jasno wynika, że IP zawiera wszystkie NP, w tym oczywiście problemy NP-zupełne. Jednak wcale nie jest jasne, czy dodanie interakcji i rzucania monetami daje nam coś nowego. Rzeczywiście, nie jest łatwym zadaniem wymyślenie interaktywnego protokołu dla problemu, który nie występuje w NP. Niemniej jednak ostatnio kwestia ta została rozstrzygnięta w całej jej ogólności, z zaskakującym wynikiem, że IP to w rzeczywistości dokładnie PSPACE. Siła probabilistycznego oddziaływania wielomianowego w czasie jest więc dokładnie taka sama jak potęga przestrzeni wielomianowej! Dla każdego problemu w PSPACE „tak" można udowodnić w czasie wielomianowym za pomocą interakcji i losowości. Innymi słowy, rozszerzenie rygorystycznej, jednoprzebiegowej procedury dowodowej, aby dopuścić bardziej zrelaksowany probabilistyczny i interaktywny protokół dowodowy (wymagany do wykorzystania tylko rozsądnej ilości czasu), jest dokładnie tym samym, co całkowite porzucenie protokołów dowodowych, na korzyść konwencjonalnego obliczenia który wykorzystuje rozsądną ilość miejsca w pamięci (ale może wymagać nieuzasadnionej ilości czasu). Skorzystano również z dalszej swobody, z definicją klasy MIP, oznaczającą interaktywne dowody z wieloma dowodzeniami. W tym przypadku interaktywne dowody mogą być przeprowadzane przez więcej niż jednego dowodzącego (ale wciąż tylko jednego weryfikatora), chociaż dowodzący nie mogą się komunikować. Wykazano, że ta klasa jest identyczna z NEXPTIME, tj. klasą problemów obliczalną wykładniczo przy użyciu magicznej monety. Fakt ten jest szczególnie interesujący, ponieważ wiadomo, że NEXPTIME jest ściśle większy niż P, a nawet NP (co nie

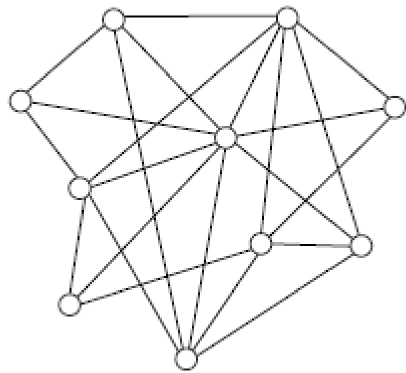
ma miejsca w przypadku PSPACE), więc dowody wielodowodzenia z normalnymi monetami są silniejsze niż dowody bezpośrednie z magicznymi monetami.

Protokoły wiedzy zerowej

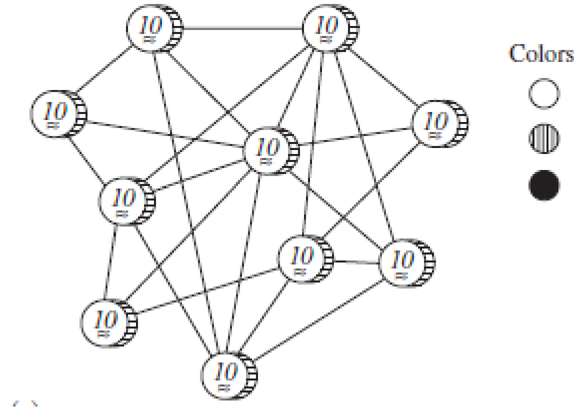
Aby zilustrować dowody interaktywne, warto podać przykład ich ekscytującego wariantu. Załóżmy, że Alicja chce przekonać Boba, że zna pewien sekret, ale nie chce, aby Bob sam poznał ten sekret. Brzmi to nieprawdopodobnie: jak przekonać kogoś, że wiesz, powiedzmy, jakiego koloru krawat ma teraz na sobie prezydent Stanów Zjednoczonych, nie ujawniając w jakiś sposób tej bezcennej informacji drugiej osobie lub osobie trzeciej? Jako kolejny przyziemny przykład rozważ pytanie Gdzie jest Waldo? książka. Każda z jego stron zawiera bardzo misterną ilustrację z wieloma różnymi postaciami, a problemem jest znalezienie Waldo, predefiniowanej, dość kolorowej postaci. Załóżmy, że Alice i Bob patrzą na jedną ze stron, a Alicja twierdzi, że wie, gdzie jest Waldo. Bob jej nie wierzy, a Alice chce go przekonać, że nie kłamie. Jednak chce to zrobić bez ujawniania mu rzeczywistej lokalizacji Waldo, aby mógł dalej próbować, wyjadając sobie serce, że znalazła Waldo, ale on nie … Jak ona może to zrobić? Jest wiele rozwiązań, a oto jedno. Alice i Bob najpierw kserują odpowiednią stronę. Bob następnie drukuje jakiś regularny, ale niemożliwy do podrobienia wzór na odwrocie kopii (np. powtarza jakieś bezsensowne słowo gęsto i z losową kierunkowością, w całym nim). Odwracając się do Boba, Alicja wycina teraz obraz Waldo z tej kopii, niszcząc resztki i triumfalnie pokazuje to Bobowi, który widzi, że to naprawdę Waldo, i widzi, że ma swój wzór na plecach. Alice nie mogła odgadnąć wzoru, więc nie mogła oszukiwać, a Bob nie może dostrzec małego wycięcia, gdzie na dużej stronie znajdowało się zdjęcie Waldo. Udowodniła mu więc, że rzeczywiście wie, gdzie jest Waldo, ale Bob niczego (lub prawie niczego) nie uczy się z tego, co widzi. Pomijając powiązania prezydentów i obrazy Waldo, chodzi o opracowanie interaktywnego protokołu probabilistycznego, zbudowanego wokół ustalonego problemu algorytmicznego P, w którym Alicja może przekonać Boba w czasie wielomianowym, że dane X jest „tak" wejściem P, ale w w taki sposób, że po zakończeniu interaktywnych wymian Bob nie wie nic o dowodach na „tak" X. Wie, z ogromnym prawdopodobieństwem, że X rzeczywiście jest „tak", a zatem Alicja miała rację, ale to jedyna nowa informacja, jaką uzyskał dzięki temu procesowi. W szczególności nie może (w czasie wielomianowym) nawet udowodnić tego samego faktu komuś innemu! Takie pozornie paradoksalne protokoły nazywane są wiedzą zerową. Zanim podamy jeden przykład, warto zauważyć, że protokoły o wiedzy zerowej mają wiele zastosowań w kryptografii i bezpiecznej komunikacji. Na przykład możemy chcieć opracować karty elektroniczne, które umożliwią pracownikom wejście do wrażliwego zakładu, ale chociaż chcemy, aby bramy otwierały się tylko dla posiadaczy legalnej karty, nie chcemy, aby personel zakładu wiedział dokładnie, kogo wpuścił. Albo załóżmy, że grupa ludzi chce założyć wspólne konto bankowe. Chcieliby mieć możliwość wypłacania i przelewania pieniędzy drogą elektroniczną oraz aby bank egzekwował pewne zasady (np. limit wypłacanych kwot dziennych). Załóżmy jednak, że chcą również uniemożliwić personelowi banku możliwość samodzielnego symulowania wypłaty, a nawet dokładnego poznania, który z nich wypłaca, tylko że pieniądze zostały wypłacone legalnie i zgodnie z zasadami. Takie przypadki wymagają umiejętności przekonania weryfikatora, że znasz jakiś sekret, klucz lub kod, ale bez ujawniania niczego poza samym tym faktem.

Trójkolorowa wiedza o zerowej wiedzy

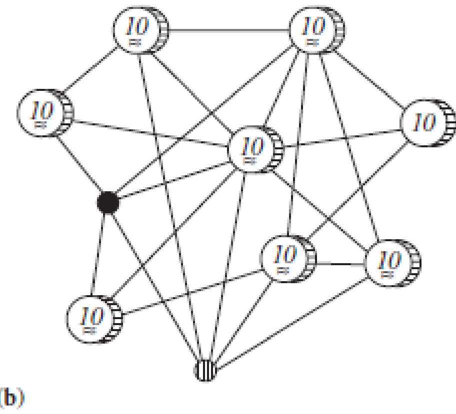
Oto protokół z wiedzą zerową. Będzie to opisane tak, jakby odbywało się między dwiema prawdziwymi ludźmi, ale przekształcenie go w pełnoprawny protokół algorytmiczny, nadający się do zastosowań elektronicznych, nie jest zbyt trudne. Opiera się na trójkolorowości. Alicja pokazuje Bobowi wykres G



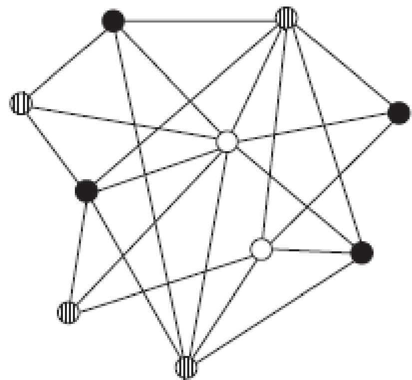
i twierdzi, że można go pokolorować trzema kolorami. Bob, w dostępnym dla niego czasie wielomianowym, nie może sam zweryfikować tego faktu, więc Alice próbuje mu to udowodnić. Zabiera wykres i potajemnie koloruje go trzema kolorami, powiedzmy czerwonym, niebieskim i żółtym. Następnie ostrożnie pokrywa kolorowe węzły małymi monetami i umieszcza wykres w widoku Boba



Mówi również Bobowi, jakich kolorów użyła. Bob jest oczywiście sceptyczny, ale mimo że jest zainteresowana wyeliminowaniem jego sceptycyzmu, Alicja nie chce ujawniać kolorystyki. W rzeczywistości nie jest zainteresowana eksponowaniem kolorowania dowolnych trzech węzłów, ponieważ mogłoby to zagrozić samej strategii kolorowania. Zamiast tego mówi, że jest gotowa odsłonić dowolną parę sąsiednich węzłów. Zatem Bob wybiera losowo krawędź wykresu, a Alicja usuwa monety z węzłów na jej końcach


(b)

Bob zauważa, że na tych dwóch węzłach są dwa różne kolory i że te dwa kolory są spośród trzech wymienionych Alicji. Teraz, oczywiście, jeśli odkryje, że odsłonięte węzły naruszają jedną z tych właściwości, to wykazał, że kolorowanie jest niezgodne z prawem, tym samym obalając twierdzenie Alicji. Jeśli te dwa kolory są różne i spośród trzech wymienionych Alicji, Bob nie może narzekać, ale nie jest też pewien, czy cały wykres jest kolorowy poprawnie. Teraz pojawia się podstęp. Zamiast zgadzać się na ujawnienie większej liczby węzłów, Alicja odzyskuje wykres i zmienia jego kolor, tym razem używając, powiedzmy, brązowego, czarnego i białego, i ponownie mówi mu, jakich kolorów użyła, zakrywa węzły i pokazuje Bobowi wykres. On ponownie wybiera przewagę, a Alice szybko odkrywa dwa węzły na jego końcach. Znowu Bob widzi dwa różne kolory z trzech, których używała. Ta procedura jest powtarzana jeszcze kilka razy, aż Bob będzie szczęśliwy. Czy Bob powinien być naprawdę szczęśliwy? Cóż, spójrzmy na rzeczy z jego punktu widzenia. Powiedzmy, że G zawiera N krawędzi (N może być co najwyżej kwadratem liczby węzłów). Po tym, jak Alicja zdała pierwszy test, tj. Bob był zadowolony z kolorów, które widział w pierwszej rundzie, oczywiście nie jest pewien, czy Alicja może pokolorować cały wykres, ale wie, że miał szansę 1 na N na złapanie ją, jeśli nie, ponieważ on może wybrać dowolną krawędź, a Alicja nie miała pojęcia, który by wybrać, kiedy pokoloruje i zakryje węzły. Stąd prawdopodobieństwo, że zda pierwszy test, nie wiedząc, jak pokolorować G, wynosi 1 − 1/N, czyli faktycznie (N − 1)/N. Teraz drugi test był całkowicie niezależny od pierwszego, z punktu widzenia Alicji, więc prawdopodobieństwo, że zda ona dwa pierwsze testy, nie wiedząc, jak pokolorować wykres, wynosi ((N − 1)/N)². A w miarę postępu procesu prawdopodobieństwo maleje przez coraz większe potęgi (N − 1)/N, a tym samym szybko zbliża się do 0 wraz ze wzrostem liczby udanych testów. Bob może zatem zatrzymać proces, gdy jest przekonany, że to prawdopodobieństwo jest pomijalnie niskie. Będzie wtedy całkowicie przekonany, że Alicja może trójkolorować wykres. (W rzeczywistości może)



W praktyce wystarcza stosunkowo mała liczba rund, nawet w przypadku dużych wykresów. A co z wiedzą Boba? Cóż, ponieważ wciąż widzi różne kolory, a Alicja nie wskazuje zgodności między zestawami kolorów w różnych testach, Bob nie ma wiedzy o związkach kolorów między dowolnymi trzema węzłami, a jego izolowana wiedza na temat wielu par węzłów jest nic nie pomoże, ponieważ kolory są zawsze inne. Argument ten można sformalizować, aby pokazać, że Bob ma zerową wiedzę o schemacie kolorowania i że w czasie wielomianowym nie może znaleźć niczego, czego nie mógłby się dowiedzieć bez informacji, które otrzymuje od Alice (chyba że P = NP). W szczególności, jak wspomniano powyżej, nie może nawet komuś udowodnić, że G może być trójkolorowe, chociaż sam jest w pełni przekonany. Jest to zatem protokół wiedzy zerowej par excellence i po raz kolejny mamy niezwykłą aplikację, która w zasadniczy sposób wykorzystuje zarówno dobre, jak i złe wiadomości, jakie ma do zaoferowania algorytmika.

Dowody sprawdzalne probabilistycznie

Wiemy więc o dowodach interaktywnych, w których weryfikator nabiera przekonania o poprawności tego, co dowodzi w sensie probabilistycznym. Wiemy też o robieniu tego bez podawania weryfikatorowi jakichkolwiek nowych informacji. Powracamy teraz do nieinteraktywnego pojęcia dowodu, w którym dowód jest przedstawiany na piśmie: nadal mamy weryfikatora, Boba, ale zamiast dowodzącej, że Alicja wchodzi z nim w interakcję, istnieje statyczny, niezmienny dokument dowodowy, który przygotowała w naprzód i dał mu. Do tego rodzaju certyfikatu nawiązywaliśmy, omawiając NP w kategoriach certyfikatów „tak”. Tak więc dane wejściowe, które próbujemy ustalić jako dane wejściowe „tak”, mają rozmiar N, a długość dokumentu dowodowego jest wielomianem w N. Subtelność różnicy polega na tym, że stały certyfikat dowodowy nie może kłamać. W interaktywnym modelu dowodzenia/weryfikatora Alicja może zmienić dowód w trakcie lotu, że tak powiem, dając różne odpowiedzi na sondy Boba, aby jego praca była bardziej trudna. Natomiast stały, pisemny dowód się nie zmienia. Jak wspomniano wcześniej, różnica między tymi dwoma konfiguracjami polega na różnicy między osobą próbującą przekazać dowód jakiegoś matematycznego twierdzenia podczas sesji interaktywnej, powiedzmy przed tablicą, a próbą przekonania się do dowodu pisemnego tego oświadczenia, powiedzmy w opublikowanej gazecie. Tak więc gra nie polega teraz na tym, że Alicja próbuje zmienić Boba w wierzącego, ale Bob sprawdza jej dowód i sam decyduje, czy w to wierzyć, czy nie. Co robi Bob? Czyta niektóre części dowodu, dowolne części, które chce, i myśli, jak tylko chce, potem jeszcze trochę czyta i myśli, a na koniec musi być przekonany, że dowód jest poprawny. Jak zwykle, dzień musi kończyć się w czasie wielomianowym, a przekonanie jest w sensie probabilistycznym, tj. Bob musi mieć pewność, że dowód jest poprawny, przy znikomym prawdopodobieństwie błędu. W prawdziwym duchu tego rozdziału i poprzedniego pozwalamy również, aby myślenie Boba było probabilistyczne; to znaczy, że może przeprowadzić obliczenie

probabilistyczne w czasie wielomianowym z rzucaniem monetą, aby pomóc mu zdecydować, co chce zrobić dalej. Aby zakończyć sformalizowanie tego pojęcia dowodu, musimy powiedzieć, co mamy na myśli, pozwalając Bobowi przeczytać dowolną część dowodu, jaką chce. Najbardziej naturalnym sposobem modelowania tego rodzaju działalności jest postrzeganie dowodu jako sekwencji bitów o długości wielomianu i umożliwienie Bobowi zbadania go w poszukiwaniu informacji, kiedy tylko ma na to ochotę. To ostatnie zdanie oznacza, że z punktu widzenia kogokolwiek innego niż sam Bob, sondy są losowe, ponieważ nie mamy możliwości wcześniejszego ustalenia, które części dowodu poprosi o obejrzenie. Ten model dowodu nazywa się PCP, co oznacza dowody sprawdzalne probabilistycznie. Aby ocenić trudność sprawdzania dowodów w ten sposób, mierzymy dwie wielkości, obie jako funkcje N (rozmiar wejściowy): liczbę rzutów monetą, której Bob potrzebuje w swoim myśleniu, oraz liczbę bitów z dowodu, że musi sondować. Drugi z nich jest szczególnie ważny, ponieważ pokazuje, jak dużą część prawdopodobnie długiego dowodu musi nawet przejrzeć Bob, aby przekonać się o jego poprawności. Nie jest zbyt trudne wykazanie, że NEXPTIME, o którym wiemy, że jest równoważny zbiorowi MIP problemów dających się udowodnić w czasie wielomianowym przy użyciu interakcji wielostronnej, jest w rzeczywistości również zbiorem problemów dopuszczających probabilistycznie sprawdzalne dowody, w których Bob może rzucić dowolną liczbę monet i poproś o wyświetlenie dowolnej liczby bitów z dowodu. Niezwykły wynik dotyczący PCP jest taki: wszystkie problemy z NP można sprawdzić za pomocą stałej liczby prób dowodu! Powiedzmy to ostrożniej. Dla każdego problemu w NP, w tym najtrudniejszych w nim – problemów NP-zupełnych – „tak" można udowodnić za pomocą certyfikatów dowodowych, dla których weryfikator musi jedynie sprawdzić część o ustalonym rozmiarze, której długość nie jest powiązana z konkretny przypadek, do którego próbuje się przekonać. Ma to dość zdumiewające konsekwencje , że jeśli chcesz zadowolić się przekonaniem z przytłaczającym prawdopodobieństwem, większość twierdzeń matematycznych można udowodnić za pomocą dowodów o rozsądnej wielkości, które można sprawdzić, czytając tylko bardzo małą, losową część dowodu o stałych rozmiarach

Badania nad kryptografią

Tematyka tej Części stanowi jeden z „najgorętszych" obszarów badawczych w informatyce i jak zawsze jest również atrakcyjnym obszarem pracy w bardziej tajnych placówkach, takich jak wywiady i inne, mniej legalne. lub akceptowalne sfery. Kryptografia, podpisy cyfrowe, handel elektroniczny i „elektroniczna gotówka", skomputeryzowane podpisywanie umów oraz bezpieczeństwo i odporność na awarie dużych systemów to tylko niektóre z modnych słów napędzających takie badania. Jak wyjaśniono wcześniej, poza wysoce aplikacyjnym charakterem takiej pracy, jej atrakcyjność wynika z faktu, że wykorzystuje ona nie tylko pomysły prowadzące do skutecznych rozwiązań pewnych problemów algorytmicznych, ale także negatywne skutki dotyczące (pozornej) trudności w rozwiązaniu innych. Interesujące jest to, że w praktycznie wszystkich tych zastosowaniach negatywne wyniki są oparte na głębokich przypuszczeniach dotyczących dolnych granic nie-wielomianowych dla pewnych problemów. Intensywnie badane są również dowody interakcji i wiedzy zerowej, a jednym z najbardziej aktywnych obszarów pracy są dowody sprawdzalne probabilistycznie. Powodem jest to, że takie dowody mają ścisły związek z zagadnieniami dotyczącymi trudności aproksymacji problemów NP-zupełnych, które zostały omówione w Części 7; w rzeczywistości niektóre wyniki PCP prowadzą do wyników nieprzybliżalnych.

Inżynieria oprogramowania

Słynna anegdota opowiada o próbie zorientowania się, czym jest programowanie komputerowe, grupie dyrektorów w firmie zatrudniającej programistów. W ciągu tygodnia nauczono ich programować i dano im do rozwiązania mały problem. Każdemu kierownikowi przydzielono profesjonalnego programistę jako asystenta. Po pomyślnym rozwiązaniu przydzielonych im problemów z „niewielką" pomocą, kadra kierownicza wyszła z tego doświadczenia z poczuciem, że programowanie jest mimo wszystko dość łatwe i nie ma powodu, dla którego ich programiści nie mogliby ukończyć swoich zadań na tak jak oni sami. Oczywiście jest duża różnica między dużymi projektami programistycznymi, obejmującymi miliony lub nawet więcej linijek kodu, a małymi ćwiczeniami programistycznymi, składającymi się z kilkudziesięciu linijek. Tak duża różnica ilościowa powoduje jakościową różnicę w złożoności zadania i wymaga zupełnie innego rodzaju zarządzania. Jedna osoba może z łatwością śledzić wszystkie szczegóły drobnego problemu w swojej głowie. W miarę narastania problemu staje się to trudniejsze i pojawia się potrzeba pisemnych zapisów celów poszczególnych elementów składających się na projekt oraz relacji między nimi. Bez takiej dokumentacji łatwo jest poczynić założenia, w jaki sposób komponent ma być używany podczas jego programowania, ale z naruszeniem tych założeń podczas programowania innych komponentów, które go wykorzystują. Może się to już zdarzyć, gdy program składa się z kilkuset wierszy. Gdy projekty stają się większe, wyrastają poza możliwości jednego programisty. Duże projekty wymagają zespołu programistów współpracujących ze sobą, a nawet kilku zespołów, z których każdy zajmuje się inną częścią całego projektu. Bardzo duże projekty, takie jak nowoczesne systemy operacyjne, składają się z dziesiątek milionów linijek kodu i zatrudniają setki programistów, a nawet więcej. Części całego projektu mogą być opracowywane przez różne firmy. To sprawia, że ukryte założenia i inne rodzaje błędów są nieuniknione. Niestety, nawet zastosowanie najlepszych dostępnych obecnie narzędzi i metod nie gwarantuje, że programy będą wolne od błędów, o czym wie każdy, kto używał komputera od dłuższego czasu. W tej Części omówimy ogólne problemy, które pojawiają się podczas projektowania dużych systemów oprogramowania oraz główne procesy i metodologie stosowane w ich rozwiązywaniu.

Ukryte założenia w lotach kosmicznych

Istnieje niestety wiele przykładów ukrytych założeń w tworzeniu oprogramowania prowadzących do czasami katastrofalnych awarii. Poniższe przykłady to znane wydarzenia w historii lotów kosmicznych. NASA wystrzeliła Mars Climate Orbiter w 1998 roku. Jej misją było okrążenie Marsa i złożenie raportu o jego warunkach pogodowych w ramach przygotowań do lądowania na Marsie w 1999 roku. Orbiter dotarł do Marsa 23 września 1999 roku, ale potem zaginął. Komisja badawcza ustaliła, że trajektoria orbitera była o około 170 kilometrów za niska, ponieważ jedna część programu kontroli naziemnej orbitera wykorzystywała jednostki angielskie, podczas gdy inne części oczekiwały danych w jednostkach metrycznych. Stało się tak pomimo istnienia jasnej specyfikacji, która wskazywała właściwe jednostki do użycia, a także najnowocześniejszych metodologii rozwoju stosowanych przez NASA. Wśród czynników sprzyjających wymienionych w raporcie komisji dochodzeniowej była niewystarczająca komunikacja między zespołami ds. rozwoju i operacji. Mars Polar Lander, który miał wylądować na Marsie około sześć tygodni później, również zaginął. Komisja kontrolna określiła awarię oprogramowania jako najbardziej prawdopodobną przyczynę tej utraty. Silniki lądownika muszą zostać wyłączone, gdy tylko wyląduje, w przeciwnym razie przewróci się. Czujniki na nogach lądownika Polar Lander generują sygnał przy kontakcie z powierzchnią. Czujniki mogą czasami generować fałszywe sygnały, a oprogramowanie jest zaprogramowane tak, aby ignorować takie sygnały, porównując dwa kolejne sygnały i działając na nie tylko wtedy, gdy oba wykazują tę samą wartość. Jednakże, gdy nogi lądownika zostaną wysunięte z pozycji złożonej do pozycji do lądowania, czujniki mogą generować

dłuższe sygnały „kontaktowe”. To samo w sobie nie stanowi problemu, ponieważ nogi są rozmieszczone na wysokości około 1500 metrów, podczas gdy oprogramowanie nie wyłączy silników, dopóki radar nie zgłosi, że lądownik znajduje się mniej niż 40 metrów nad powierzchnią. Niestety fałszywy sygnał wykryty podczas uruchamiania nie jest kasowany i powoduje wyłączenie silnika, gdy tylko lądownik osiągnie wysokość 40 metrów. To wystarczy, aby rozbić się na powierzchni. Raport stwierdza: To zachowanie zostało zrozumiane i oprogramowanie lotu musiało zignorować te zdarzenia; jednak wymaganie nie opisywało tych zdarzeń konkretnie, a co za tym idzie, projektanci oprogramowania nie uwzględnili ich właściwie. Innym spektakularnym przykładem ukrytych przypuszczeń jest eksplozja, która miała miejsce podczas dziewiczego lotu rakiety Ariane 5 Europejskiej Agencji Kosmicznej w dniu 4 czerwca 1996 roku. Około 40 sekund po wystrzeleniu rakieta dokonała gwałtownej zmiany toru lotu, a w rezultacie rozpadł się i eksplodował. Komisja śledcza wyśledziła awarię z powodu błędu oprogramowania określonego typu. Zastosowany język programowania umożliwił programowi naprawienie się po takich błędach, a cztery możliwe przypadki z siedmiu były rzeczywiście odpowiednio chronione. Powód, dla którego pozostałe trzy przypadki nie były chronione, nie został udokumentowany w kodzie, ale później został zidentyfikowany w analizie, która wykazała, że tego rodzaju błąd nie może wystąpić w tych przypadkach. Jak się okazało, ta analiza rzeczywiście była poprawna dla wcześniejszych modeli Ariane, dla których napisano oprogramowanie. Kiedy to oprogramowanie zostało ponownie użyte w Ariane 5, która ma inne charakterystyki toru lotu, nie powiodło się. Nie wchodząc tutaj w więcej szczegółów, należy zauważyć, że chociaż była to główna usterka, w połączeniu z szeregiem innych aspektów systemu spowodowała katastrofalną awarię.

Problem z oprogramowaniem

Te przykłady wyraźnie pokazują potrzebę zdyscyplinowanej metody pisania programów, aby zapewnić ich niezawodność. Badanie takich metod nazywa się inżynierią oprogramowania. Obejmuje aspekty techniczne, takie jak języki programowania i narzędzia do zadań takich jak testowanie, debugowanie i weryfikacja, a także praktyki zarządzania. Inżynieria oprogramowania różni się od innych dyscyplin inżynierskich z powodu odmiennego charakteru przedmiotu: algorytmów i programów. Mają one charakter dyskretny; to znaczy zajmują się pojedynczymi i odrębnymi bytami, a mianowicie bitami. Natomiast inne dyscypliny inżynierskie zajmują się zjawiskami fizycznymi, które zwykle mają charakter ciągły. Ważną miarą złożoności systemu jest liczba jakościowo różnych stanów, jakie może on mieć. Na przykład, podczas gdy samochód może poruszać się z nieskończoną liczbą prędkości od zera do, powiedzmy, 150 kilometrów na godzinę, w celu sterowania samochodem za pomocą manualnej skrzyni biegów istnieje tylko siedem różnych stanów: od trzech do pięciu przełożenia do przodu, położenie neutralne i wsteczny. W każdym z tych stanów prędkość samochodu jest prostą funkcją prędkości obrotowej silnika. Chociaż zmienne układu ciągłego mogą przyjmować nieskończenie wiele wartości, zwykle mają stosunkowo niewielką liczbę jakościowo różnych stanów. Systemy dyskretne, a zwłaszcza komputery, mają ogromną liczbę stanów. Jeśli dodasz jeden bit do pamięci komputera, pomnożysz liczbę stanów przez dwa, ponieważ każdy stary stan dzieli się na dwa nowe: jeden, w którym wartość dodatkowego bitu wynosi zero, a drugi, w którym jest to jeden. Liczba możliwych stanów dla komputera z pamięcią zaledwie 280 bitów przekracza całkowitą szacowaną liczbę atomów we wszechświecie! Współczesne komputery mają pamięć zawierającą miliardy bitów, z niewyobrażalną (ale mimo to skończoną) liczbą możliwych stanów. Złożoność systemów komputerowych jest zatem znacznie wyższa niż systemów ciągłych, co czyni je mniej przewidywalnymi. W wyniku tej różnicy systemy ciągłe są bardziej podatne na analizę matematyczną i umożliwiają inżynierom poleganie na współczynnikach bezpieczeństwa. Na przykład podczas projektowania mostu dla określonego obciążenia, projekt zawsze zakłada większe obciążenie. Przy współczynniku bezpieczeństwa wynoszącym trzy most powinien teoretycznie wytrzymać trzykrotność wymaganego obciążenia. Będzie to wymagało mocniejszej konstrukcji, która może nawet zrekompensować niektóre wady

konstrukcyjne. Jednak błąd jednobitowy w programie komputerowym może spowodować, że całkowicie zejdzie z toru, powodując katastrofalną awarię. Ten efekt powiększenia systemów dyskretnych sprawia, że pojęcie współczynników bezpieczeństwa dla oprogramowania jest bezsensowne, a tym samym usuwa jedno z najpotężniejszych narzędzi inżynierskich ze sfery inżynierii oprogramowania. Metody analizy systemów dyskretnych również pozostają w tyle za metodami dla systemów ciągłych. Na przykład, istnieje obszerna wiedza związana z weryfikacją programów komputerowych, jak opisano w Części 5. Badania te doprowadziły nawet do kilku projektów weryfikacyjnych, które zakończyły się sukcesem komercyjnym. Wymagają one jednak dużych nakładów pracy ze strony wysoko wykwalifikowanych badaczy, a formalna weryfikacja większości pisanego oprogramowania jest obecnie niemożliwa. Takie wysiłki skupiają się zatem głównie na rdzeniach systemów krytycznych dla bezpieczeństwa. Co do reszty, musimy zadowolić się mniej skutecznymi, ale bardziej praktycznymi metodami. Oprogramowanie jest znacznie bardziej elastycznym medium niż fizyczne materiały używane w innych dyscyplinach inżynierskich. Po zbudowaniu mostu jego wymiana będzie miała bardzo istotny powód. Podobnie, kupując komputer, spodziewasz się, że będzie on używany przez kilka lat, zanim wymienisz go na nowy. Ponieważ jednak programy komputerowe są najwyraźniej tak łatwe do zmiany - wymaga to jedynie zmiany zawartości dysku twardego komputera - klienci często oczekują (i wiele razy dostaje) częste zamienniki dla ich oprogramowania (zwykle nazywane „aktualizacjami”). Drugą stroną tej monety jest to, że producenci oprogramowania nie martwią się o jakość swojego początkowego produktu w taki sam sposób, jak robią to producenci sprzętu. Zamiast tego polegają na aktualizacjach, aby zapewnić rozwiązania problemów wykrytych po wprowadzeniu produktu na rynek. To niewiele robi, aby stworzyć poczucie zaufania w branży oprogramowania. Co gorsza, założenie, że oprogramowanie jest łatwe do modyfikacji, jest błędne. Ogromna złożoność systemów oprogramowania z jednej strony, a rozmiar problemów, które takie systemy mają rozwiązać z drugiej, oznaczają, że programy są bardzo złożone, znacznie przekraczające nasze możliwości analityczne. Zazwyczaj więc próba naprawienia jednego błędu może spowodować powstanie kilku innych. Metodologie inżynierii oprogramowania, takie jak te omówione w tym rozdziale, mogą być wykorzystane do złagodzenia tego problemu. Jednak wiążą się one z własnym kosztem, który należy uwzględnić w całkowitym koszcie modyfikacji. Ostatnim znanym przykładem jest błąd Y2K. U schyłku dwudziestego wieku działało dużo oprogramowania, które zostało zbudowane przy założeniu, że lata można przedstawić za pomocą dwóch cyfr. Na przykład liczba „80” reprezentuje rok 1980; logicznie rzecz biorąc, liczba „01” oznaczałaby rok 1901. Chociaż w 1901 nie było żadnych komputerów elektronicznych, nadal konieczne byłoby odwołanie się do tej daty; na przykład może to być data urodzenia osoby, która w 1980 r. była uprawniona do świadczeń z zabezpieczenia społecznego. Jednak, gdy zbliżał się rok 2000, stało się oczywiste, że konieczne będzie reprezentowanie lat takich jak 2001, ale liczba „01” została już zajęta! (Pierwsze rzeczywiste przypadki tego problemu miały miejsce w 1998 roku, kiedy niektóre karty kredytowe, których data ważności wynosiła 2000, zostały odrzucone przez komputery myśląc, że był to 1900). W czasie pisania programów ten wybór reprezentacji był rozsądny. Niektóre z tych programów zostały napisane w latach 60. i nikt nie spodziewał się, że będą nadal działać ponad 30 lat później. W tamtych czasach rozmiary pamięci i dysków były znacznie mniejsze i droższe niż obecnie, a przechowywanie nadmiarowych cyfr „19” w każdym polu daty byłyby zbyt kosztowne. (Co zaskakujące, niektóre programy napisane na początku lat 90. nadal używały reprezentacji dwucyfrowej, chociaż tak naprawdę nie było na to usprawiedliwienia tak blisko 2000 roku.) Jedna z rzeczy, które wiemy o tym problemie, to to, że nie można go rozwiązać przez komputer w zasadzie . To nadal nie wyklucza częściowego rozwiązania, które działa w wielu, jeśli nie w większości praktycznych przypadków, ale nawet tak wiele nie jest dziś łatwo dostępnych. takich programów było bardzo dużo, ponieważ daty pojawiają się niemal wszędzie. Aby podać tylko jeden przykład, w programach kontrolnych niektórych wind pojawiają się daty, ponieważ muszą one śledzić harmonogram konserwacji. Niestety, nieubłagany bieg czasu zamienił te programy,

które były poprawne w momencie napisania, w błędne. Spowodowało to przepychankę w celu rozwiązania problemu w połowie i pod koniec lat 90., co okazało się ogromnym przedsięwzięciem. W programach rozpowszechniło się założenie, że lata są przedstawiane za pomocą dwóch cyfr. Konieczne było zbadanie każdego wiersza kodu, aby ustalić, czy w jakikolwiek sposób manipuluje datami. Jeśli tak, to musiało to zostać skorygowane w sposób zgodny z nową strategią dat. Ten projekt, ogólnie bardzo udany, pochłonął dużą inwestycję czasu i pieniędzy oraz opóźnił inne plany rozwoju oprogramowania. Jak widzieliśmy w Części 8, problem z weryfikacją oprogramowania jest nierozstrzygnięty i dlatego nie ma prawdziwej nadziei, że ktoś kiedyś napisze program, który będzie w stanie wykryć wszystkie wystąpienia podobnego błędu. Automatyczne poprawianie błędów jest również nieobliczalne. Chociaż możliwe jest opracowywanie narzędzi, które pomagają ludziom wykrywać i poprawiać błędy (co rzeczywiście zostało zrobione przez tak zwane fabryki Y2K pod koniec lat 90.), procesu nigdy nie da się w pełni zautomatyzować.

Modułowość i interfejsy

Kupując nowy samochód, otrzymujesz instrukcję obsługi, która wyjaśnia przyrządy i sterowanie, gdzie znajdują się bezpieczniki, jak zmienić opony i inne szczegóły techniczne. W części dotyczącej sterowania światłami znajdziesz wyjaśnienia dotyczące obsługi reflektorów, kierunkowskazów, świateł postojowych itd., ale nie instrukcje, które mówią, że należy użyć kierunkowskazu, zanim zechcesz skręcić w prawo, który bezpiecznik sprawdza, czy światła się nie włączają lub które kraje wymagają jazdy z włączonymi reflektorami w ciągu dnia. Są to ważne kwestie, które musisz znać, aby móc obsługiwać swój samochód; jednak jak wiele innych kwestii związanych w taki czy inny sposób z oświetleniem samochodu, pojawiają się one w innym miejscu w instrukcji obsługi lub w zupełnie innych dokumentach. Może fajnie jest mieć wszystkie istotne szczegóły w jednym miejscu, ale jest ich po prostu za dużo. Aby móc szybko i dokładnie uzyskać dostęp do tych informacji, należy je podzielić na osobne tematy, z których każdy pojawia się w innej sekcji jakiegoś dokumentu. A podział musi być na tyle logiczny, abyśmy mogli dowiedzieć się, gdzie znaleźć wszystko, czego potrzebujemy. Ta sama zasada dotyczy programów komputerowych. Często pomijanym aspektem programów jest to, że są one pisane dla ludzi bardziej niż dla komputerów. Na pierwszy rzut oka wydaje się to absurdalne: programy komputerowe są oczywiście napisane do wykonywania na komputerach, a kompilator nie dba o to, jak wyraźnie jest napisany program. Jednak programiści muszą czytać i rozumieć te programy podczas ich pisania i modyfikowania. Ponieważ modyfikacja oprogramowania jest tak powszechna, każdy udany program będzie musiał zostać zmodyfikowany w trakcie jego życia, a programiści odpowiedzialni za modyfikacje niekoniecznie będą tymi, którzy napisali program w pierwszej kolejności. Nawet oryginalny programista zazwyczaj nie pamięta szczegółów programu po kilku miesiącach. Dlatego organizacja programu według rozdziałów i wersetów jest tak samo ważna dla programów komputerowych, jak dla instrukcji obsługi samochodu, i z tych samych powodów: aby ludzie mogli znaleźć potrzebne informacje. Programy komputerowe należy podzielić na małe i spójne części, zwane modułami, z których każdy może być rozumiany oddzielnie. Jest to szczególnie ważne kiedy każdy moduł musi być rozwijany przez osobny zespół programistów. Jeśli jednak istnieje wiele powiązań między modułami, nie można ich zrozumieć w odosobnieniu, a zespoły je rozwijające nie mogą działać osobno, ponieważ każda decyzja jednego zespołu może mieć wpływ na wiele innych zespołów, a cały proces grzęźnie. Dlatego moduły muszą być względnie niezależne; ten cel nazywa się modułowością. Oczywiście moduły nie mogą być całkowicie niezależne, ponieważ niektórzy (klienci) muszą korzystać z usług dostarczanych przez inne moduły (dostawców). Jednak modułowość oznacza, że powinna istnieć minimalna ilość informacji, które muszą być dzielone między klientem a moduły dostawców. Informacje te nazywają się interfejsem modułu dostawcy i powinny zawierać wszystkie szczegóły, które klienci muszą wiedzieć o usługach oferowanych przez dostawcę, ale nic o tym, jak te usługi wykonuje. Na przykład interfejs modułu, który implementuje kolejki składa się z metod

używanych do wykonywania różnych operacji kolejki (dodawanie i usuwanie elementów, pobieranie elementu frontowego itp.), ale nie z implementacji tych metod. Interfejsy powinny być jasno udokumentowane, najlepiej w sposób formalny, który pozwala kompilatorowi sprawdzić, czy programiści nie tworzą niepożądanych zależności. Cel ten nazywany jest ukrywaniem informacji, co oznacza, że klient nie może polegać na implementacji modułu dostawcy, a jedynie na jego interfejsie. To umożliwia podejście do budowania dużych systemów w stylu Lego, w którym moduły można zastąpić innymi implementacjami tego samego interfejsu. Różne typy języków programowania mają różne sposoby dzielenia programu na moduły. Pojęcie modułu może być tak proste, jak zbiór funkcji zapisanych w jednym pliku. Może być bardziej wyrafinowany, a niektóre języki wyraźnie rozróżniają interfejs od implementacji modułu. Jednak język, który nie obsługuje modułowości w taki czy inny sposób, jest po prostu bezużyteczny w przypadku dużych projektów. Na przykład w językach obiektowych naturalną jednostką modułową są klasy. Klasy abstrakcyjne bez implementacji są czystymi interfejsami (i rzeczywiście tak się nazywają w JAVA). Przykładem jest klasa Queue z Części 3. Jak tam wspomniano, interfejs ten może być zaimplementowany przez takie klasy jak LinkedQueue, a programista modułu klienta nie musi być świadomy, która konkretna implementacja jest używana. Kompilator może łatwo sprawdzić, czy rzeczywiście używana jest odpowiednia implementacja. Interfejsy JAVA zawierają nazwy i typy parametrów wszystkich metod, które klienci mogą wywoływać na obiektach kolejki. Niestety to nie wszystkie informacje, które musi znać programista modułu klienta; brakuje znaczenia tych metod, które odróżnia np. stosy od kolejek. Świadczą o tym stwierdzenia dotyczące metodologii projektowania według umowy, opisanej w Części 5. Interfejs z umową zawiera pełne informacje potrzebne klientom. Istnieje wiele metodologii, które próbują poprowadzić proces rozbicia dużego problemu na stosunkowo niezależne moduły. Nie będziemy tutaj wchodzić w szczegóły tych metodologii. Dość powiedzieć, że to zadanie jest nadal bardziej sztuką niż nauką i wymaga dużego doświadczenia.

Modele cyklu życia

Aby opracować metody zarządzania inżynierią oprogramowania, najpierw konieczne jest zrozumienie procesu tworzenia oprogramowania. Proces ten okazuje się dość złożony i trudny do sformalizowania. Istnieją różne jego modele, zwane modelami cyklu życia; każdy model ma inny pogląd na proces iw rezultacie obsługuje różne metodologie. Wszystkie modele cyklu życia oparte są na następujących rodzajach czynności, które zilustrujemy na przykładzie edytora tekstu.

Pozyskiwanie wymagań. (Nazywana również analizą wymagań.) Jest to proces odkrywania tego, czego klient naprawdę potrzebuje. Klienci mogą posiadać wiedzę na temat własnej domeny biznesowej, ale często nie rozumieją implikacji algorytmicznych i systemowych wprowadzenia systemu komputerowego do swojej firmy. Może to prowadzić do nieporozumień, które muszą być dokładnie wyjaśnione, aby doprowadzić do pomyślnego wyniku. Efektem tej czynności jest zwykle dokument wymagań. Czasami nie ma klienta, z którym można by pracować, jak w przypadku, gdy firma produkuje oprogramowanie, które ma być sprzedawane z półki. W tym przypadku producent musi przewidywać potrzeby klienta, co sprawia, że pozyskiwanie wymagań jest trudniejsze, a nie łatwiejsze! (I rzeczywiście, wiele produktów kończy się niską sprzedażą, ponieważ ten proces nie przewidział tego, czego klienci naprawdę potrzebują). W naszym przykładzie dotyczącym przetwarzania tekstu dokument wymagań będzie zawierał listę konkretnych potrzeb klienta, takich jak wykorzystanie złożonej notacji matematycznej, sprawdzanie pisowni i automatyczne generowanie etykiet adresowych.

Projekt. W tym procesie problem jest dzielony na moduły, a odpowiednie algorytmy i struktury danych są wybierane do reprezentowania domeny problemu i wykonywania wymaganych operacji. Czynność ta wymaga gruntownego zrozumienia algorytmiki w celu dokonania wyborów, które zapewnią

poprawność i wydajność powstałego produktu. Projekt przykładowego edytora tekstu może określać moduły do takich zadań, jak graficzny interfejs użytkownika, obsługa tekstu, skład matematyczny i skład liter (w tym generowanie etykiet adresowych). Projekt modułu obsługi tekstu określałby strukturę danych do przechowywania zawartości tekstu; może to być na przykład lista znaków ze specjalnymi symbolami oznaczającymi łamanie linii lub lista linii, z których każdy jest wektorem zawierającym znaki pojawiające się w linii. Moduł ten określi również algorytmy służące do wykonywania takich czynności jak wyrównywanie wierszy i sprawdzanie pisowni. Realizacja. Jest to proces tłumaczenia projektu na określony język programowania. Po pierwsze, każdy moduł jest implementowany osobno. Następnie moduły są składane razem, tworząc kompletny system; proces ten nazywa się integracją i często jest to etap, na którym ujawniają się ukryte założenia. Nasz edytor tekstu jest teraz wrzucony do określonego języka programowania i można z nim poeksperymentować (lub z jego częściami) i zobaczyć, co naprawdę robi.

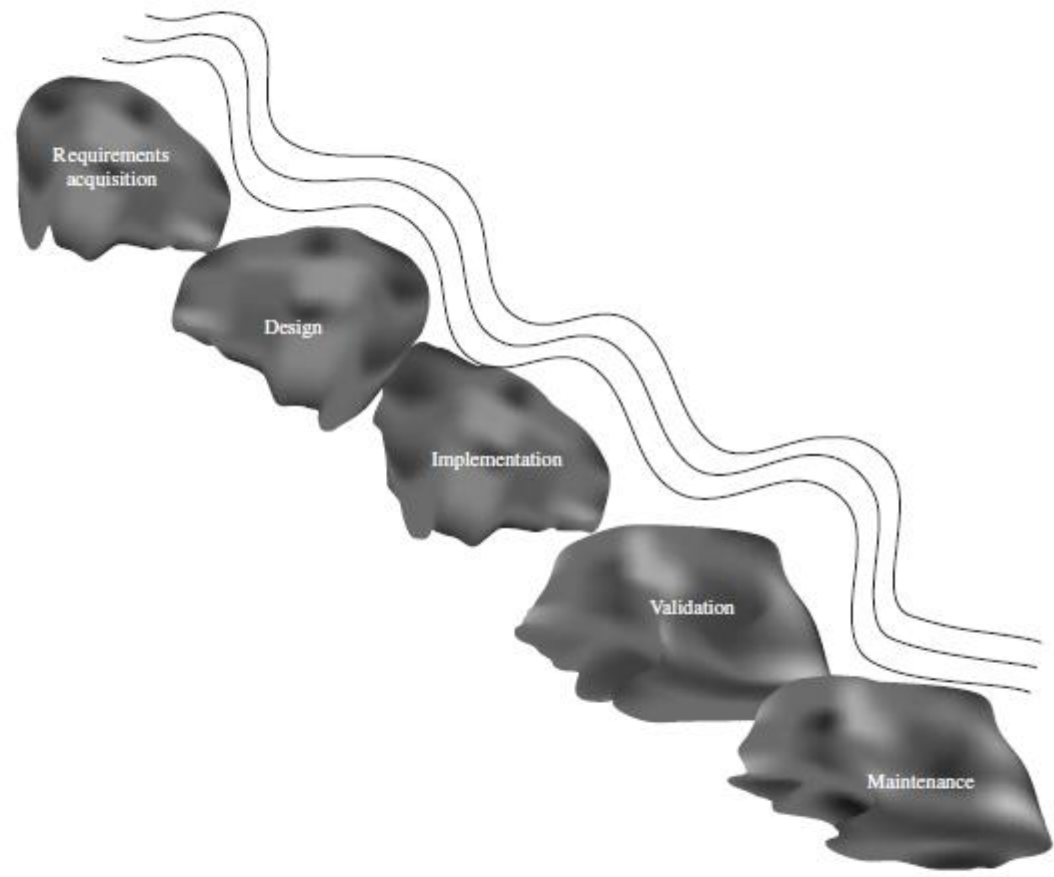
Walidacja. Jest to proces upewniania się, że każdy moduł, a także cały system, spełniają swoje specyfikacje. Walidacja może przybierać różne formy. Zwykle obejmuje obszerne testowanie poszczególnych modułów (tzw. testowanie jednostkowe) oraz całego systemu (tzw. testowanie integracyjne). Niestety, jak wspomniano w rozdziale 5, testowanie nie wystarcza do znalezienia wszystkich błędów. Formalna weryfikacja może zapewnić, że system spełnia swoją specyfikację, ale, jak wspomniano wcześniej, nie jest jeszcze praktyczna do ogólnego użytku. Również fakt, że system spełnia specyfikację, nie musi oznaczać, że robi to, czego chciał klient! Weryfikacja wymaga formalnej specyfikacji w celu sprawdzenia programu, ale przełożenie wymagań na formalną specyfikację jest trudnym problemem i tak samo podatnym na błędy jak programowanie. Nie ma niezawodnego sposobu na sprawdzenie specyfikacji pod kątem wymagań klienta, ponieważ są one w mózgu klienta i są niedostępne dla naszych formalnych narzędzi. W dobrze zaprojektowanym procesie każdy moduł aplikacji do edycji tekstu może być testowany osobno. Wymaga to napisania specjalnych programów testowych, ponieważ moduły nie są zaprojektowane do samodzielnej pracy. Jednak ta „dodatkowa" praca jest warta wysiłku, ponieważ znacznie łatwiej jest znaleźć i naprawić problemy w jednym module niż w całym systemie.

Utrzymanie. To słowo jest tutaj użyte w szczególnym technicznym sensie. W przeciwieństwie do systemów mechanicznych, programy komputerowe nie podlegają procesom środowiskowym, które mogą spowodować ich przegrzanie, zużycie lub w inny sposób utratę swoich właściwości operacyjnych. Nie trzeba „przerabiać" programu po 200 000 mil lub 10 latach (w zależności od tego, co nastąpi wcześniej) - będzie on nadal działał w nieskończoność dokładnie tak samo, jak pierwszego dnia. Na tym właśnie polega problem! Aby programy pozostały użyteczne, muszą się zmieniać. Zmiany mogą być wymagane z powodu zmian w środowisku operacyjnym programu; na przykład, gdy klient zmienia komputer typu mainframe na komputer osobisty lub podczas uaktualniania systemu operacyjnego. Środowisko operacyjne obejmuje znacznie więcej niż sam komputer: na przykład program może wymagać modyfikacji, gdy zmienią się przepisy stanowe regulujące zachowanie domeny aplikacji, co może się zdarzyć w przypadku oprogramowania kasjera, gdy zostanie wprowadzony nowy rodzaj podatku. Zmiany mogą być również wynikiem nowych próśb klientów; gdy klienci zaczynają korzystać z nowego systemu, odkrywają więcej rzeczy, które chcieliby, aby program dla nich zrobił. Wreszcie, niestety, mogą być wymagane modyfikacje, aby naprawić błędy, których dostawca nie wykrył przed udostępnieniem programu klientom. Nowe żądania klientów pojawiające się po wdrożeniu systemu są prawie zawsze nieuniknione, ponieważ wprowadzenie systemu skomputeryzowanego do środowiska, w którym wcześniej go nie było, zmienia to środowisko. To, co początkowo mogło być programem oszczędzającym pracę, wykonującym zadanie, które w innym przypadku można by wykonać ręcznie, teraz staje się istotną częścią biznesu. Na przykład księgowy może zacząć korzystać z komputera dla wygody. Odkrycie, że przyspiesza to jego pracę, może skutkować podjęciem decyzji o pozyskaniu

nowych klientów. Teraz jednak księgowy jest zależny od komputera i nie może się bez niego obejść. Na przykład, jeśli nietypowe zadania, których oprogramowanie nie było w stanie obsłużyć, były wcześniej wykonywane ręcznie, w przypadku większej liczby klientów również muszą one zostać zautomatyzowane. Statystyki pokazują, że największa inwestycja w oprogramowanie jest na etapie utrzymania; może to osiągnąć 80% całości! Dlatego dobrym pomysłem jest włożenie dużego wysiłku w inne czynności wymienione powyżej, aby ułatwić przyszłą konserwację. Większość metodologii inżynierii oprogramowania opiera się na tej przesłance. Jeśli poczta zmieni się z pięciocyfrowych kodów pocztowych na dziewięciocyfrowe, pospieszymy do dostawcy naszej aplikacji do przetwarzania tekstu, aby zmodyfikować generator etykiet adresowych tak, aby obsługiwał dziewięciocyfrowe kody. Jeśli pierwotny projekt traktował pięć jako „magiczną liczbę” bez zastanowienia się nad możliwością, że może się ona zmienić, jest całkiem prawdopodobne, że to założenie jest osadzone w wielu częściach programu. Modyfikacja będzie wtedy bardzo trudna, a dostawca pobierze za to wysoką opłatę. W takim przypadku możemy lepiej wydać nasze pieniądze, przechodząc do innego dostawcy. W rzeczywistości zmiana na dziewięciocyfrowe kody pocztowe to prawdziwy przykład, który spowodował duże koszty dla tych dostawców, którzy nie byli na to przygotowani. Jest to jednak przyćmione przez omówiony już przykład problemu Y2K.

Model wodospadu

Najbardziej podstawowy model cyklu życia nazywa się modelem kaskadowym. Zakłada, że powyższe czynności następują po sobie w ścisłej kolejności. Nazwa tego modelu wywodzi się z widoku następujących po sobie czynności, takich jak woda spadająca z klifu w szeregu kroków.



Z tego punktu widzenia dokument wymagań musi być w pełni przygotowany, zanim będzie możliwe rozpoczęcie projektowania. Ponadto po rozpoczęciu projektowania dokument wymagań jest

zamrożony i nie można go modyfikować. Powodem tego jest to, że wszelkie zmiany wymagań poczynione po zakończeniu projektowania lub, co gorsza, po wdrożeniu, wymagają cofnięcia się do początku i odpowiedniej modyfikacji projektu (i wdrożenia). Może to być niezwykle kosztowne. Rzeczywiście, modyfikacje wczesnych etapów procesu wymagają zmian w kolejnych etapach. Późniejsze etapy dodają dużo szczegółów, przez co ich zmiana jest bardziej skomplikowana. W rezultacie im dalej w procesie dokonuje się zmiana, tym jest ona bardziej kosztowna. Oszacowano, że koszt naprawy błędu w fazie wymagań wzrasta dziesięciokrotnie, jeśli jego wykrycie jest opóźnione do fazy implementacji, oraz o czynnik 100, jeśli zostanie wykryty, gdy system już działa. Niestety, na wcześniejszych etapach, w których dokonujemy najważniejszych i najbardziej wpływowych wyborów, problem rozumiemy dość niejasno. W miarę postępów w projektowaniu i wdrażaniu nasze zrozumienie problemu rośnie, aż do momentu, w którym naprawdę go rozumiemy, nie mamy żadnych znaczących wyborów do dokonania! W praktyce na wczesnych etapach nieuchronnie podejmowane będą błędne decyzje, które będą musiały zostać poprawione później. Oznacza to, że przed rozpoczęciem wdrożenia należy włożyć wiele wysiłku w walidację wymagań i projektu. Jest to dość trudne i nie ma zbyt wielu satysfakcjonujących formalnych narzędzi pomocnych w tych zadaniach. Często mamy do czynienia z mniej precyzyjnymi, ale wciąż użytecznymi metodami, takimi jak przeglądy projektów, gdzie wymagania i projekt są przedstawiane najbardziej doświadczonym osobom, które starają się wykryć ewentualne wady. Ponieważ koszt wprowadzenia zmian w wymaganiach i projekcie jest tak wysoki, należy je z wyprzedzeniem przewidzieć. Przy pewnym doświadczeniu można to zrobić całkiem dobrze, ale wyraźnie trzeba będzie wprowadzić zmiany. W związku z tym model kaskadowy jest zwykle pokazywany z „przepływem wstecznym", który dopuszcza tę rzeczywistość, ale metodologie oparte na modelu kaskadowym oferują niewielkie wsparcie, gdy tak się dzieje. Zaletą modelu kaskadowego jest to, że sprawia, że proces jest widoczny, a tym samym łatwiejszy w zarządzaniu. Proces jest zorientowany na dokumenty, a każdy etap modelu ma jasną definicję, jakie produkty (często nazywane produktami dostarczanymi) mają być dostępne po jego zakończeniu. Produkty te, którymi mogą być dokumenty lub programy, są oczywistymi punktami kontroli całego procesu. Kierownictwo (a także klient) może wykorzystać rezultaty do śledzenia postępów projektu. Ta mocna strona modelu jest również słabością, ponieważ wysiłek poświęcony na udokumentowanie procesu jest czasem lepiej poświęcony na zapewnienie, że zaowocuje satysfakcjonującym produktem. Ponadto, gdy zmieniają się wymagania, praktycznie niemożliwe jest rozpoczęcie procesu od zera. W rezultacie programiści „fałszują", próbując zmodyfikować istniejące dokumenty, aby wyglądały tak, jakby nowe wymagania były tam od samego początku. Chociaż jest to przydatne (i zostało nawet usankcjonowane przez niektórych wpływowych metodologów), z konieczności jest podatne na błędy. W miarę wprowadzania kolejnych zmian struktura staje się pełna luk, których łatki nie do końca wypełniają. W końcu każdy dodatek stwarza więcej problemów niż naprawia, w którym to momencie cały proces wymyka się spod kontroli.

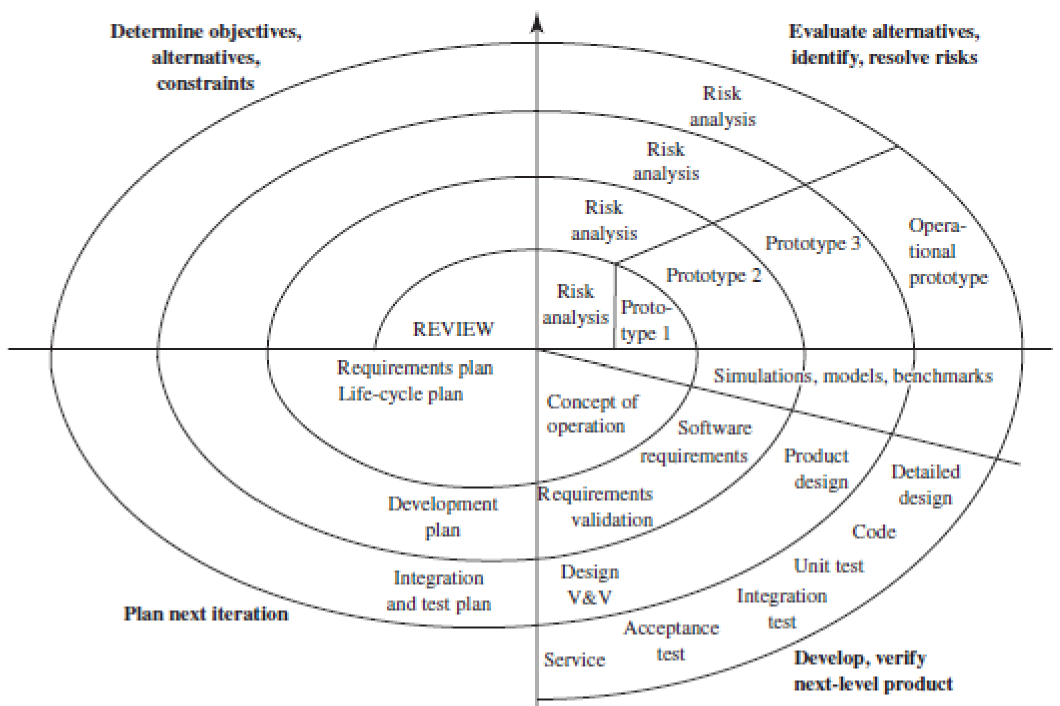
Rozwój ewolucyjny

Model kaskadowy z pewnością ma właściwą kolejność czynności; program napisany bez dobrego projektu będzie wadliwy, a projekt wykonany bez dobrego zrozumienia wymagań nie doprowadzi do rozwiązania problemu klienta. Jednak ocena produktu końcowego jest możliwa dopiero pod koniec tego długiego procesu, a jak już powiedzieliśmy, naprawa wczesnych błędów jest bardzo kosztowna. Ewolucyjny model rozwoju stara się zmniejszyć to ryzyko, zalecając jak najszybsze wyprodukowanie początkowej wersji roboczej. Klient może przekazać przydatne informacje zwrotne na temat przydatności produktu, faktycznie z niego korzystając. Jest to znacznie bardziej prawdopodobne, aby odkryć ukryte założenia i inne problemy niż jakakolwiek ilość czytania dokumentów. Informacje zwrotne z tego są następnie wykorzystywane do tworzenia drugiej wersji, która prowadzi nawet do trzeciej. Każda wersja jest tworzona zgodnie z czynnościami określonymi przez model kaskadowy, ale

szybciej i przy znacznie mniejszej dokumentacji. W konsekwencji powstałe systemy mogą być mniej dobrze skonstruowane, a zatem bardziej kosztowne w utrzymaniu. Model najlepiej zastosować, gdy ryzyko wczesnych błędów jest szczególnie wysokie, na przykład przy tworzeniu nowego typu aplikacji, z którą programista (a być może nawet klient) nie ma doświadczenia. Aby jak najszybciej dostać się do działającej wersji, jest on zwykle produkowany jako system barebone, koncentrując się na najważniejszych funkcjach i pomijając resztę. Taka wersja nazywana jest prototypem. Kolejne wersje rozwijają pierwotny prototyp, dodając więcej funkcji, aż do uzyskania w pełni funkcjonalnego systemu końcowego. Proces ten nazywa się programowaniem eksploracyjnym. Bardziej radykalnym podejściem jest wyrzucenie pierwotnego prototypu i rozpoczęcie od zera po raz drugi. Takie podejście, zwane szybkim prototypowaniem, ma tę zaletę, że początkowy prototyp można stworzyć bardzo szybko, ponieważ jego wewnętrzna jakość nie jest istotna. Wyciągając wnioski z początkowego doświadczenia, programiści mogą teraz zbudować drugi prototyp wyższej jakości, nie martwiąc się o doposażenie starego projektu w nowe spostrzeżenia. Co więcej, szybki prototyp można napisać w innym języku programowania niż drugi. Niektóre języki, takie jak LISP, doskonale nadają się do szybkiego prototypowania, ponieważ uwalniają programistę od niektórych bardziej nudnych aspektów innych języków, takich jak konieczność deklarowania typów wszystkich zmiennych. Wynikająca z tego szybkość rozwoju rekompensuje ryzyko związane z ich ignorowaniem. Drugi prototyp można wtedy napisać ostrożniej w bardziej restrykcyjnym języku. Jeśli programiści dobrze rozumieją dziedzinę problemu, rozwój ewolucyjny może nie być konieczny, ponieważ prawdopodobnie mogliby stworzyć szczegółową specyfikację bez eksperymentowania. Miałoby to miejsce na przykład, gdyby doświadczony zespół programistów pracował nad nowym produktem, podobnym do tych, które opracował wcześniej. Szybkie prototypowanie jest szczególnie przydatne, gdy problem do rozwiązania wymaga szczególnie innowacyjnych pomysłów lub po prostu nie jest dostatecznie dobrze zrozumiany, jak np. sztuczna inteligencja.

Model spiralny

Spiralny model cyklu życia jest uogólnieniem poprzednich.



Opiera się na procesie iteracyjnym, dzięki czemu umożliwia rozwój ewolucyjny, a także szybkie prototypowanie, gdzie każdy cykl może być oparty na modelu kaskadowym. Głównym wkładem modelu spiralnego jest skupienie się na zarządzaniu ryzykiem. Każda iteracja na spirali rozpoczyna się od fazy, w której określane są cele iteracji wraz z alternatywnymi sposobami ich osiągnięcia (u góry po lewej na rysunku). Po tym następuje faza oceny ryzyka, w której badane są zagrożenia związane z każdą alternatywą (prawy górny róg rysunku). Na tym etapie można na przykład ocenić wybór platform sprzętowych, języka programowania i narzędzi programowych. W przypadku nowej technologii, gdzie ryzyko jest szczególnie wysokie, może być konieczne przeprowadzenie oceny technologii na małą skalę w celu prawidłowej oceny i zmniejszenia związanego z nią ryzyka. Po wybraniu alternatywy jest ona realizowana i oceniana w trzeciej fazie (prawy dolny róg rysunku). Ta ocena jest następnie wykorzystywana do planowania następnej iteracji (na dole po lewej). Sam model spiralny nie określa szczegółów każdej iteracji, która mogłaby podążać za modelem kaskadowym lub w inny sposób przydzielać czynności. Podobnie jak model kaskadowy, model spiralny jest przede wszystkim ukierunkowany na sterowanie procesem i dlatego jest również oparty na dokumentach. W rzeczywistości ma dodatkowe rezultaty, w wyniku czego z dodatkowych trzech faz dodanych w celu zarządzania ryzykiem. Jest to uważane za niewielką cenę do zapłacenia w projektach wysokiego ryzyka.

Rozwój obiektowy

Jak widzieliśmy w Części 3, języki programowania obiektowego wyrosły z potrzeby modelowania zdarzeń w świecie rzeczywistym. Jest zatem naturalne, że koncepcje klas i obiektów są szczególnie odpowiednie do pozyskiwania wymagań i projektowania. Etapy obiektowej inżynierii oprogramowania są podobne do etapów innych metodologii przedstawionych powyżej. Jednak metodologie zorientowane obiektowo mają ważną zaletę polegającą na stosowaniu wspólnego słownictwa w ramach działań, opartego na pojęciach klas i obiektów. Np. program do zarządzania sklepem w sieci supermarketów jest w naturalny sposób określony pod kątem obiektów oznaczających klientów, kasjerów, terminale kasowe, artykuły spożywcze, magazyny, dostawców itp. Faza projektowania może dodawać inne rodzaje obiektów dla wewnętrznych celów algorytmicznych, ale jest to raczej rozwinięcie specyfikacji niż zupełnie inny rodzaj dokumentu. We wcześniejszych metodologiach przejście między czynnościami wiąże się z przechodzeniem między różnymi formalizmami. Ta potrzeba zmiany reprezentacji jest jednym ze słabych punktów metodologii niezorientowanych obiektowo, ponieważ prawdopodobnie zerwie połączenie między powiązanymi działaniami. Na przykład, często zdarza się, że programiści ignorują opracowane dla nich projekty, ponieważ są ukształtowani w formalizmie, który nie jest bezpośrednio związany z używanym przez nich językiem programowania. Powszechne słownictwo w metodologiach zorientowanych obiektowo sprawia, że jest to znacznie mniej prawdopodobne. Pozwala również na pewne mieszanie działań; na przykład projekt może być przeplatany programowaniem. W ten sposób doświadczenie zdobyte przy wdrażaniu jednej części systemu może wyjaśnić kwestie przydatne przy projektowaniu innych części. Dopóki proces jest dobrze zarządzany, może to przyspieszyć rozwój, zapewniając bardziej terminowe informacje zwrotne bez negatywnego wpływu na jakość. Jednym z ważnych celów inżynierii oprogramowania zawsze było ponowne wykorzystanie: możliwość korzystania z artefaktów (programów, projektów, a nawet wymagań) generowanych dla jednego projektu w kolejnym projekcie. Zaleta ponownego wykorzystania jest oczywista: pozwala deweloperowi szybko tworzyć nowe projekty po początkowej inwestycji w pierwszy tego typu projekt. Technologia obiektowa oferuje techniczne środki umożliwiające ponowne wykorzystanie. Ponieważ klasy hermetyzują typy danych wraz ze skojarzonymi z nimi operacjami, są one naturalną jednostką do ponownego użycia. Jest to jedna z głównych korzyści rozwoju obiektowego.

Ekstremalne programowanie

Podstawową zasadą modelu wodospadu jest to, że koszt zmian dramatycznie wzrasta z czasem, tak że naprawa błędów na początku procesu tworzenia staje się coraz bardziej kosztowna w miarę przechodzenia z etapu na etap. W rezultacie konieczne jest jak największe przewidywanie przyszłych zmian i uwzględnienie ich w projekcie. Może to sprawić, że projekt będzie bardziej złożony i nieporęczny niż to konieczne. Niedawno opracowano technikę zwaną refaktoryzacją, która ma pomóc w rozwiązaniu tego problemu. Zamiast zamrażać projekt na początku wdrożenia, ten pogląd przyjmuje odwrotne podejście: projekt powinien być elastyczny i zmieniać się wraz z wdrożeniem. Nie jest to łatwe. Zwykle zdarza się, że implementacja jest modyfikowana w odpowiedzi na presję klientów, zmieniające się środowiska i wykryte błędy, ale odbywa się to za pomocą „łat", które próbują zmodyfikować jakiś aspekt programu bez ogólnego widoku. W rezultacie program odbiega od pierwotnego projektu i coraz trudniej go modyfikować. Metodologia refaktoryzacji stara się zachować jakość projektu pomimo zmian w kodzie. Programowanie zmienia się między modyfikacjami, które dodają nowe funkcje lub naprawiają błędy, a refaktoryzacją, co poprawia strukturę programu bez wpływu na jego funkcjonalność. Jest to podobne do tego, co dzieje się, gdy musisz dokonać modyfikacji w swoim domu. Na przykład, kiedy dodajesz nowe gniazdko elektryczne, najpierw otwierasz ścianę, aby uzyskać dostęp do okablowania. Po zainstalowaniu nowych przewodów i gniazdka, zakrywasz otwór i odmalowujesz ścianę. Te ostatnie działania nie mają nic wspólnego ze sposobem funkcjonowania nowego gniazdka; dostarczy prąd, nawet jeśli nie zakryjesz dziury. Wykonuje się je w celu przywrócenia muru do pierwotnego stanu. Jeden z ekspertów w dziedzinie technologii obiektowej porównał sprzątanie projektu do spłacania długów, podczas gdy modyfikowanie programu podczas psucia projektu jest jak zaciągnięcie pożyczki. Niespłacona pożyczka nalicza odsetki, a jeśli nie jest spłacana przez długi czas, może przekroczyć zdolność kredytobiorcy do spłaty. Podobnie, jeśli program zbytnio odbiega od projektu, staje się niemożliwy do zarządzania. W końcu dalsze modyfikacje staną się niemożliwe, ponieważ każda taka próba przyniesie więcej problemów niż rozwiąże. Czasami warto pożyczyć pieniądze, ale dług należy spłacić jak najszybciej. Podobnie, czasami przydatne jest wprowadzenie szybkich zmian w programie, aby wykorzystać jakąś okazję. Jednak program powinien zostać zrefaktoryzowany tak szybko, jak to możliwe, aby ponownie dopasować go do dobrego projektu. Refaktoryzacja oferuje zestaw szczegółowych metod ulepszania projektu programu bez wpływu na jego funkcjonalność. W ten sposób nowe spostrzeżenia dotyczące struktury programu są wykorzystywane do ulepszania jego projektu. Takie spostrzeżenia mogą być . Istnieje również ryzyko ponownego użycia, jak pokazuje historia eksplozji Ariane 5 wspomniana na początku . Niestety są one znacznie mniej oczywiste. zdobyte podczas procesu dodawania nowej funkcjonalności, znajdowania i poprawiania błędów, czy podczas przeglądów kodu. Dzięki refaktoryzacji możliwe jest korygowanie błędów projektowych (lub w inny sposób modyfikowanie początkowego projektu) bez ponoszenia dużej kary nawet na znacznie późniejszym etapie procesu. Może to nie być skuteczne w bardzo dużych projektach, ale działa całkiem dobrze w małych i średnich projektach. Ponieważ błędy projektowe nie są tak zaporowe, początkowy projekt może być prosty, co upraszcza również późniejsze etapy. Wszystko to prowadzi do zupełnie innego stylu programowania, zwanego programowaniem ekstremalnym. Ekstremalna metodologia programowania wysoko ceni prostotę. Nazywa się to „ekstremalnym", ponieważ wykorzystuje każdą dobrą praktykę do maksimum. Na przykład zaleca częste i automatyczne testowanie (co jest również podstawową przesłanką do refaktoryzacji). Rozwój ewolucyjny jest również doprowadzony do logicznego zakończenia na wielu poziomach. Po pierwsze, integracja systemów odbywa się często (przynajmniej raz dziennie). Oznacza to, że cały czas dostępny jest w pełni działający system. Chociaż taka przejściowa wersja systemu może nie spełniać wszystkich wymagań, może być wykorzystana do oceny aktualnego stanu rozwoju w najbardziej bezpośredni sposób – poprzez faktyczne jego wykorzystanie. Po drugie, rozwój opiera się na krótkich cyklach (około

trzech tygodni), z których każdy skompresuje pozyskiwanie wymagań, projektowanie, implementację i walidację w jedną ciągłą czynność. Finalnie wydania klientów są dokonywane co kilka miesięcy, co oznacza, że klient musi być zaangażowany w projekt przez cały czas, a nie tylko na początkowym etapie. Rzeczywiście, przedstawiciel klienta na miejscu jest podstawowym wymogiem dla ekstremalnego programowania. W programowaniu ekstremalnym przeglądy kodu wykonywane są w sposób ciągły poprzez praktykę programowania w parach, w której każdą czynność programistyczną wykonują dwie osoby. W dowolnym momencie jedna z par będzie używać komputera do pisania lub modyfikowania programu, podczas gdy druga „spogląda przez ramię", aby spróbować zidentyfikować problemy, od błędów w pisowni po błędy logiczne. Okresowo „kierowca" i „nawigator" zamieniają się rolami. Mogłoby się wydawać, że dwie osoby wykonujące pracę jednej będą o połowę mniej produktywne. Jednak badania wykazały, że programowanie w parach jest prawie tak samo wydajne, jak dwie osoby pracujące oddzielnie pod względem ilości wykonywanej pracy, ale skutkuje programami o znacznie wyższej jakości. Wciąż nie ma wystarczającego doświadczenia z tą metodologią do pełnej oceny, ale kiedy działa, wydaje się być bardzo skuteczna. Spodziewamy się, że zdobędzie znaczące miejsce obok innych, cięższych metodologii.

Psychologia inżynierii oprogramowania

Projektanci metodologii często zapominają o najważniejszym czynniku w inżynierii oprogramowania: analitykach, projektantach, programistach i menedżerach. To wszystko są ludzie, z ludzkimi mocnymi i słabymi stronami. Jedną ze słabości jest tendencja aby łamać zasady, co utrudnia egzekwowanie wymagań metodologii. Możliwe jest zmuszenie programistów do tworzenia obszernej dokumentacji; prawie niemożliwe jest zmuszenie ich do stworzenia użytecznej dokumentacji. Skłonność ludzi do łamania zasad jest również mocną stroną, jeśli jest właściwie stosowana. Powszechnie wiadomo, choć często zapomina się, że poza metodologicznymi istnieje wiele czynników, które mają duży wpływ na wydajność i produktywność ludzi. Słynna anegdota opowiada o dużej świetlicy dla studentów w uniwersyteckim centrum obliczeniowym, w którym na jednym końcu znajdowało się kilka automatów. Kiedy kilku studentów narzekało na hałas dobiegający z tego rogu pokoju, administracja przeniosła maszyny w inne miejsce. Zaraz potem pojawił się poważniejszy problem: dwaj konsultanci, których centrum obliczeniowe pomagało studentom w rozwiązywaniu ich problemów, zostało zalanych tworząc długie kolejki przed ich pokojem. Okazało się, że hałas wokół automatów był często powodowany przez studentów rozmawiających ze sobą o swoich problemach komputerowych, a ponieważ problemy często były podobne, większość z nich rozwiązywano na miejscu. Do konsultantów zgłaszano tylko nietypowe problemy. Wraz z usunięciem tej nieformalnej, ale bardzo skutecznej usługi, cała sytuacja zmieniła się na gorsze. W ostatnich latach pojawiło się kilka metodologii, które próbują rozwiązać problem człowieka. Zamiast nazywać siebie „lekkimi metodologiami", co jest negatywnym terminem, który odróżnia ich od innych „ciężkich" metodologii, zaczęli nazywać siebie zwinnymi. Termin ten leży u podstaw zmiany punktu ciężkości z zarządzania dokumentami na zarządzanie oparte na ludziach. Metodologie zwinne, których jednym z przykładów jest programowanie ekstremalne, próbują dać ludziom motywację i wsparcie, których potrzebują do wykonywania swojej pracy, i ufają im, że wykonają ją dobrze. W rezultacie kładą duży nacisk na komunikację między klientami, projektantami i programistami i opowiadają się za tym, aby wszyscy pracowali blisko siebie; jeśli to możliwe, w tym samym pokoju. Gdy członkowie zespołu skutecznie się komunikują, potrzeba znacznie mniej dokumentacji papierowej. Zamiast śledzić postępy za pomocą dodatkowej dokumentacji, metodyki zwinne wykorzystują działające oprogramowanie. O wiele łatwiej jest ocenić działający produkt niż dokument projektowy. Oczywiście oznacza to, że proces musi często wspierać produkcję stabilnych wersji. W wyniku tej filozofii zespoły stosujące metodyki zwinne mogą szybciej reagować na zmiany niż te, które korzystają ze stałych planów. Tradycyjne metodologie zgodne ze strategią „linii produkcyjnej" i postrzegają programistów (takich jak analitycy, projektanci, koderzy i testerzy) jako

wymiennych w ramach ich poszczególnych kategorii. Metodologie zwinne skupiają się zamiast tego na indywidualnym rzemiośle, bez sztywnych granic i gdzie programiści zdobywają wiedzę i doświadczenie współpracując z innymi. Zaczynają jako praktykanci, wykonując prostsze i bardziej żmudne części pracy, ale nie są trzymani z dala od zadań, które wymagają większej wiedzy. W miarę postępów podejmują się niektórych bardziej skomplikowanych zadań, aż osiągną pozycję, w której sami stają się rzemieślnikami. Ważne jest, aby zrozumieć, że nie odpowiada to zwykłemu przejściu od programisty do menedżera. Rzemieślnik przyjmuje na siebie więcej obowiązków niż praktykant, w tym obowiązki związane z zarządzaniem, ale to nie uniemożliwia mu robienia tego, co robi najlepiej -rzeczywistego tworzenia oprogramowania. . Nie trzeba dodawać, że takie podejście kładzie nacisk na jednostkę, a nie na proces, i jest bardzo preferowane przez samych programistów. Oczywiste jest, że większe zespoły lub te opracowujące aplikacje krytyczne dla życia muszą używać bardziej formalnych procesów z większą ilością dokumentacji niż mniejsze zespoły opracowujące mniej krytyczne oprogramowanie. Rzeczywiście, istnieje wiele odmian zwinnych metodologii, ukierunkowanych na różne typy projektów. W miarę jak podejście to stanie się bardziej znane i rozpowszechnione, prawdopodobnie zostanie ono przetestowane w większych organizacjach, w tym w tych, które opracowują krytyczne aplikacje. Miłośnicy zwinnych metodologii tworzenia oprogramowania twierdzą, że nie ma niczego, czego nie można zrobić przy użyciu tych metodologii. Czas pokaże, czy mają rację.

Etyka zawodowa

Ze względu na coraz większe wykorzystanie oprogramowania w wielu krytycznych systemach z jednej strony, a naszą niewystarczającą zdolność weryfikacji jego poprawności z drugiej, duży nacisk należy położyć na rzetelność i profesjonalizm osób, które specyfikują, projektują i rozwijają systemy komputerowe. Muszą dołożyć wszelkich starań, aby tworzone przez nich oprogramowanie było najwyższej jakości, zgodnie z najlepszymi dostępnymi praktykami w tej dziedzinie. Dwie główne profesjonalne organizacje komputerowe, ACM i IEEE Computer Society, przygotowały kodeksy etyczne dla specjalistów komputerowych. Wyszczególniają one szczegółowe obowiązki programistów systemów w odniesieniu do ogółu społeczeństwa, a w szczególności organizacji, dla których pracują. Na przykład, poucza się ich, aby w swojej pracy przyczyniali się do dobrobytu społeczeństwa i ludzi, unikali krzywdzenia innych, byli uczciwi i uczciwi oraz szanowali prywatność, poufność i własność intelektualną innych. Powinni dążyć do najwyższej jakości i efektywności swojej pracy zawodowej, edukować społeczeństwo w zakresie implikacji systemów skomputeryzowanych oraz doskonalić własną edukację techniczną. Menedżerowie powinni tworzyć środowiska pracy, które ułatwiają pracownikom przestrzeganie kodeksu i muszą mieć pewność, że potrzeby wszystkich osób, na które mają wpływ ich produkty, są brane pod uwagę. Powiązany problem dotyczy certyfikacji. Czy profesjonaliści komputerowi powinni być certyfikowani przez jedno ze stowarzyszeń zawodowych i czy taka certyfikacja powinna być wymogiem przy pracy nad krytycznymi projektami? Stowarzyszenie Komputerowe IEEE ma dla swoich członków dobrowolny program certyfikacji, który oprócz wymagań technicznych dotyczących certyfikacji, wymaga również przestrzegania kodeksu etyki. Oczekuje się, że takie programy, wraz z włączeniem kursów etycznych do programu nauczania informatyki, doprowadzą do wyższych standardów etycznych w zawodzie, a tym samym do wyższej jakości produktów.

Badania nad inżynierią oprogramowania

Pod koniec lat siedemdziesiątych kandydat do pracy przybył do jednego z ośrodków badawczych IBM na wykład i rozmowę kwalifikacyjną. Temat, na który wykładał, miał coś wspólnego z porównywaniem języków programowania. Po około 15 minutach jego wystąpienia stało się jasne, że planuje zaprezentować metodę porównywania składni języków programowania. Jedna z osób na widowni, ekspertka od języków formalnych, podniosła palec i powiedziała potulnie: „Ale ten problem jest

nierozstrzygnięty. Pokażę ci prawdziwy praktyczny algorytm.” Oczywiście nigdy nie dostał pracy . . . W dyscyplinach inżynierskich istnieje tendencja do ciągłego napięcia między teorią a praktyką i potrzeba czasu, aby idee teoretyczne dotarły do praktyków. Niektóre pomysły szybko się przyjmą, inne zabierają więcej czasu, a niektóre pozostają czystą teorią (co często nie umniejsza ich znaczenia). W algorytmice względy wydajnościowe (rozdział 6) bardziej bezpośrednio przemawiały do praktyków niż poprawność. Wynika to prawdopodobnie z dwóch powodów. Po pierwsze, rażąco niewydajny program jest bezużyteczny, podczas gdy program z kilkoma rzadko ujawnianymi błędami nadal może być używany (choć z pewną frustracją). Po drugie, odkrycie nowego, wydajniejszego algorytmu, coś najczęściej wykonane przez teoretyków, jest zwykle łatwe do przetłumaczenia na kod, a gdy już to zrobi, kod może być używany w wielu programach. Z drugiej strony udowodnienie poprawności programu jest niezwykle trudne i musi być przeprowadzane osobno dla każdego nowego programu.

Badania nad poprawnością algorytmów wpłynęły na inżynierię oprogramowania na wiele sposobów. Najintensywniejsze wykorzystanie metod formalnych w przemyśle to weryfikacja, czyli formalne udowodnienie, że produkt spełnia swoją specyfikację.. Istnieją również dość wyrafinowane narzędzia, które mogą pomóc zweryfikować dowód przedstawiony przez eksperta. Narzędzia te mają dwie zalety. Po pierwsze, konstruują całkowicie formalne dowody, na poziomie szczegółowości nieosiągalnym dla człowieka (nie dlatego, że jest to dla człowieka zbyt trudne intelektualnie, ale z przeciwnego powodu: jest zbyt długi i żmudny). Po drugie, pozwalają użytkownikom skoncentrować się na koncepcjach i strategiach wysokiego poziomu, pozwalając narzędziu wypełnić szczegóły. Mimo to weryfikacja wymaga dużo czasu i wysiłku i jest stosowana głównie przez producentów sprzętu, ponieważ błędy sprzętowe są znacznie bardziej kosztowne w naprawie niż błędy oprogramowania. Na przykład wszystkie podstawowe operacje zmiennoprzecinkowe na mikroprocesorze AMD Athlon zostały mechanicznie zweryfikowane pod kątem zgodności ze standardową specyfikacją. Prace zostały wykonane w AMD przed wyprodukowaniem Athlona i odkryto kilka błędów. Motorola i Intel również odniosły imponujące sukcesy w korzystaniu z narzędzia weryfikacji. Ważnym wkładem formalnych metod badawczych do praktyki programowania jest projektowanie na podstawie umowy, o którym wspomniano w Części 5. Twierdzenia, które można napisać w językach takich jak EIFFEL, nie są wystarczająco silne, aby wyrazić wszystkie matematyczne właściwości programu potrzebne do weryfikacji. Są one jednak wykonywalne w tym sensie, że komputer może sprawdzić, czy są one prawdziwe podczas wykonywania programu. Zatem projektowanie na podstawie umowy jest dobrym przykładem użytecznego formalnego rozumowania dotyczącego programów. Zwiększa jakość programów, a także szybkość rozwoju (biorąc pod uwagę, że pozwala to na wczesne wykrywanie błędów), a to rekompensuje dodatkowy wysiłek wymagany przy pisaniu asercji. Ponadto zachęca programistów do zastanowienia się nad formalnymi właściwościami ich kodu, a nawet do udowodnienia sobie (na razie bez automatycznej pomocy), że ten konkretny problem nazywa się problemem równoważności dla języków bezkontekstowych i został udowodniony być nierozstrzygalnym. są poprawne. W miarę jak narzędzia do dowodzenia twierdzeń stają się coraz bardziej wyrafinowane, będą w stanie zaoferować większą pomoc w tym procesie. Badania doprowadziły również do postępu w zakresie języków wymagań i specyfikacji, głównie tych, które są zarówno wizualne, jak i matematyczne, i które są używane do określania złożonych zachowań.

Systemy reaktywne

W poprzedniej części omówiono ogólne problemy, które pojawiają się, gdy musimy opracować nie tylko algorytmy i programy, ale także duże i złożone systemy. W tym rozdziale skoncentrujemy się na jednym szczególnie problematycznym typie systemu oraz na jego najtrudniejszym i śliskim aspekcie. Rodzaje systemów, o których myślimy, mają głównie charakter reaktywny, a trudnym aspektem jest określenie, przeanalizowanie i wdrożenie ich zachowania w czasie. Niektóre systemy są rzeczywiście złożone, ale mają charakter transformacyjny; są typu wejścia/procesu/wyjścia, a ich zalecona praca jest wykonywana wielokrotnie dla każdego nowego zestawu danych wejściowych. Oznacza to, że swoją złożoność zawdzięczają obliczeniom i przepływowi danych. W takich przypadkach możemy powiedzieć, że nasz przyjaciel, mały Runaround, ma dużo do myślenia lub dużo podnoszenia i przenoszenia. Typowy komponent takiego systemu czeka na przybycie wszystkich danych wejściowych, przetwarza je, wysyła dane wyjściowe do kilku innych komponentów i wraca do stanu uśpienia, dopóki nie zostaną wyświetlone nowe dane wejściowe. Systemy tego rodzaju można w zadowalający sposób opisywać i analizować za pomocą technik przepływu danych, które identyfikują przetwarzanie zachodzące wewnątrz różnych komponentów oraz dane, które przepływają między nimi. Zatem dynamiczne zachowanie systemu transformacyjnego jest w dużej mierze zdeterminowane przez połączenia przepływowe między składnikami lub obiektami. Znacznie bardziej problematyczne są te systemy (zwłaszcza te duże i złożone), które są silnie sterowane kontrolą lub zdarzeniami. Takie systemy nazwano reaktywnymi. Ich rolą w życiu jest reagowanie na wiele różnych rodzajów zdarzeń, sygnałów i warunków w zawiły sposób. Systemy reaktywne niekoniecznie muszą być współbieżne lub rozproszone, chociaż wiele z nich jest. Wiele z nich ma również krytyczne znaczenie czasowe, często muszą reagować na wydarzenia w czasie rzeczywistym. Tutaj mały Runaround lub wiele z nich, jeśli system jest również współbieżny, musi być niezwykle czujny, responsywny i szybki. W rzeczywistości nie będzie przesadą stwierdzenie, że ogromna część skomputeryzowanych systemów jest reaktywna lub ma dominujące reaktywne części. Przykłady obejmują stosunkowo małe systemy, takie jak magnetowidy (VCR), telefony komórkowe i bankomaty, oraz znacznie większe i złożone, takie jak systemy samochodowe i awioniki, zakłady chemiczne, sterowania systemy, sterowniki telefoniczne i komunikacyjne, roboty przemysłowe oraz interaktywne pakiety oprogramowania, takie jak edytory tekstu i edytory programów. Systemy te muszą utrzymywać zawiłe dynamiczne relacje z otoczeniem, odpowiednio i na czas reagować na naciskanie przycisków, wzrost temperatury powyżej poziomu krytycznego, odkładanie odbiorników, przesuwanie kursorów na ekranie i tak dalej. Często zawierają one również wewnętrznie rozległą interakcję reaktywną; to znaczy pomiędzy różnymi komponentami tworzącymi system. A w dobie Internetu coraz więcej systemów intensywnie korzystających z sieci reaguje również, a manipulacja w jednym miejscu powoduje reakcje w innym, z dużą ilością tej reaktywności zachodzącej jednocześnie. Tak więc dominująca część złożoności systemu reaktywnego nie wynika ze złożonych obliczeń lub przepływu danych, ale ze skomplikowanych wzorców przyczynowo-skutkowych, wyzwalaczy/odpowiedzi, zwykle połączonych z wysokim stopniem współbieżności i aspektów czasowych. Głównym problemem związanym z reaktywnością jest określenie zachowania systemu w czasie, jasno i poprawnie oraz w sposób, który można łatwo i niezawodnie wdrożyć, przeanalizować i zweryfikować. Co się stanie i kiedy? Dlaczego te rzeczy się wydarzą i co jeszcze spowodują, że stanie się na ich przebudzeniu? Czy w międzyczasie mogą się zdarzyć inne rzeczy? Czy niektóre rzeczy są obowiązkowe, a inne po prostu dozwolone? Jakie są ograniczenia czasowe, gdy coś się dzieje? Jaki jest skutek tego, że oczekiwane rzeczy nie dzieją się wtedy, gdy powinny? Co może się nie wydarzyć w żadnych okolicznościach? I tak dalej. Co ciekawe, reaktywność nie jest wyłączną cechą systemów komputerowych stworzonych przez człowieka. Występuje również w układach biologicznych, które pomimo tego, że są dużo mniejsze niż my ludzie i nasze domowe artefakty, mogą być również dużo bardziej skomplikowane. I występuje również w

systemach ekonomicznych i społecznych, które są dużo większe niż pojedynczy człowiek. Te również mają skomplikowany charakter reaktywny i są w stanie w pełni zrozumienia i analizowania ich, a także przewidywanie ich przyszłych zachowań, wymaga tego samego rodzaju myślenia, co systemy komputerowe. Prowadzi to do przekonania, że niektóre rozwiązania oferowane przez informatykę i inżynierię systemów mogą być również wykorzystane do radzenia sobie z taką niekomputerową reaktywnością. Podwójnie możemy również dowiedzieć się wiele o tym, jak radzić sobie z skomputeryzowaną reaktywnością, obserwując Matkę Naturę zajmującą się własnymi systemami reaktywnymi.

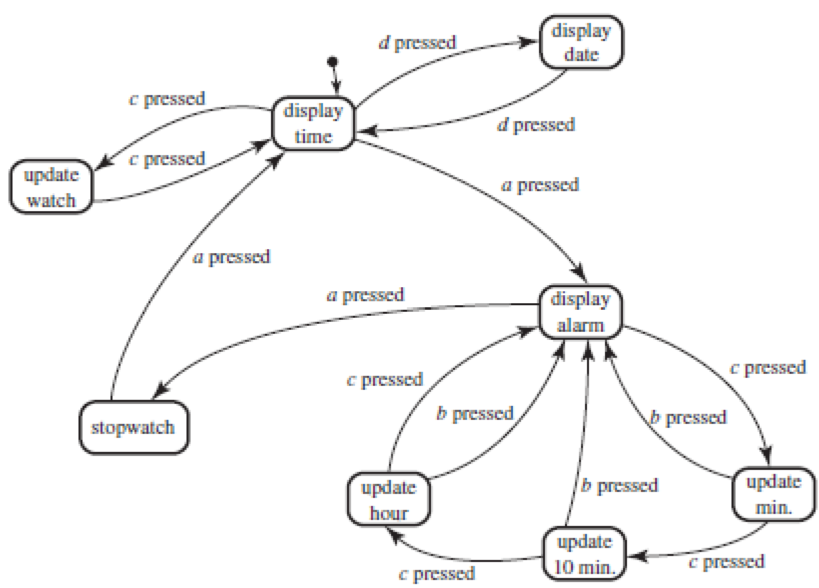
Formalizmy wizualne dla zachowań reaktywnych

Głównym artefaktem potrzebnym do opracowania niezawodnego systemu reaktywnego jest ogólny model systemu, który składa się ze starannie połączonej kompleksowej reprezentacji aspektów strukturalnych i behawioralnych systemu. Służy jako narzędzie dla projektantów i projektantów do uchwycenia ich przemyśleń i włączenia elementów z wymagań i projektu, a następnie może prowadzić aż do pomyślnego wdrożenia. Pod pewnymi względami model przypomina zestaw planów narysowanych przez architekta w celu opisania domu lub mostu. Różnica polega jednak na dynamice: systemy reaktywne zmieniają się w czasie; robią rzeczy; zachowują się. Nie dotyczy to domów czy mostów. Tak więc, podczas gdy opis strukturalny systemu reaktywnego można uznać za jego kręgosłup, jego zachowanie jest w zasadniczym sensie jego sercem i duszą. Zachowanie w systemie reaktywnym jest jak silnik w samochodzie; bez niego też nie mogą się „poruszać”. Ponadto zachowanie w czasie jest znacznie mniej namacalne niż ogólna funkcjonalność reaktywnego systemu lub jego fizyczna struktura, a przede wszystkim ten aspekt sprawia, że rozwój reaktywnych systemów jest tak trudny i podatny na błędy. Jednym z podejść do problemu specyfikowania takich systemów są formalizmy wizualne, języki diagramatyczne i intuicyjne, ale matematycznie rygorystyczne. Przy wszystkich innych rzeczach równych, obrazy są zwykle lepiej rozumiane niż tekst czy symbole, a właściwie użyte mogą służyć do myślenia na wyższym poziomie abstrakcji niż tekst. Ale te języki to nie tylko grafika. Podobnie jak języki programowania wysokiego poziomu wymagają nie tylko edytorów i narzędzi do wyświetlania, ale także – i to znacznie ważniejsze! - kompilatory, interpretery i narzędzia do debugowania, więc języki do modelowania zachowania systemów reaktywnych wymagają znacznie więcej niż ładnych diagramów z ładnymi edytorami graficznymi. Potrzebujemy środków do uruchamiania lub wykonywania modeli oraz środków do kompilacji ich w konwencjonalny kod, czynności zwanej generowaniem kodu lub syntezą kodu. Zatem formalizmy wizualne dla zachowań reaktywnych, podobnie jak konwencjonalne języki programowania obliczeń, muszą być uzupełnione składnią, która określa, co jest dozwolone, i semantyką, która określa, co oznaczają dozwolone rzeczy. Wizualizacja często opiera się na użyciu pól i strzałek, ze związkami topologicznymi między nimi, takimi jak hermetyzacja, połączenie i sąsiedztwo. Takie języki są często hierarchiczne i modułowe. Istnieje wiele języków określania zachowań reaktywnych, z których kilka można zaklasyfikować jako w pełni rozwinięte formalizmy wizualne. Niektóre z najbardziej interesujących to sieci Petriego i diagramy SDL = zarówno wizualne formalizmy, jak i Esterel, Signal i Lustre, które z wyglądu przypominają bardziej języki programowania, chociaż mają również graficzne nakładki. Opiszemy teraz jeden przykład formalizmu wizualnego dla zachowań reaktywnych, zwany wykresami stanów.

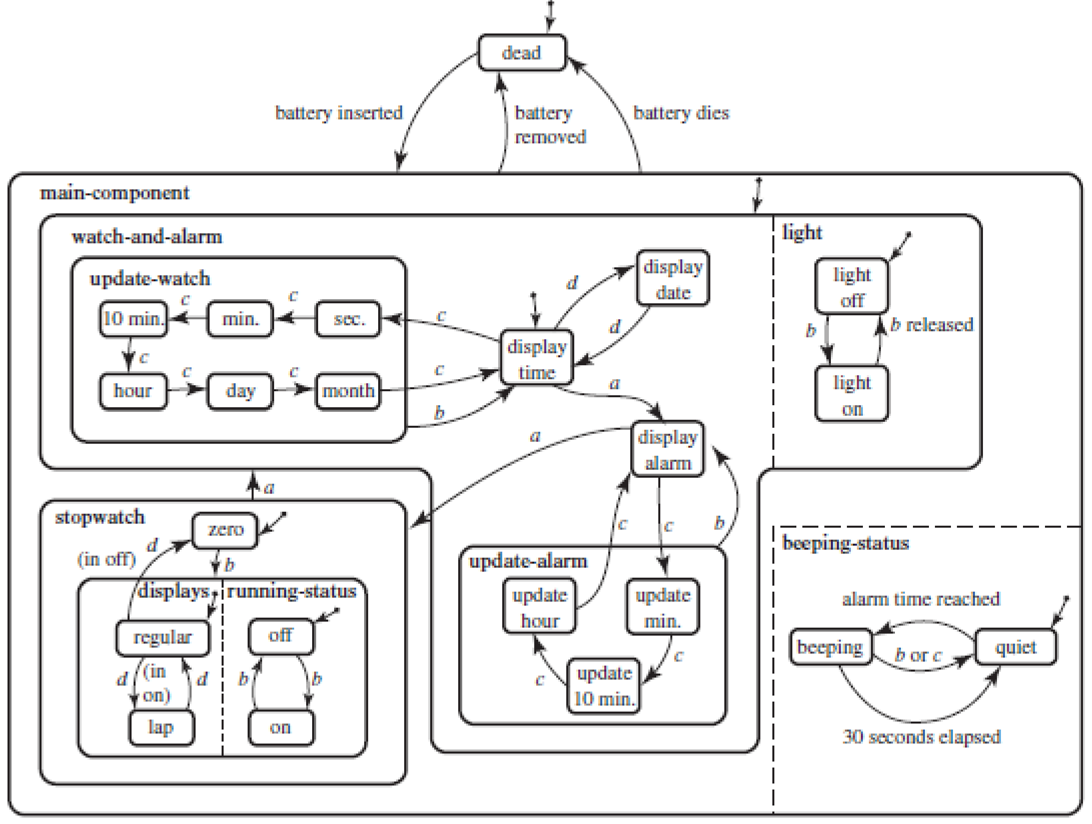
Schematy zachowań reaktywnych

Automaty skończone i związane z nimi diagramy przejść stanów wydają się być zadowalającym punktem wyjścia do określenia zachowania reaktywnego. Identyfikujemy stany systemu lub tryby działania i przystępujemy do określania zdarzeń i warunków, które powodują przejścia między stanami oraz działań (na przykład wysyłanie i odbieranie danych, rozpoczynanie lub zatrzymywanie działań itp.), które są w nich wykonywane. Część 9 zawierał prosty przykład diagramu opisującego część

zachowania zegarka cyfrowego . Istnieje jednak kilka problemów związanych z naiwnym używaniem diagramów stanów w przypadku złożonych przypadków. Po pierwsze, diagramy są „płaskie", podczas gdy reaktywne zachowanie nawet stosunkowo małych systemów w naturalny sposób spada na poziomy szczegółowości. (Odcięcie zasilania elektrycznego jest wydarzeniem bardzo wysokiego poziomu, podczas gdy przesuwanie kursora ekranowego nad jakąś ikoną jest wydarzeniem niskiego poziomu.) Te poziomy są korzystne nie tylko dla jasności i zrozumienia, ale także podczas samego procesu tworzenia. Rozważ zegarek z rysunku.



Chcielibyśmy kontynuować doprecyzowanie go, opisując zachowanie samej funkcji stopera (na przykład, jak i kiedy jest uruchamiana i zatrzymywana) oraz różne możliwości aktualizacji. Wszystko, co mamy do aktualizacji, to stan o nazwie update-watch, który prawdopodobnie oznacza lub zawiera pewną liczbę podstanów, które zajmują się aktualizacją sekund, minut, godzin i miesięcy, podobnie jak istnieją trzy stany do aktualizacji alarmu. Po drugie, diagramy stanów są sekwencyjne i nie uwzględniają w naturalny sposób współbieżności. Załóżmy na przykład, że nasz zegarek ma oświetlenie do podświetlania, które można włączać lub wyłączać niezależnie od innych rzeczy, które dzieją się gdzie indziej. Jeśli światło jest całkowicie oddzielną jednostką sterowaną, powiedzmy, oddzielnym przyciskiem, to wystarczy nam nowy dwustanowy diagram opisujący je jako osobny system. W rzeczywistości jednak funkcja światła w zegarku cyfrowym jest mniej prosta. Może nie zawsze mieć zastosowanie - być może nie działa w stanie stopera. Oczywiście możliwe jest połączenie jasnych informacji z resztą opisu, dając po dwa stany dla każdego ze starych; jeden na wypadek, gdy światło jest włączone, a drugi na wypadek, gdy jest wyłączone. Jeśli jednak podejście to zostanie przyjęte ogólnie do radzenia sobie ze współbieżnością, skutkuje to wykładniczym wzrostem liczby tatów, które należy wyraźnie opisać. (Dlaczego?) Oczywiście te problemy, które pojawiają się nawet w tak małych systemach, jak zegarki cyfrowe, są znacznie bardziej dotkliwe w dużych i złożonych systemach reaktywnych, które zazwyczaj mają ogromną liczbę stanów. Próbując zaradzić tym niedociągnięciom, rozszerzono diagramy stanów, uzyskując język zwany wykresami stanów. Rysunek pokazuje wykres stanu dla bardziej szczegółowej wersji zegarka z rysunku powyżej, który zawiera teraz komponent świetlny, udoskonalone stany stopera i aktualizacji oraz status sygnału dźwiękowego.



(Dla zwięzłości, pominęliśmy słowo „wciśnięty" w zdarzeniach naciśnięcia przycisku). Wykresy stanów pozwalają na wielopoziomowe stany, rozłożone w sposób i/lub sposób, a tym samym obsługują zwartą specyfikację. Aby uchwycić hierarchię stanów, prostokąty regularnych diagramów stanów można ułożyć w klastrowy, hermetyczny sposób. Na przykład stan obserwowany na rysunku składa się z sześciu zamkniętych podstanów, powiązanych między sobą wyłącznym „lub": bycie w trybie obserwowania aktualizacji oznacza w rzeczywistości przebywanie dokładnie w jednym z jego sześciu podstanów. Mogliśmy pogrupować te sześć stanów dla jasności lub w celu stopniowego rozwoju. Być może najpierw zdecydowaliśmy, że będzie stan do aktualizacji (co rzeczywiście mamy), a dopiero później przystąpiliśmy do samodoskonalenia. Co więcej, sześć stanów aktualizacji ma co najmniej jedną wspólną właściwość: wszystkie reagują na naciśnięcie b, zatrzymując proces aktualizacji i powracając do czasu wyświetlania. Gdyby użyto „płaskich" diagramów, wymagałoby to sześciu oddzielnych strzałek, po jednej wychodzącej z każdego stanu. Jeśli jesteśmy w aktualizacji-watch i przycisk b jest wciśnięty, wychodzimy i wprowadzamy czas wyświetlania. Jednakże, ponieważ bycie w trybie obserwacji aktualizacji znajduje się w jednym z podstanów, następuje pożądany efekt. Podobnie, jeśli bateria zostanie wyjęta lub wygaśnie, następuje przejście do stanu martwego, bez względu na to, w jakim byliśmy. Są to czasami nazywane przerwaniami. Aby określić współbieżne składniki stanu, wykresy stanów używają partycjonowania z linią przerywaną. Dodają one wymiar jednoczesności i są nazywane składowymi ortogonalnymi mapy stanów. Relacja między komponentami ortogonalnymi to nie „lub", ale „i". Na przykład, jeśli jesteśmy w stanie stopera z rysunku , ale nie w stanie zero, to musimy być jednocześnie zarówno na wyświetlaczach, jak i w stanie pracy. W każdym z nich istnieją dwie możliwości, powiązane przez „lub" i dające w sumie cztery możliwe konfiguracje stanów. Dla kontrastu z tymi czterema, rozważ składnik światła, w którym dodaliśmy dwa stany zamiast 12, których potrzebowalibyśmy bez składowych ortogonalnych, oraz status dźwiękowy, w którym dodano dwa zamiast 29. (Skąd się wzięły 12 i 29 ?) To zmniejszenie rozmiaru pomaga przezwyciężyć omówiony wcześniej problem wykładniczego rozrostu. Małe strzałki oznaczające stany początkowe w zwykłych

diagramach stanów mogą pojawić się na dowolnym poziomie w schemacie stanów. Tutaj nazywane są wartościami domyślnymi. W ramach głównego komponentu stanu wysokiego, na przykład, stan kombinowany składający się z ortogonalnych komponentów zegarka i alarmu oraz światła jest domyślny, więc jeśli bateria jest włożona w stanie martwym, wprowadzamy tę kombinację, a nie stoper. (Oczywiście wchodzimy również w stan ciszy w stan pikania.) Teraz, w każdym z tych prostopadłych składowych jest również domyślna strzałka - musi być jedna, w przeciwnym razie nie wiedzielibyśmy, w który z ich podstanów wejść - tak że naprawdę kończą się w konfiguracji „czas wyświetlania, wyłączanie, cisza". Wykresy stanów mają szereg dodatkowych funkcji, takich jak możliwość określenia ograniczeń czasowych oraz możliwość oparcia przejść na zachowanie w przeszłości. Powinniśmy jednak pamiętać, że formalizmy, takie jak wykresy stanów, są w stanie opisać tylko część kontrolną systemu reaktywnego, a nie jego przepływ danych czy aspekty strukturalne. Specyfikacja behawioralna musi być połączona ze specyfikacją struktury systemu, tak jak niesilnikowe części samochodu muszą być połączone z silnikiem, a te połączenia nie są takie proste. Jednym z najczęściej stosowanych podejść do tego jest łączenie języka zachowań reaktywnych, takich jak schematy stanów, z ideami orientacji obiektowej. Omówimy to podejście później.

Wykonanie modelu

Jednym z najbardziej interesujących pojęć, jakie wyszły z badań nad systemami i inżynierią oprogramowania, jest specyfikacja wykonywalnych specyfikacji lub, aby lepiej dopasować się do używanej tu terminologii, modele wykonywalne. Wykonywanie modelu może odbywać się albo bezpośrednio, w sposób analogiczny do uruchamiania interpretera w konwencjonalnym programie komputerowym, albo pośrednio, poprzez kompilację do kodu i uruchomienie kodu. Dostępnych jest szereg potężnych narzędzi obsługujących takie możliwości. Sednem wykonania modelu jest możliwość wykonania pojedynczego kroku dynamicznego działania systemu, z uwzględnieniem wszystkich konsekwencji. Podczas kroku środowisko może generować zdarzenia zewnętrzne, modyfikować wartości warunków i zmiennych itp. Takie zmiany wpływają następnie na stan systemu: wyzwalają nowe zdarzenia, aktywują i dezaktywują czynności, modyfikują warunki i zmienne, i tak na. I oczywiście każda z tych zmian może z kolei powodować wiele innych, często wywołując skomplikowane reakcje łańcuchowe. W wielu rodzajach systemów ograniczenia czasowe i czasowe odgrywają ważną rolę w określaniu sposobu wykonania kroku. Semantyka formalizmu użytego do określenia zachowania musi zawierać wszystkie informacje potrzebne do precyzyjnego uchwycenia tych zmian. Obliczenie efektu kroku na podstawie aktualnego stanu i zmian dokonanych przez otoczenie zwykle wiąże się ze skomplikowanymi procedurami algorytmicznymi, które wywodzą się i odzwierciedlają tę semantykę. W przypadku map stanów mechanizm wykonawczy musi być w stanie śledzić dynamikę wszystkich komponentów ortogonalnych we wszystkich aktywnych mapach stanu, biorąc pod uwagę reakcje łańcuchowe zdarzeń i zmian stanu oraz przeprowadzając wszystkie zalecone przez nie zmiany, w tym wszelkie działania które są powiązane ze stanami lub przejściami. Najprostszym sposobem wykonania lub „uruchomienia" modelu za pomocą skomputeryzowanego narzędzia jest interaktywny sposób „krok po kroku". Na każdym kroku użytkownik emuluje system środowiska, generując wydarzenia i zmieniając wartości. Narzędzie z kolei reaguje, przekształcając system w nowy wynikowy status. Jeśli model jest reprezentowany wizualnie, zmiana statusu zostanie również odzwierciedlona wizualnie, powiedzmy, poprzez zmiany koloru lub podkreślenia na diagramach. Tytułem ilustracji powróćmy na chwilę do przykładu zegarka. Obserwując topologiczne właściwości wykresu można łatwo wywnioskować, że zachowanie światła nie ma zastosowania do stanów stopera, ponieważ jest ono ortogonalne do stanu zegarka i alarmu, a wyłączne dla topwatcha. Jednak ten związek między światłem a obserwacją i alarmem ma bardziej subtelne implikacje, których nie można łatwo dostrzec bez faktycznego wykonania modelu. W szczególności załóżmy, że jesteśmy w konfiguracji „godzina, światło wyłączone, sygnał dźwiękowy", co oznacza, że aktualizujemy godzinę, światło jest wyłączone, a

brzęczyk emituje sygnał dźwiękowy, ponieważ nadeszła godzina alarmu. (To tak, jakby powiedzieć, że trzymamy zegarek w dłoniach i właściwie go „obsługujemy", zaczynając od opisanej sytuacji.) Teraz wciskamy przycisk b. Co się dzieje? Cóż, czy nam się to podoba, czy nie, trzy pozornie niepowiązane ze sobą rzeczy wydarzą się jednocześnie. Sygnał dźwiękowy zostanie zatrzymany (z powodu przejścia „b lub c" w tryb cichy), światło włączy się, dopóki b nie zostanie zwolnione, a aktualizacja zakończy się, a zegarek powróci do czasu wyświetlania! Gdy ta sekwencja zostanie uruchomiona za pomocą narzędzia, które wykonuje wykresy stanu, zmiany te pojawią się na wykresie stanu w sposób animowany, zwykle w specjalnym kolorze stanów, w których aktualnie znajduje się system, oraz ostatnio przebytych przejść. Poprzez interaktywne wykonywanie scenariuszy, które odzwierciedlają sposób, w jaki oczekujemy, że nasz system będzie się zachowywał, jesteśmy w stanie zweryfikować, czy rzeczywiście to zrobi, przed ostatecznym wdrożeniem. Jeśli stwierdzimy, że reakcja systemu nie jest zgodna z oczekiwaniami, możemy wrócić do modelu, zmienić go i ponownie uruchomić ten sam scenariusz. Jest to analogiczne do jednoetapowego lub wsadowego debugowania konwencjonalnych programów. Podczas wykonywania, użytkownik odgrywa rolę wszystkich części modelu, które są zewnętrzne w stosunku do wykonywanej części, nawet jeśli te części zostaną ostatecznie określone i staną się w ten sposób wewnętrznymi. Gdy mamy już podstawową umiejętność wykonania kroku, nasz apetyt rośnie. Możemy teraz chcieć, aby model wykonywał się w sposób nieinteraktywny. Aby np. sprawdzić, czy połączenie telefoniczne łączy się wtedy, gdy powinno, możemy przygotować odpowiednią sekwencję zdarzeń i sygnałów w pliku wsadowym, ustawić model tak, aby startował w stanie początkowym i poprosić nasze narzędzie o iteracyjne wykonywanie kroków, wczytanie zmian z pliku. Graficzna informacja zwrotna z takiego wykonania wsadowego staje się (często całkiem atrakcyjną) animacją diagramów. W rzeczywistości nie musimy ograniczać się do tworzenia samodzielnie opracowanych scenariuszy: możemy chcieć zobaczyć, jak model działa w okolicznościach, których sami nie chcemy szczegółowo określać. Chcielibyśmy zobaczyć jego działanie w losowych warunkach, zarówno w typowych, jak i mniej typowych sytuacjach. Możemy chcieć włączyć punkty przerwania do mechanizmu wykonawczego, powodując jego zawieszenie, a narzędzie do podjęcia określonych działań, gdy pojawią się określone sytuacje. Działania te mogą obejmować chwilowe wejście w tryb interaktywny w celu dokładnego monitorowania postępów krok po kroku, aż po wykonanie gotowego kodu opisującego czynność niskiego poziomu. Zdolności te trafiają w sedno potrzeby modeli wykonywalnych - aby zminimalizować nieprzewidywalne w rozwoju złożonych systemów reaktywnych. Wszystko to można posunąć o wiele dalej, ponieważ same wykonania modeli są programowane lub metaprogramowane za pomocą środków zewnętrznych. W ten sposób narzędzie wykonawcze można skonfigurować tak, aby wyszukiwało predefiniowane punkty przerwania i gromadziło informacje dotyczące postępu systemu w miarę jego przebiegu. Jako przykład możemy chcieć wiedzieć, ile razy podczas typowego lotu samolotu, który określamy, radar traci namierzony cel. Ponieważ inżynierowi może być trudno złożyć typowy scenariusz lotu, możemy wykorzystać moc naszego narzędzia, instruując je, aby uruchomiło wiele typowych scenariuszy, używając zgromadzonych wyników do obliczenia informacji o średnim przypadku. Narzędzie będzie następnie podążać za typowymi scenariuszami, generując losowe liczby w celu wybrania nowych zdarzeń zgodnie z predefiniowanymi rozkładami prawdopodobieństwa. Statystyki są następnie gromadzone przy użyciu odpowiednich punktów przerwania i prostych obliczeń. Podstawowe idee stojące za tymi technikami są oczywiście dobrze znane w testowaniu i debugowaniu programów. Chodzi tu jednak o rozszerzenie ich na formalizmy wizualne wysokiego poziomu używane do modelowania złożonych zachowań reaktywnych, na długo przed kosztownymi etapami implementacji i wdrażania finalnego systemu. Wykonanie modelu może ujawnić błąd, tj. zachowanie, które różni się od tego, co zamierzaliśmy. Oczywiście, jeśli tak się stanie, chcielibyśmy dowiedzieć się, co spowodowało anomalne zachowanie. Dlaczego wydarzyło się coś nieoczekiwanego? Dlaczego nie wydarzyło się coś innego, mimo że myśleliśmy, że powinno? Co by się stało, gdyby jakieś zewnętrzne wydarzenie miało miejsce lub miało miejsce wcześniej lub później? I tak

dalej. Wiele takich pytań odnosi się do zachowania modelu przed momentem wykrycia błędu. Aby móc odpowiedzieć na takie pytania, narzędzie musi przechowywać historię swoich przeszłych działań wraz z ich przyczynami. W przypadku braku takich informacji sprowadzamy się do wielokrotnego uruchamiania modelu od początku, zatrzymując się w różnych punktach, aby zaobserwować stany pośrednie i zobaczyć, gdzie jego zachowanie odbiega od tego, czego oczekujemy. Kolejnym krokiem w tym rosnącym apetycie na możliwości analizy zachowań reaktywnych jest oczywiście weryfikacja w rozumieniu Części 5. Typową właściwością, którą bardzo często chcemy zweryfikować, jest osiągalność, która określa, czy po uruchomieniu w jakimś W takiej sytuacji system może kiedykolwiek dojść do sytuacji, w której jakiś określony warunek stanie się spełniony. Warunek ten można postawić w celu odzwierciedlenia niepożądanych lub pożądanych sytuacji. Co więcej, możemy sobie wyobrazić, że test jest skonfigurowany tak, aby raportował pierwszy znaleziony scenariusz, który prowadzi do określonego stanu, lub raportował wszystkie możliwe, tworząc szczegóły samych scenariuszy. Czy takie testy są realistyczne? Czy moglibyśmy na przykład poddać model testowi osiągalności, po którym będziemy wiedzieć na pewno, czy istnieje jakakolwiek możliwość jego wystąpienia w każdych możliwych okolicznościach? Odpowiedź jest podobna do tej, którą udzieliliśmy, omawiając weryfikację w Części 5 oraz ograniczenia obliczeń w Częściach 7 i 8: w zasadzie nie, ale w wielu praktycznych sytuacjach tak. Rzeczywiście, w ostatnich latach poświęcono wiele pracy, aby weryfikacja programu zrobiła duży krok dalej, prowadząc do możliwości weryfikacji modeli wizualnych pod kątem złożonych zachowań reaktywnych. Wierzymy, że automatyczna weryfikacja krytycznych właściwości w systemach reaktywnych stanie się powszechna i że metody i narzędzia modelowania behawioralnego będą stawały się coraz potężniejsze, oferując środki do rutynowej weryfikacji właściwości w miarę rozwoju modelu.

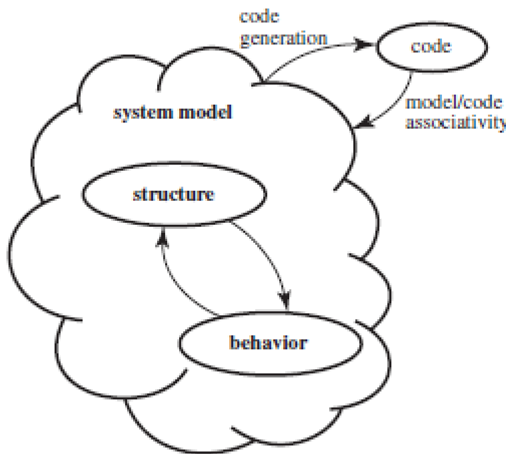
Synteza kodu

Jak wyjaśniono wcześniej, bezpośrednie wykonanie modelu jest analogiczne do uruchamiania programów za pomocą interpretera. Jednak wiele narzędzi generuje automatycznie z kodu modelu w jakimś konwencjonalnym języku, takim jak C++ lub JAVA, a następnie wykonuje ten kod. To jest analogia kompilacji. Kiedy patrzysz na model w wykonaniu, tak naprawdę nie możesz odróżnić.6 Jednym z głównych zastosowań danych wyjściowych generatora kodu jest obserwowanie działania systemu w warunkach zbliżonych do rzeczywistych. Na przykład kod może zostać przeniesiony i wykonany w rzeczywistym środowisku docelowym lub, jak to często bywa na wcześniejszych etapach, w symulowanej wersji środowiska docelowego. W ten sposób kod można połączyć z graficznym interfejsem użytkownika (GUI) systemu – makieta ekranowa tablic kontrolnych systemu, wraz z obrazami wyświetlaczy, przełączników, dźwigni, pokręteł i wskaźników – która reprezentuje rzeczywisty interfejs użytkownika końcowego systemu. GUI można wtedy manipulować w realistyczny sposób za pomocą myszy i klawiatury. Ważną kwestią jest to, że zachowanie symulowanego systemu nie jest napędzane przez pospiesznie napisany kod przygotowany specjalnie na potrzeby prototypu, ale przez kod, który został wygenerowany automatycznie z modelu, który zazwyczaj zostanie dokładnie przetestowany i przeanalizowany przed generowaniem kodu. Co więcej, gdy dostępne są części rzeczywistego środowiska docelowego, można je również połączyć z kodem, a przebiegi stają się jeszcze bardziej realistyczne. Generowanie kodu z modeli zachowań reaktywnych zbudowanych za pomocą formalizmów wizualnych może być zatem wykorzystane do celów wykraczających poza zespół programistów. Interfejsy GUI systemu oparte na kodzie mogą być używane jako część standardowej komunikacji między klientem a wykonawcą lub wykonawcą a podwykonawcą. Nie jest nierozsądne, aby taka działająca wersja modelu systemu była wymaganym produktem na niektórych etapach rozwoju. Dobre narzędzie do generowania kodu będzie miało również mechanizm debugowania, za pomocą którego użytkownik może prześledzić wykonywane części kodu z powrotem do modelu graficznego. Punkty przerwania można wstawić, aby zatrzymać przebieg, gdy wystąpią określone

zdarzenia, w którym to momencie można sprawdzić stan modelu, a elementy można modyfikować w locie przed wznowieniem przebiegu. Jeśli pojawią się poważne problemy, zmiany można wprowadzić w oryginalnym modelu, który jest następnie ponownie kompilowany do kodu i ponownie uruchamiany. Można zażądać plików śledzenia, rejestrując kluczowe informacje do przyszłej inspekcji i tak dalej.

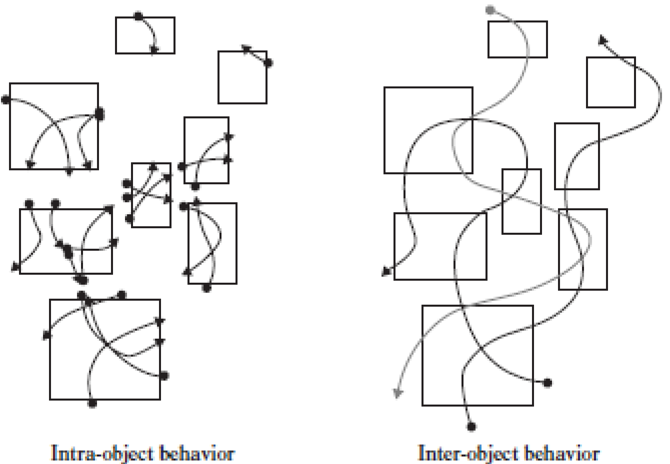
Dwa style zachowań

Jak określana jest struktura reaktywnego systemu i jak ta specyfikacja jest połączona z jego zachowaniem? Zaproponowano kilka metod, a jedną z bardziej rozpowszechnionych jest oparcie całego procesu modelowania i rozwoju na paradygmacie obiektowym. Prowadzi to do tak zwanej specyfikacji i analizy obiektowej. Główną ideą jest podniesienie pojęć z poziomu programowania obiektowego, jak opisano w rozdziale 3, na poziom modelowania i wykorzystanie formalizmów wizualnych. W przypadku struktury systemu do określania klas i ich wzajemnych relacji używany jest język diagramów zwany diagramami modeli obiektowych. Jeśli chodzi o określanie zachowania, większość podejść do modelowania obiektowego opiera zachowanie obiektu na maszynie stanów, a wiele z nich zaleca skonstruowanie wykresu stanu dla każdej klasy, przechwytującego pożądane zachowanie dowolnej jego instancji. Po utworzeniu instancji klasy kopia schematu stanu klasy rozpoczyna działanie, kontrolując zachowanie tej instancji. Szczegóły tego związku między strukturą a zachowaniem są o wiele bardziej skomplikowane, niż można to tutaj przedstawić. Klasy reprezentują dynamicznie zmieniające się kolekcje konkretnych obiektów, a modelowanie behawioralne musi dotyczyć kwestii związanych z tworzeniem i niszczeniem obiektów, delegowaniem komunikatów, modyfikacją relacji i konserwacja, kolejkowanie zdarzeń, agregacja klas, dziedziczenie, przetwarzanie wielowątkowe i tak dalej. Powiązania między zachowaniem a strukturą muszą być zdefiniowane wystarczająco szczegółowo i wystarczająco rygorystycznie, aby wspierać konstrukcję narzędzi, które umożliwiają omówione powyżej rodzaje wykonywania modeli i generowania kodu. Rysunek poniższy ilustruje te podstawowe części modelowania systemu.



Innymi słowy, zachowanie jest określone w tej konfiguracji w sposób oparty na stanie obiekt po obiekcie, zapewniając reaktywność każdego obiektu za pomocą, powiedzmy, mapy stanu. Gdyby to była wymyślona przez nas gra piłkarska, to podejście wymagałoby od nas szczegółowego zaprogramowania każdego z graczy – a także sędziów, piłki, drewnianych ram bramek itd. – określając sposób, w jaki reagują i reagują na każde zdarzenie lub wydarzenie, które nadchodzi ich droga. Po wykonaniu tej czynności możemy „uruchomić” system lub zasymulować grę, ponieważ mamy pełną informację o każdej możliwej parze wyzwalacz/odpowiedź dostępna dla każdego obiektu. Jak na razie dobrze. Jednak kiedy ludzie myślą o systemach reaktywnych, najczęściej myślą naturalnie w

kategoriach scenariuszy zachowań. Nie ma zbyt wielu ludzi mówiących takie rzeczy jak „Cóż, mechanizm nagrywania mojego magnetowidu może być w trybie bezczynności, w trybie nagrywania lub w trybie selekcji środkowej; w pierwszym przypadku są to możliwe dane wejściowe i reakcje magnetowidu, . . . ; w drugim przypadku oto, co się dzieje, itd.” Raczej zauważasz, że mówią rzeczy takie jak „Jeśli nacisnę ten przycisk, a następnie obrócę pokrętło, aby wskazać tutaj, na wyświetlaczu pojawi się następujący komunikat” lub mniej szczegółowe rzeczy, takie jak „Wybór daty i godziny, a następnie naciśnięcie Nagrywaj, powoduje, że magnetowid wykonuje następujące czynności . . . Mówią również o zabronionych scenariuszach, takich jak „Dopóki jest podłączony do gniazdka elektrycznego, magnetowid nigdy się nie wyłączy, gdy szpula kasety się obraca”. Wiele osób uważa, że o wiele bardziej naturalne jest opisywanie i omawianie zachowania reaktywnego systemu na podstawie jego scenariuszy, a nie na podstawie reaktywności każdego z jego składników opartej na stanie. Dotyczy to w szczególności niektórych wczesnych i późniejszych etapów procesu tworzenia systemu — np. podczas przechwytywania i analizy wymagań oraz podczas testowania i konserwacji. Mamy więc tutaj do czynienia z ciekawą i subtelną dychotomią. Jedna strona ma opisy behawioralne oparte na stanie, które pozostają w obiekcie i opierają się na zapewnieniu pełnego opisu reaktywności każdego z nich. Rodzaj podejścia wewnątrzobiektowego: „wszystkie fragmenty historii dla każdego obiektu”. Druga strona zawiera opisy zachowań oparte na scenariuszach, które przecinają granice obiektów systemu w celu dostarczenia zrozumiałych opisów scenariuszy zachowań (i zachowań zakazanych). Rodzaj podejścia międzyobiektowego: „każda historia podana poprzez wszystkie jej istotne obiekty”. To drugie jest bardziej intuicyjne i naturalne do uchwycenia przez ludzi i dlatego pasuje do wymagań i etapów testowania, ale to pierwsze podejście wydaje się być tym, które jest potrzebne do wdrożenia. Rzeczywiście, wydaje się, że implementacja systemu wymagałaby dostarczenia do każdego obiektu pełnego opisu obsługiwanych przez niego reakcji behawioralnych, tak aby można go było wykonać bezpośrednio lub poddać procesowi, który wygeneruje kod implementujący ten opis. Rysunek 3 jest próbą graficznego zilustrowania tych dwóch podejść.

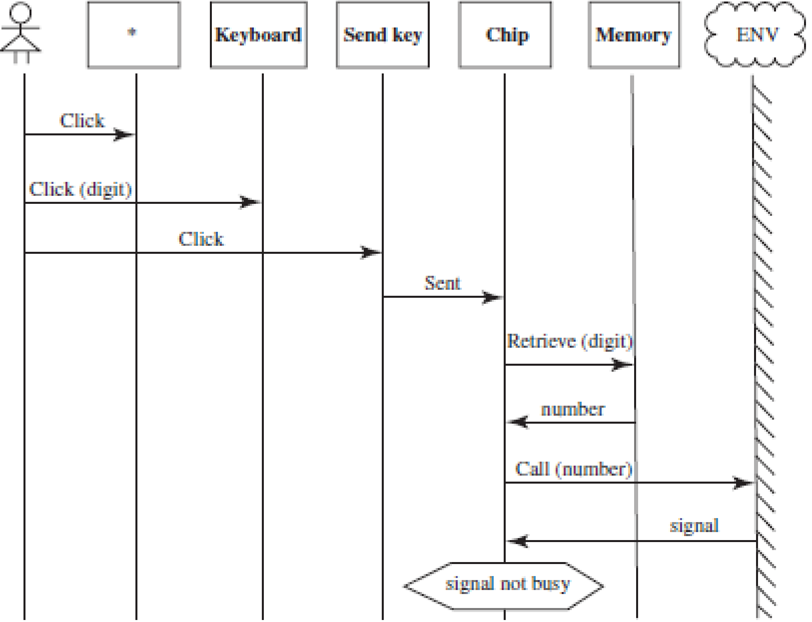


Po lewej stronie mamy każdy obiekt wypełniony wszystkimi jego małymi elementami zachowania - globalne zachowanie całego systemu wyprowadzone z wielkiej kombinacji tych wszystkich - a po prawej każda sekwencja zachowań przechodzi przez wszystkie istotne obiekty. Gdybyśmy chcieli na przykład opisać „zachowanie” typowego biura, dużo bardziej naturalne byłoby opisanie scenariuszy międzyobiektowych, takich jak wysyłanie przez pracownika 50 kopii dokumentu (może to dotyczyć pracownika sekretariatu, kserokopiarki, pokoju pocztowego itp.), czynności biurowych, których nie wolno wykonywać, o ile nie są one inicjowane przez szefa najwyższego szczebla, lub w jaki sposób informacje o urlopach i zwolnieniu lekarskim są organizowane i przekazywane do listy płac gabinet.

Porównajmy to ze stylem intra-obiektowym, w którym musielibyśmy podać pełną informację o sposobach działania i reaktywności szefa, sekretarza, pracowników, kserokopiarki, pokoju pocztowego itp. Tak więc, w przeciwieństwie do scenariuszy, stosowanych zazwyczaj do określania wymagań we wczesnych etapach rozwoju systemu, modelowanie za pomocą wykresów stanu lub bezpośrednio z kodem jest zwykle przeprowadzane na późniejszym etapie (zwykle projektowania) i skutkuje behawioralnym specyfikację dla każdej instancji obiektu (lub zadania lub procesu), podając szczegóły jego zachowania we wszystkich możliwych warunkach i we wszystkich „historiach". Ta specyfikacja obiekt po obiekcie będzie później testowana pod kątem scenariuszy i ewoluuje w implementację systemu, ponieważ pod koniec dnia ostateczny system będzie składał się z kodu (lub sprzętu) sterującego dynamicznym zachowaniem każdego obiektu. U podstaw tego procesu leży założenie - które wkrótce zakwestionujemy = że zachowanie oparte na scenariuszach międzyobiektowych nie jest wykonalne ani możliwe do wdrożenia. Rzeczywiście, jak działałby system opisany przez scenariusze? Co by zrobił w ogólnych warunkach dynamicznych? Skąd mamy wiedzieć w każdym punkcie, które scenariusze powinny się uruchomić i zacząć działać? W jaki sposób powinniśmy aktywnie upewniać się, że rzeczy, które muszą się wydarzyć, rzeczywiście się wydarzą, rzeczy, które mogą się wydarzyć, czasami się wydarzą, a rzeczy, które mogą się nie wydarzyć, rzeczywiście się nie wydarzą? Jak to wszystko egzekwujemy i co robimy, gdy napotykamy na niedookreślenie (brakujące informacje), przespecyfikowanie (niedeterminizm) i sprzeczności (zderzenia między „konieczne" i „nie wolno")?

LSCs dla zachowania międzyobiektowego

Formalizmem wizualnym używanym od wielu lat do określania scenariuszy, wywodzącym się z branży telekomunikacyjnej, jest język wykresów sekwencji komunikatów (MSC). Scenariusze są określone w MSC jako sekwencje interakcji komunikatów między instancjami obiektów. MSC są popularne w świecie zorientowanym obiektowo w fazie wymagań, w której inżynierowie identyfikują przypadki użycia – ogólne wzorce zachowań wysokiego poziomu – a następnie określają scenariusze, które je tworzą. Przechwytuje to pożądane zależności między instancjami obiektów oraz między nimi a środowiskiem zewnętrznym (np. użytkownikiem). Instancje obiektów są reprezentowane w MSC za pomocą pionowych linii, a wiadomości między tymi instancjami są reprezentowane przez poziome (lub czasami pochylone) strzałki. Warunkowe osłony, przedstawione jako wydłużone sześciokąty, określają stwierdzenia, które po osiągnięciu mają być prawdziwe. Ogólnym efektem takiego wykresu jest określenie scenariusza zachowania, składającego się z komunikatów przepływających między obiektami i rzeczy, które po drodze muszą być prawdziwe. Rysunek przedstawia prosty przykład MSC dla funkcji szybkiego wybierania telefonu komórkowego. Sekwencja komunikatów, które przedstawia, składa się z następujących elementów: użytkownik klika klawisz ∗, a następnie klika cyfrę na klawiaturze, a następnie klawisz Send, który wysyła wskazanie Sent do wewnętrznego Chipa, który z kolei wysyła cyfrę do Pamięci, aby pobrać numer telefonu skojarzony z klikniętą cyfrą. Chip następnie wysyła ten numer do otoczenia (np. anteny firmy komórkowej) w celu wykonania połączenia, po czym odbierany jest sygnał z otoczenia. Wreszcie, wykres zawiera warunek ochronny, który stwierdza, że sygnał rzeczywiście nie jest zajęty. Semantyka MSC jest egzystencjalna: wykres potwierdza, że scenariusz, który opisuje, reprezentuje możliwą sekwencję zdarzeń w życiu systemu. Zależne od czasu znaczenie samego scenariusza określają dwie proste zasady. Najpierw wzdłuż pionowej linii obiektu czas biegnie od góry do dołu. Po drugie, zdarzenie wysłania wiadomości poprzedza zdarzenie jej odebrania. Dlatego MSC nie mówią zbyt wiele o tym, co system faktycznie zrobi po uruchomieniu. Można ich użyć do powiedzenia, co może się wydarzyć, ale nie do określenia tego, co musi się wydarzyć. Na wykresie z rysunku 14.4, możemy na przykład zapytać, czy pamięć może „zdecydować", że nie będzie odsyłać numeru w odpowiedzi na żądanie od Chipa?

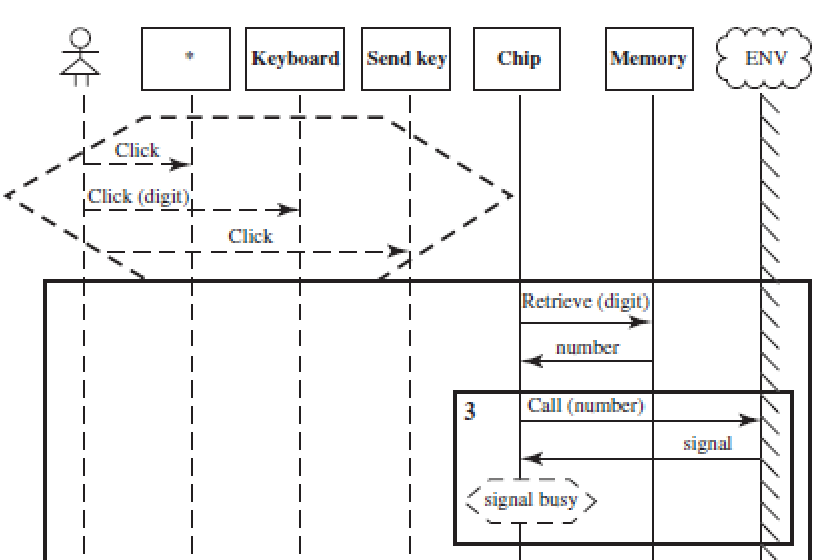


Czy warunek stwierdzający, że sygnał nie jest zajęty, musi być prawdziwy? Co się stanie, jeśli tak nie jest?

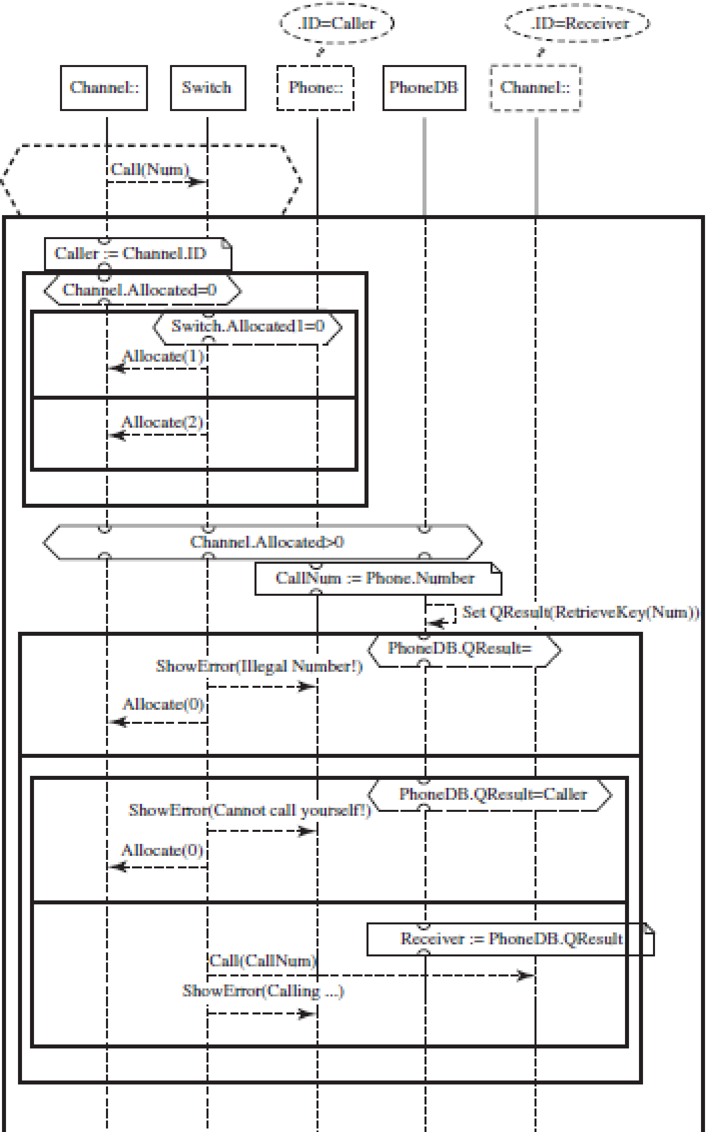
Takie wykresy rzeczywiście mogą być użyte do uchwycenia przykładowych scenariuszy oczekiwanego zachowania, które później zostaną porównane z ostatecznym systemem wykonywalnym. Jednak nie wystarczą, jeśli chcemy ich użyć do faktycznego stwierdzenia i potwierdzenia tego, co robi system. Chcielibyśmy móc powiedzieć, co może się wydarzyć, a co musi się wydarzyć, a także – jak wspomniano powyżej – co może się nie wydarzyć. Takie zabronione zachowania są czasami nazywane antyscenariuszami. Jeśli wystąpią podczas wykonywania systemu, jest coś bardzo nie tak: albo coś w specyfikacji behawioralnej nie zostało prawidłowo potwierdzone, albo implementacja nie spełnia poprawnie specyfikacji. Chcielibyśmy również móc określić wiele scenariuszy, które łączą się ze sobą, a nawet ze sobą, w subtelny sposób. Chcemy mieć możliwość określania ogólnych scenariuszy, tj. takich, które reprezentują wiele konkretnych scenariuszy, ponieważ mogą być tworzone przez różne obiekty tej samej klasy. Potrzebujemy zmiennych i środków do określania ograniczeń czasowych i tak dalej. MSC zostały rozszerzone na kilka sposobów, aby pomóc rozwiązać niektóre z tych problemów. Jedno z ostatnich rozszerzeń, zwane wykresami sekwencji na żywo lub LSC, wzięło swoją nazwę od możliwości określania żywotności, tj. rzeczy, które muszą wystąpić. LSC umożliwiają rozróżnienie między możliwym a koniecznym zachowaniem, zarówno globalnie, na poziomie całego wykresu, jak i lokalnie, podczas określania zdarzeń, warunków ochrony i postępu w czasie na wykresie. Tak więc w języku LSC występują dwa rodzaje wykresów: uniwersalne (oznaczone ciągłą linią graniczną) i egzystencjalne (opisane przerywaną linią graniczną). Bardziej interesujące są wykresy uniwersalne, które służą do określania zachowania opartego na scenariuszu, które ma zastosowanie do wszystkich możliwych uruchomień systemu. Uniwersalny wykres składa się z dwóch części, co zamienia go w rodzaj konstrukcji jeśli-to: prechart, który określa scenariusz, który, jeśli jest spełniony, zmusza system do spełnienia również drugiej części, głównego wykresu. udostępnia zestaw par scenariuszy akcji/reakcji, które muszą być spełnione przez cały czas podczas każdego uruchomienia systemu. W LSC, żywe elementy, określane jako gorące, oznaczają rzeczy, które muszą się wydarzyć, a inne, określane jako zimne, oznaczają rzeczy, które mogą się wydarzyć. Elementy gorące umożliwiają wymuszanie zachowania (i anty-zachowania), a elementy zimne mogą być używane do określania

struktur kontrolnych, takich jak rozgałęzienie i iteracja. Rysunek 14.5 pokazuje uniwersalny LSC, który jest w rzeczywistości wzbogaconą wersją MSC z rysunku powyżej

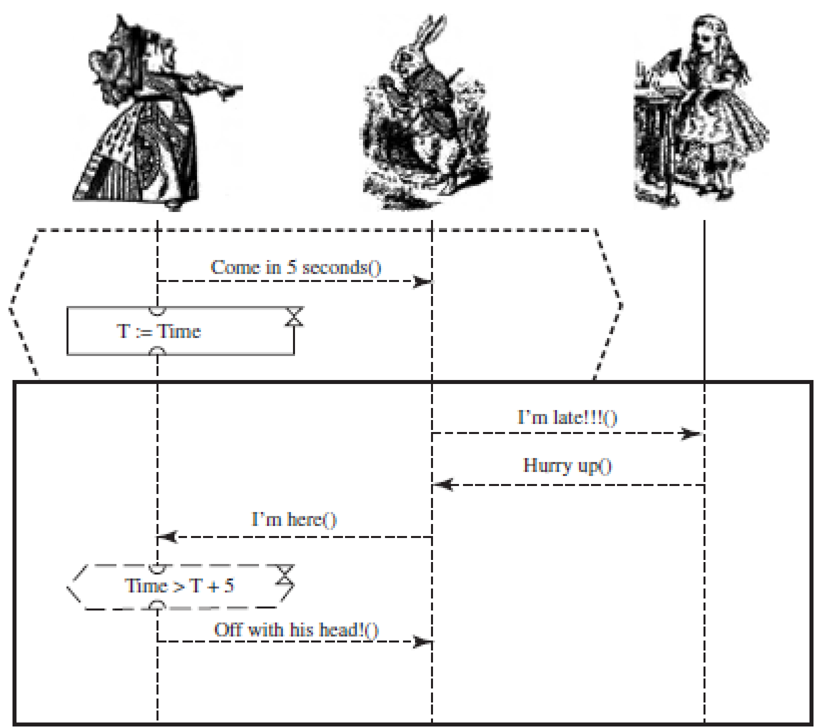
.



Pierwsze trzy zdarzenia znajdują się na wykresie wstępnym, a pozostałe na wykresie głównym. W związku z tym LSC stwierdza, że ilekroć użytkownik kliknie ∗, po którym następuje cyfra, a następnie klawisz Wyślij, musi zostać spełniona reszta scenariusza. (W szczególności, jeśli trzy zdarzenia z wykresu wstępnego nie zostaną zakończone, np. użytkownik nie naciśnie klawisza Wyślij, nic się nie stanie i niczego nie oczekuje się od systemu.) Komunikaty na głównym wykresie są gorące (przedstawione za pomocą ciągłych strzałek , w przeciwieństwie do linii przerywanych na precharte), podobnie jak linie pionowe. W związku z tym musi nastąpić postęp wzdłuż wszystkich linii na głównym wykresie, a wiadomości muszą zostać wysłane i odebrane, aby wykres był satysfakcjonujący. Dodatkowo dodano pętlę, w ramach której chip może wykonać do trzech prób odbioru niezajętego sygnału z otoczenia. Pętla jest kontrolowana przez warunek zimny (linia przerywana): dopóki sygnał jest zajęty, pętla trzyrundowa jest kontynuowana, ale jeśli tak nie jest, pętla zostaje zakończona (co oznacza, że cały wykres jest spełniony). W tym przykładzie wykorzystaliśmy semantykę zimnego warunku: jeśli jest prawdziwy, gdy zostanie osiągnięty podczas działania systemu, to dobrze, ale nawet jeśli jest fałszywy, nic złego się nie dzieje, a wykonanie po prostu przesuwa się w górę o jeden poziom, poza najbardziej wewnętrzny wykres lub podwykres. W przeciwieństwie do tego, stan wysokiej temperatury musi być prawdziwy po osiągnięciu. Jeśli to nieprawda, to bardzo źle rzeczywiście; w rzeczywistości jest to niewybaczalny błąd lub naruszenie, a system musi przerwać. Na przykład dobrym sposobem na określenie scenariusza przeciwnego z wykorzystaniem gorących warunków (np. otwarcie drzwi windy, gdy nie powinno, lub wystrzelenie pocisku, gdy radar nie jest namierzony na celu) jest uwzględnienie całego niepożądanego scenariusza w prechart, po którym następuje główny wykres, który zawiera jeden gorący stan, który zawsze jest fałszywy. (Dlaczego to działa?) LSC obsługują wiele dodatkowych funkcji, które nie zostaną tutaj szczegółowo opisane. Język jest wystarczająco potężny, aby określić większość aspektów zachowań reaktywnych. Rysunek jest częścią pełnej specyfikacji centrali wielotelefonicznej, której nie będziemy dalej wyjaśniać.



Zawiera między innymi symboliczne instancje, które odnoszą się do dowolnego telefonu lub kanału, warunki wiążące, konstrukcje jeśli-to-inaczej, gorące stany i nie tylko. Rysunek pokazuje proste użycie czasu w LSC, w połączeniu z instrukcjami przypisania i zimnym warunkiem.



Prechart pokazuje Królową Kier, która instruuje Białego Królika, aby przybył na swoje miejsce za pięć sekund, a następnie patrzy na zegarek, odnotowując czas. W rezultacie Biały Królik spotyka Alicję i mówi jej, że się spóźnia (wiadomość jest zimna, więc nie musi tego robić, ale może). Alicja pośpiesza Białego Królika, a po przybyciu na miejsce Królowej posłusznie informuje o swoim przybyciu. Królowa następnie sprawdza, czy nie upłynęło więcej niż pięć sekund. Jeśli rzeczywiście tak jest, wydaje rozkaz usunięcia głowy Białemu Królikowi; w przeciwnym razie (tj. jeśli zimny stan jest fałszywy) scenariusz kończy się spokojnie, a Biały Królik pozostaje z nienaruszoną anatomią . . .

Podejście play-in/play-out

Wokół LSC opracowano ostatnio dwuaspektową metodologię, zwaną playin/play-out. Pozwala użytkownikowi wygodnie określić zachowanie międzyobiektowe w oparciu o scenariusz, a następnie wykonać je bezpośrednio. Tak więc jest to tak naprawdę tylko sposób na zaprogramowanie systemu za pomocą LSC, a następnie uruchomienie programu. Pierwsza technika polega na przyjaznym dla użytkownika sposobie „zagrywania" zachowania bezpośrednio z GUI systemu (lub jego abstrakcyjnej wersji, takiej jak diagram modelu obiektowego), podczas którego LSC są generowane automatycznie. Druga technika umożliwia „odtworzenie" zachowania, to znaczy wykonanie systemu jako ograniczonego przez sumę informacji opartych na scenariuszu, symulując w ten sposób zachowanie systemu dokładnie tak, jak gdyby zostało określone w konwencjonalnym stanie. w oparciu o modę wewnątrzobiektową. Techniki te są obsługiwane przez narzędzie zwane Play-Engine. Główną ideą procesu play-in jest podniesienie poziomu abstrakcji w specyfikacji behawioralnej oraz praca z podobną wersją opracowywanego systemu nie tylko przy uruchamianiu modelu, ale także przy jego przygotowaniu. Dzięki temu osoby, które nie są zaznajomione z LSC lub nie chcą bezpośrednio pracować z takimi językami formalnymi, mogą określić wymagania behawioralne systemów za pomocą wysokopoziomowego i intuicyjnego mechanizmu. Mogą to być eksperci dziedzinowi, inżynierowie aplikacji, inżynierowie wymagań, a nawet potencjalni użytkownicy. „play-in" oznacza, że programista systemu najpierw buduje GUI systemu, bez wbudowanego zachowania, ale z podziałem na obiekty i

ich podstawowe izolowane możliwości. Na przykład przełącznik ma możliwość włączenia lub wyłączenia, a wyświetlacz kalkulatora jest określony jako zdolny do pokazania dowolnej sekwencji do, powiedzmy, 10 znaków. W przypadku systemów bez GUI lub zestawów obiektów wewnętrznych możemy po prostu użyć reprezentacji strukturalnej, takiej jak diagram modelu obiektowego jako GUI. W każdym razie użytkownik „odtwarza" GUI, klikając przyciski, obracając pokrętła i wysyłając wiadomości do obiektów w intuicyjny sposób przeciągnij i upuść. W podobny sposób grając w GUI, często używając myszki do wyboru spośród możliwości, użytkownik opisuje pożądane reakcje systemu i warunki, które mogą lub muszą wystąpić. W tym czasie odpowiednie LSC są konstruowane automatycznie. Pożądane modalności konstruowanego wykresu i jego elementów (uniwersalne/egzystencjalne, gorące/zimne) mogą być wybrane w procesie. Podczas odtwarzania użytkownik po prostu odtwarza aplikację GUI, tak jak zrobiłby to podczas wykonywania konwencjonalnego modelu opartego na stanie wewnątrz obiektu, ale ogranicza się do działań użytkownika końcowego i środowiska zewnętrznego. Proces rozgrywania wymaga, aby silnik gry monitorował odpowiednie premapy wszystkich uniwersalnych wykresów, a jeśli zakończy się pomyślnie, wykona ich główne wykresy, wypatrując naruszeń. Podstawowy mechanizm można przyrównać do nadmiernie posłusznego obywatela, który przez cały czas kręci się i trzyma Wielką Księgę Zasad. Tacy ludzie nie robią nic, chyba że zostaną o to poproszeni, i nigdy nie robią niczego, jeśli narusza to inną zasadę. Aby to osiągnąć, przez cały czas skanują i monitorują wszystkie reguły, a po wykonaniu dowolnej czynności (np. uniesienie palca) wielokrotnie dokonują wymaganych konsekwencji. Oczywiście, w ten sposób można dokonać wyborów i odkryć niespójności w przepisach. Silniejszym sposobem na wykonanie LSC jest użycie techniki zwanej smart play-out, w której wykorzystuje się zaawansowane techniki sprawdzania modelu z weryfikacji programu do obliczenia najlepszego sposobu, w jaki system ma reagować na działanie użytkownika, unikając w ten sposób wielu pułapki naiwnej egzekucji. Szczegóły jednak nie zostaną tutaj opisane. Istnieją dwa sposoby wykorzystania możliwości wykonywania LSC. Pierwszym z nich jest postrzeganie LSC jako wzbogacających konwencjonalny cykl rozwoju systemu, a drugim jest używanie ich jako samej możliwej do wdrożenia specyfikacji behawioralnej, która może prowadzić do nowego rodzaju cyklu rozwojowego. W pierwszym podejściu wykonywalne zachowanie oparte na scenariuszach oferuje ulepszenia niektórych standardowych etapów rozwoju systemu: wygodniejsze przechwytywanie wymagań, możliwość określenia bardziej zaawansowanych wymagań behawioralnych, sposób wykonywania bogatych przypadków użycia, narzędzia do dynamicznego testowania wymagania przed zbudowaniem rzeczywistego modelu systemu lub implementacją oraz środki do testowania systemów poprzez dynamiczne porównanie dwóch plików wykonywalnych z podwójnym widokiem w czasie wykonywania. Drugie podejście jest jednak bardziej radykalne. Wymaga rozważenia możliwości alternatywnego sposobu projektowania rzeczywistego zachowania systemu, który ma charakter scenariuszowy i międzyobiektowy. Ten pogląd proponuje ideę, że LSC (lub inny porównywalny język międzyobiektowy, taki jak diagramy czasowe lub logika temporalna) mogą faktycznie stanowić implementację systemu. Mechanizm play-out stanowiłby wówczas rodzaj „uniwersalnego systemu reaktywnego", który wykonuje zbiór LSC w sposób interpretacyjny, jakby była to konwencjonalna implementacja. Z tego punktu widzenia specyfikacja behawioralna systemu reaktywnego nie musiałaby wiązać się z dużym modelowaniem wewnątrzobiektowym (np. za pomocą maszyn stanów, wykresów stanów lub kodu). Ten pomysł jest wciąż dość wstępny, ale wydaje się obiecujący, ponieważ zachowanie oparte na scenariuszach jest sposobem, w jaki większość ludzi myśli o reaktywności. Kiedy stanie się możliwe uchwycenie tego myślenia w naturalny sposób i bezpośrednie wykonanie go, będziemy mieli środki do określenia możliwego do wdrożenia zachowania naszych systemów, które jest dobrze dopasowane do sposobu, w jaki o nim myślimy. A to może mieć wpływ na jakość, niezawodność i szybkość rozwoju złożonego systemu.

Opracowywanie systemów czasu rzeczywistego

Niektóre systemy reaktywne, zwane systemami czasu rzeczywistego, charakteryzują się tym, że czas odgrywa kluczową rolę w ich zachowaniu. Nie wystarczy, aby taki system prawidłowo reagował na bodźce, musi to robić w ściśle określonych terminach. Na przykład system zapobiegania kolizjom statku powietrznego musi wcześnie wydać ostrzeżenie , aby dać pilotowi czas na podjęcie działań mających na celu uniknięcie kolizji, a system przeciwhamulcowy w samochodzie musi działać wystarczająco szybko, aby zapobiec wypadkowi. Podobnie, system antyrakietowy musi zidentyfikować trajektorię nadlatującego pocisku na czas, aby wystrzelić własną broń przechwytującą. Tworzenie systemów czasu rzeczywistego jest szczególnie trudne, ponieważ rzeczywisty czas każdego elementu systemu musi być uwzględniony w obliczeniach jego czasów reakcji. W szczególności konwencjonalne systemy operacyjne są zwykle nieodpowiednie do wdrażania systemów czasu rzeczywistego, ponieważ nie zapewniają niezbędnych gwarancji czasu. W tym celu istnieją specjalne systemy operacyjne czasu rzeczywistego; zazwyczaj oferują one bardziej ograniczone usługi niż ich odpowiedniki ogólnego przeznaczenia, ale gwarantują ścisły czas reakcji. Określanie zachowania systemów czasu rzeczywistego jest również szczególnie trudne, ponieważ oprócz kwestii reaktywności i konwencjonalnych obliczeń, muszą one przestrzegać krytycznych ograniczeń czasowych. Chociaż wiele podejść do reaktywnej specyfikacji systemu może również dotyczyć aspektów czasu rzeczywistego, ostatnia propozycja jest dostosowana specjalnie do reaktywności zależnej od czasu. Nazywa się MASS, akronimem języka specyfikacji schematu aktywacji marionetek. Metafora marionetki sugeruje oddzielenie mechanizmu aktywacji od działań marionetek. W MASS niereaktywne aspekty systemu można określić przy użyciu dowolnego z wielu proponowanych formalizmów dla programów sekwencyjnych lub systemów transformacyjnych. Aspekty reaktywne, szczególnie te zależne od czasu, są ujmowane przez zestaw reakcji, z których każda określa reakcję systemu na jakieś zdarzenie, z możliwym ograniczeniem czasowym. Na przykład reakcja

[Włącz > Włącz → Aktywuj piec] < 2 s

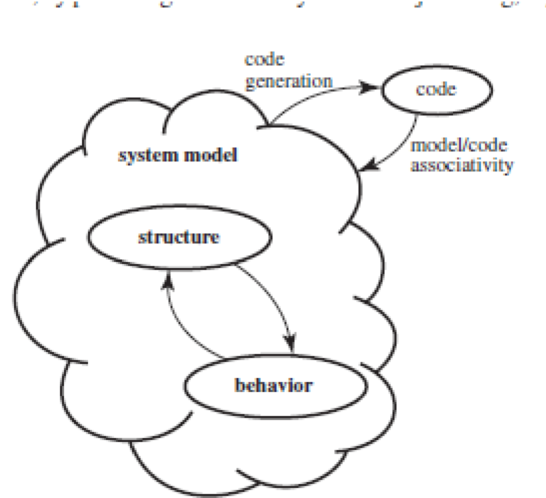
oznacza, że w ciągu dwóch sekund od momentu włączenia przełącznika zadanie Aktywuj piec musi zostać zakończone. Szczegóły tego, co dokładnie jest zaangażowane w to zadanie, są określone osobno, ponieważ nie współdziała z żadną inną czynnością w systemie. Reakcje mogą również mieć przerwane zdarzenia; na przykład reakcja [Train-out→Gate(Open)] : Train-in < 15sec określa, że brama musi zakończyć otwieranie nie później niż 15 sekund od momentu, gdy pociąg wyjechał z przejazdu kolejowego; jeśli jednak inny pociąg wjedzie na skrzyżowanie przed zakończeniem tego zadania, jest on przerywany. Inna reakcja, taka jak [Train-in→Gate(Close)] < 10sek określa, że brama powinna zakończyć zamykanie w ciągu 10 sekund od zdarzenia Train-in.

Tutaj pierwsza reakcja musi zostać przerwana, w przeciwnym razie wystąpią sprzeczne wymagania. Również w tym przykładzie szczegóły dotyczące otwierania i zamykania bramy mogą być określone innymi środkami i mogą być zaimplementowane w dowolnym języku programowania, dla którego można wyprowadzić ograniczenia czasowe; w praktyce ogranicza to wybór do języków asemblerowych lub języków niskiego poziomu, takich jak C. W ten sposób MASS pozwala projektantowi systemu czasu rzeczywistego skoncentrować się na aspektach aktywacji systemu w czasie rzeczywistym bez martwienia się o specyfikę aktywowane zadania w tym samym czasie.

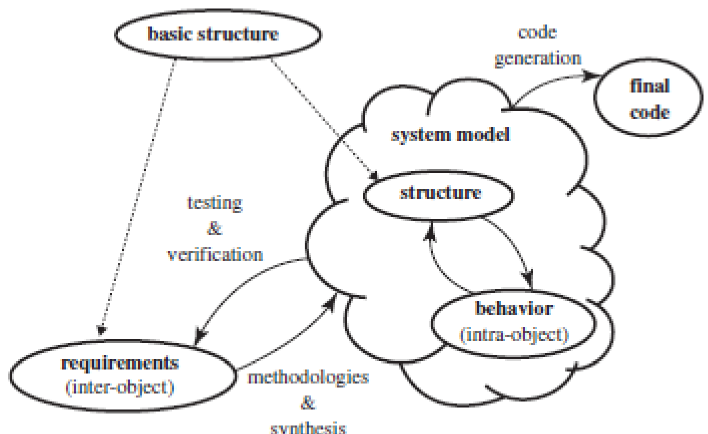
Badania nad systemami reaktywnymi

Większość tematów poruszanych w części poświęconej badaniom w poprzedniej części dotyczy wszelkiego rodzaju dużych systemów komputerowych, przy czym szczególny przypadek systemów reaktywnych nie stanowi wyjątku. To samo dotyczy tematów badawczych dotyczących poprawności i weryfikacji z Części 5. Jeśli już, wiele problemów staje się bardziej dotkliwych, gdy systemy są

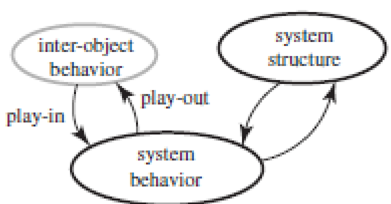
reaktywne, a zwłaszcza gdy mają rygorystyczne ograniczenia czasu rzeczywistego. Ponadto prowadzi się wiele badań nad formalizmami wizualnymi, ich wygodą i siłą wyrazu oraz ich implementacją i analizą. Jednym z godnych uwagi wysiłków jest zunifikowany język modelowania, UML, który rzekomo zebrać pod jednym dachem wiele powiązanych ze sobą schematycznych notacji dla rozwoju systemu (niekoniecznie systemów reaktywnych). Sercem behawioralnym UML jest zorientowana obiektowo wersja schematów stanów. MSC są również częścią UML i są używane do określania wymagań i sekwencji testowych pod nazwą diagramów sekwencji. UML jest oficjalnym standardem rozwoju systemu, koordynowanym i wydawanym przez Object Management Group. Jest to ciągły wysiłek, z okresowymi zaproszeniami do składania wniosków i rozszerzeniami, a od czasu do czasu wydawane są nowe wersje normy. Zespół UML podjął szeroko zakrojone próby zdefiniowania i zapisania znaczeń różnych języków, które go tworzą, oraz ich wzajemnych powiązań, ale wysiłki te są nieformalne i często niekompletne. Innym problemem, jaki niektórzy ludzie mają z UML, jest jego rozległy zakres, obejmujący wiele różnych języków używanych do wielu różnych celów. W szczególności UML udostępnia kilka różnych języków do określania reaktywnego zachowania, które można łatwo wykorzystać do nieumyślnego określenia rzeczy więcej niż raz na różne sposoby. Rodzi to subtelne kwestie spójności, które nie zostały jeszcze odpowiednio rozwiązane. Ponieważ UML jest akceptowanym standardem, i prawdopodobnie będzie tylko rósł w użyciu, semantycy są zajęci pracą, próbując zdefiniować niektóre z bardziej centralnych części UML w rygorystyczny sposób, czyniąc go podatnym na analizę komputerową i wymyślając sposoby zapewnienia spójności modeli UML. Weryfikacja systemów reaktywnych jest tematem szeroko zakrojonych prac, a niektóre techniki weryfikacji zostały przystosowane do pracy z wizualnymi modelami reaktywnymi i są włączane do narzędzi programistycznych na skalę przemysłową. Według wszelkiego prawdopodobieństwa trend ten będzie się utrzymywał, a przewidywalna przyszłość powinna nieść ze sobą możliwość automatycznej weryfikacji pewnych krytycznych właściwości złożonych systemów. Relacje między zachowaniem międzyobiektowym i wewnątrzobiektowym oraz pojawienie się języków takich jak LSC, rodzą wiele problemów związanych z różnicą między wymaganiami dotyczącymi zachowania a konwencjonalnym zachowaniem możliwym do wdrożenia. Podczas gdy weryfikacja dotyczy sprawdzenia, czy to pierwsze jest prawdziwe w odniesieniu do drugiego, inny temat badań dotyczy syntezy tego drugiego z pierwszym. Zobacz Rysunek, który pokazuje model systemu z rysunku



po prawej, ponieważ odnosi się do wymagań po lewej stronie.



Relacje polegają na (1) upewnieniu się, że to drugie obowiązuje dla pierwszego (poprzez testowanie i weryfikację) oraz (2) konstruowaniu pierwszego z drugiego (poprzez metodologie i syntezę). Na przykład byłoby miło, gdybyśmy mogli zapewnić wydajny algorytm do syntezy zwartych wykresów stanu z LSC. Niestety, złożoność najgorszego przypadku większości wersji problemu syntezy dla systemów skończonych jest bardzo zła - przynajmniej w czasie wykładniczym. Dobrą wiadomością jest jednak to, że chociaż to samo dotyczy problemu z weryfikacją takich systemów, nie przeszkodziło to w opracowaniu niezwykle przydatnych narzędzi weryfikacyjnych. W każdym razie synteza jest również tematem wielu badań. Rysunek jest modyfikacją rysunku, mającą na celu zilustrowanie możliwości wykorzystania odtwarzanego zachowania międzyobiektowego jako rzeczywistej implementacji systemu reaktywnego.



Pomysł ten wymaga również dalszych badań, takich jak: opracowanie metodologii i heurystyk pozwalających ustalić, jakie rodzaje systemów byłyby najbardziej podatne na podejście; opracowanie kryteriów i wytycznych dla ustalenia kompletności takiej specyfikacji; oraz opracowanie algorytmów do określania wewnętrznej spójności specyfikacji opartej na scenariuszu oraz równoważności takiej specyfikacji z konwencjonalną. Ponadto, inteligentna rozgrywka wydaje się być interesującą linią przyszłej pracy, w której techniki weryfikacji są wykorzystywane nie do udowodnienia czegoś na temat modelu lub programu, ale do faktycznej pomocy w jego uruchomieniu w sposób, który pozwala uniknąć naruszeń i pułapek, przyczyniając się w ten sposób do „prawidłowego" ich wykonania. Wydaje się, że te i inne pokrewne tematy będą zajmować badaczy w tej dziedzinie już od dłuższego czasu. Wreszcie, kuszący nowy obszar badań obejmuje wykorzystanie technik, języków i narzędzi do reaktywnego rozwoju systemów w celu modelowania natury. Wydaje się, że wiele rodzajów układów biologicznych wykazuje reaktywność w dużym stopniu i na wielu poziomach szczegółowości, w tym na poziomie molekularnym i komórkowym, a także na poziomie całego organizmu. W ciągu ostatnich kilku lat nastąpił gwałtowny wzrost prac w tej dziedzinie i możliwe, że zaczniemy oglądać skomplikowane modele złożonych systemów biologicznych zbudowanych przy użyciu technik wywodzących się z informatyki. Takie modele będą wykorzystywane nie tylko do wspomagania biologów w wizualizacji i

zrozumieniu systemów biologicznych w animowanym zachowaniu, ale także do odkrywania luk lub błędów w wiedzy biologicznej, a nawet do przyczynienia się do rzeczywistych odkryć poprzez przewidywanie, które będą napędzać eksperymenty laboratoryjne.

Algorytmika i inteligencja

Pytanie, czy komputery potrafią myśleć, jak ktoś kiedyś powiedział, jest tak samo jak pytanie o to, czy okręty podwodne potrafią pływać. Analogia jest całkiem trafna. Chociaż wszyscy mniej więcej wiemy, do czego zdolne są łodzie podwodne – i rzeczywiście potrafią robić rzeczy podobne do pływania – „prawdziwe" pływanie jest czymś, co kojarzymy z bytami natury organicznej, takimi jak ludzie i ryby, a nie z łodziami podwodnymi. Podobnie, chociaż czytelnik musi już mieć całkiem dobre pojęcie o tym, czym jest algorytmika, a co za tym idzie możliwościami komputerów, prawdziwe myślenie kojarzy się w naszych umysłach z ludźmi, a może także z małpami i delfinami, ale nie z kolekcje bitów i bajtów oparte na krzemie. Ten nieco mniej techniczny rozdział dotyczy relacji między obliczeniami maszynowymi a ludzką inteligencją. Problem jest obciążony emocjonalnie i prawie wszystko, co o nim mówi, wywołuje gorące kontrowersje. Ze swojej strony postaramy się jak najlepiej uniknąć tych kontrowersji. Zamiast tego omówimy znaczenie lub nieistotność opisanej tutaj dziedziny algorytmiki dla komputerowej symulacji ludzkiej inteligencji. Wskazywane będą głównie problemy i trudności, a nie rozwiązania. Obszarem badań najbardziej istotnym dla tego rozdziału jest sztuczna inteligencja, w skrócie AI. W większości miejsc sztuczna inteligencja prowadzona jest na wydziałach informatyki, podczas gdy w innych istnieje albo osobny wydział sztucznej inteligencji, albo międzywydziałowa grupa związana być może z informatyką, psychologią poznawczą i neurobiologią. Obecnie na wielu uniwersytetach wszystkie te dyscypliny zostały połączone w centra nauki o mózgu lub kognitywistyki. Fakt, że jest to rozdział zamykający, nie powinien prowadzić czytelnika do wniosku, że jedynym powodem studiowania algorytmiki jest możliwość, że pewnego dnia będziemy w stanie skomputeryzować inteligencję. Jakkolwiek intrygujący może zabrzmieć ten pomysł, zadanie napisania tej książki nie zostało podjęte w celu zrelacjonowania naukowych podstaw sztucznej inteligencji. Algorytmika ma swoje własne zalety i można z łatwością przedstawić mocne argumenty za jej znaczeniem, głębią i stosowalnością, niezależnie od tego, czy rozważany jest temat tego rozdziału, czy nie. Tutaj mamy do czynienia z nowym, „miękkim" wymiarem. Fascynujący i ekscytujący, ale też kontrowersyjny i spekulacyjny. Przede wszystkim, jak zobaczymy, jest zupełnie inaczej. W kategoriach czysto technicznych można by powiedzieć, że niniejszy rozdział koncentruje się na innej swobodzie, którą można wykorzystać przy projektowaniu algorytmów, którą należy dodać do paralelizmu i rzucania monetą, o których mowa w rozdziałach 10 i 11. Ta swoboda obejmuje wprowadzenie zasady praktyczne, wykształcone domysły lub, używając przyjętego terminu, heurystyki. Charakter tej nowej placówki i jej motywujące przykłady odróżniają ją od praktycznie wszystkich dotychczas omawianych zagadnień, dlatego też potraktowano ją jako ostatnią. Podczas gdy heurystyki reprezentują szczególną naturę części sterującej inteligentnych programów, istnieją również trudności związane z przedstawianiem i manipulowaniem odpowiednimi dla nich danymi, a mianowicie wiedzą w jej różnych postaciach.

Niektóre historie robotów

Mówiąc o pływaniu, oto co wydarzyło się w latach 70., kiedy w niektórych kręgach istniało wiele naiwnych emocji wokół możliwości zbudowania naprawdę inteligentnych robotów, w jednym z najbardziej szanowanych ośrodków amerykańskiej informatyki. Delegacja Marynarki Wojennej Stanów Zjednoczonych odwiedziła centrum, aby dowiedzieć się, czy mogliby wykorzystać zgromadzoną tam wiedzę do zbudowania robota, który samodzielnie byłby w stanie nurkować pod statkami i wykonywać prace konserwacyjne w zanurzeniu. Naukowcy z dumą pokazali tym ludziom wyniki ich ostatnich wysiłków w robotyce – skomputeryzowane ramię robota połączone z kamerą wideo, które mogło czytać, rozumieć i wykonywać polecenia, takie jak „zbuduj wieżę z trzech bloków" lub „umieść czerwona piramida na niebieskim bloku." Z grzeczności członkowie delegacji wysłuchali, obserwowali, a następnie grzecznie wyszli, głęboko rozczarowani. Ci ludzie nie mieli pojęcia o niewiarygodnych

trudnościach związanych z osiągnięciem nawet tak przyziemnych zachowań w automatycznym systemie komputerowym. Na pewnym etapie rozwoju oprogramowania sterującego ramię robota próbowało budować wieże z trzech bloków, zaczynając od góry! „Nauczenie" robota grawitacji, nawet w jego bardzo ograniczonym świecie, składającym się z kilku bloków i piramid, nie było sprawą błahą. Inna historia z udziałem tej samej grupy badaczy opowiada o firmie handlowej, która twierdziła, że wyprodukowała robota wykonującego rutynowe czynności porządkowe w odpowiedzi na polecenia wydawane prostym językiem, takie jak „zmyć naczynia" lub „przygotować obiad dla czterech osób". ". Firma miała zademonstrować publicznie możliwości robota w jednym z lokalnych domów towarowych. Naukowcy z grupy, doskonale wiedząc, co można osiągnąć przy obecnym stanie wiedzy, byli pewni, że to oszustwo. Tak więc, podczas gdy wielu widzów, którzy zebrali się, aby zobaczyć cud, było zajętych oglądaniem sceny z przodu, byli zajęci za kulisami, próbując odkryć sztuczkę. Pewnie po chwili znaleźli osobę, która za pomocą nadajnika radiowego fizycznie sterowała ruchami robota, sprawiając wrażenie, że reaguje na polecenia instruktora. Historie te wydają się zaprzeczać stwierdzeniu zawartemu w Części 1, co oznacza, że komputery mogą sterować niezwykle wyrafinowanymi robotami przemysłowymi, które konstruują złożone elementy składające się z wielu komponentów. Nie ma sprzeczności. Roboty te są zaprogramowane do wykonywania długich i skomplikowanych sekwencji operacji według starannie przygotowanej receptury. Ogólnie rzecz biorąc, można je przeprogramować do wykonywania różnych zadań, a czasami są w stanie dostosować swoje zachowanie, aby dostosować się do zmieniających się sytuacji. Nie są jednak w stanie spojrzeć na swoje otoczenie, zdecydować, co należy zrobić, a następnie ułożyć plan i wykonać go do końca. Z tego powodu nikt nie wie, jak zaprogramować robota do budowy ptasiego gniazda ze sterty gałązek. Odnotowano pewne sukcesy w radzeniu sobie z bardzo ograniczonymi światami bloków i piramid, ale nie z gałązkami o różnych kształtach i rozmiarach lub z dużą i różnorodną gamą elementów maszyn. Radzenie sobie z nimi wymaga poziomów inteligencji, które znacznie przekraczają dzisiejsze możliwości. Nawet umiejętność uchwycenia prostej sceny, takiej jak normalny salon (przy użyciu jakiegoś wizualnego sprzętu sensorycznego) i „zrozumienia" jej, coś, co każde dziecko może zrobić, wykracza daleko poza obecne możliwości. Inny przykład podany w rozdziale 1 dotyczył kontrastu między tomografią komputerową (syntetyzującą przekrój mózgu z dużej liczby zdjęć rentgenowskich wykonanych pod różnymi kątami) a możliwością wywnioskowania wieku osoby ze zwykłego zdjęcia. Tutaj też nie ma sprzeczności. Podczas gdy pierwsze zadanie realizowane jest za pomocą skomplikowanych, ale dobrze zdefiniowanych procedur algorytmicznych, drugie wymaga prawdziwej inteligencji. Komputeryzacja inteligencji, uczynienie jej algorytmiczną, to coś, o czym wiemy zdecydowanie za mało.

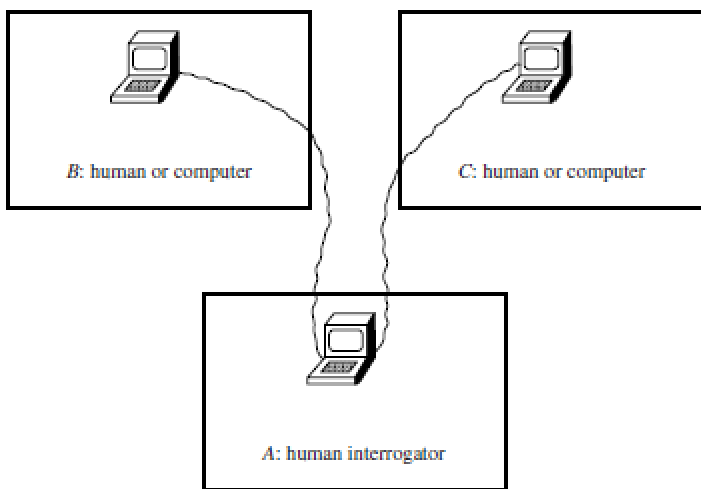
Inteligencja algorytmiczna?

Czym jest inteligencja? Konkurując z filozofami, psychologami poznawczymi i badaczami sztucznej inteligencji w tej kwestii, nic nie można zyskać. Jednak z punktu widzenia laika wydaje się, że samo pojęcie sztucznej inteligencji, czy też, żeby zmienić nazwę, by pasowało do reszty książki, inteligencja algorytmiczna,jest sprzecznością w kategoriach. Mamy tendencję do postrzegania inteligencji jako naszej podstawowej nieprogramowalnej, a zatem niealgorytmicznej cechy. Wielu ludziom sama idea inteligentnej maszyny nie brzmi dobrze. Przedstawiono różne argumenty, aby uczynić nie do pomyślenia koncepcję inteligentnej maszyny myślącej. Niektórzy twierdzą, że myślenie musi koniecznie obejmować emocje i uczucia, a żaden komputer nie może nienawidzić, kochać ani wpadać w złość. Inni twierdzą, że inteligentne myślenie z konieczności pociąga za sobą oryginalność i żaden komputer nie może niczego stworzyć, jeśli nie zostanie zaprogramowany do tego z wyprzedzeniem, w którym to przypadku nie jest już oryginalny. Z tego punktu widzenia komputera nigdy nie można nazwać „inteligentnym". Z drugiej strony wiele osób uważa, że ludzki mózg sam w sobie jest tylko maszyną, choć skomplikowaną. Tak więc, zgodnie z tezą Churcha/Turinga (zob. rozdział 9) komputer elektroniczny może w zasadzie symulować ludzki umysł i działać tak, jakby był inteligentny. Nazywa się

to słabym twierdzeniem AI. Idąc krok dalej, zwolennicy silnej sztucznej inteligencji twierdzą, że taki komputer byłby naprawdę świadomy. Więcej o tej debacie powiemy poniżej. Wydaje się jednak, że maszyna uważana za inteligentną musi przynajmniej być w stanie wykazywać podobne do ludzkiego zachowania intelektualne. W tym celu nie potrzebujemy go, aby chodzić, widzieć lub mówić jak człowiek, tylko rozumować i odpowiadać jak człowiek. Co więcej, jakiekolwiek okażą się uzgodnione kryteria wywiadu, ktoś powinien być w stanie sprawdzić, czy maszyna-kandydat je spełnia. A kto, jeśli nie prawdziwy, inteligentny człowiek, ma kwalifikacje do przeprowadzenia takiego testu? To prowadzi nas do idei, że maszynę należy nazwać inteligentną, jeśli może przekonać przeciętnego człowieka, że pod względem intelektu nie różni się od innego przeciętnego człowieka.

Test Turinga

Jak możemy ustawić rzeczy, aby taki test był możliwy? W 1950 roku Alan Turing zaproponował następującą metodę, obecnie powszechnie nazywaną testem Turinga. Test odbywa się w trzech salach. W pierwszym jest przesłuchujący człowiek, nazwijmy go Alicją, w drugim drugi człowiek, a w trzecim komputer kandydat. Śledcza Alice zna pozostałą dwójkę tylko pod imionami Bob i Carol, ale nie wie, kto jest człowiekiem, a kto komputerem. Trzy pokoje są wyposażone w terminale komputerowe, a terminal Alicji jest połączony z terminalami Boba i Carol.



Teraz Alicja ma, powiedzmy, godzinę na ustalenie prawidłowej tożsamości Boba i Carol. Alicja może zadawać dowolne pytania lub wypowiedzi każdemu z nich, a komputer musi dołożyć wszelkich starań, aby oszukać Alicję, sprawiając wrażenie człowieka. Mówi się, że komputer zda test, jeśli Alicja nie wie, który z Bobów lub Carol jest naprawdę komputerem po upływie wyznaczonego czasu. (Właściwie wymagamy, aby komputer przeszedł szereg testów w jednej sesji, z różnymi przesłuchującymi, aby zminimalizować możliwość, że Alicja po prostu zgadnie, która jest która).

Spróbujmy wyczuć ogromną trudność z tym związaną. Zastanów się, jak inteligentny program musiałby zareagować na następujące pytania Alicji:

1. Czy jesteś komputerem?

2. Jaka jest godzina?

3. Kiedy zamordowano prezydenta Kennedy'ego?

4. Ile wynosi 2 276 448 × 7 896?

5. Czy białe mogą wygrać jednym ruchem z następującej pozycji w szachach: . . .?

6. Opisz swoich rodziców.

7. Jak uderza cię następujący wiersz: . . .?

8. Co myślisz o Charlesie Dickensie?

9. Jakie jest Twoje zdanie na temat programów eksploracji kosmosu, biorąc pod uwagę fakt, że miliony ludzi na całym świecie cierpią z głodu?

Zauważ, że zaprogramowany komputer musi być w stanie swobodnie rozmawiać w języku naturalnym, takim jak angielski. Rezygnujemy z potrzeby słyszenia i mówienia, a co za tym idzie łączy elektronicznych. Jednak rozumienie i syntetyzowanie języka uważane są za dwa z kamieni węgielnych ludzkiej inteligencji, a zatem są tutaj wymaganym warunkiem wstępnym. Pierwsze dwa pytania są dość proste. Odpowiedź na pierwsze powinno brzmieć „nie", a na drugie program może korzystać z wbudowanego zegara, który jest obecny w większości komputerów. Pytanie (3) wymaga, aby program miał dostęp do bardzo dużej puli wiedzy, porównywalnej z wiedzą człowieka. Pytanie (4) wydaje się proste - komputery wydają się radzą sobie najlepiej z takimi pytaniami - ale mimo to jest subtelne, ponieważ komputer musi chwilę poczekać, zanim odpowie; natychmiastowa odpowiedź zdradziłaby jego tożsamość. Pytanie (5) wymaga oczywiście pewnej wiedzy o szachach, w tym umiejętności rozumowania o nowej pozycji. Chociaż odpowiedzi na pytania podobne do pytania (6) mogą być przygotowane z wyprzedzeniem, Alicja może odpowiedzieć na odpowiedź, podążając za pewnym nieoczekiwanym szczegółem, na który komputer może nie mieć gotowej odpowiedzi. Pytania (7), (8) i (9) są znacznie trudniejsze, ponieważ ilustrują potrzebę okazywania preferencji, opinii, stanowisk i emocji, wyjaśniania ich i odnoszenia ich do innych. Krótko mówiąc, konkurujący komputer musi być obdarzony prawdziwą osobowością. Jedną z trudności w zaprogramowaniu komputera do zdania testu Turinga jest duża różnorodność obszarów, do których może prowadzić rozmowa. Odnotowano pewne widoczne sukcesy w naśladowaniu ludzkiej konwersacji w bardzo ograniczonych obszarach dyskursu. Program o nazwie ELIZA, napisany w 1966 roku, był w stanie dać imponujący występ jako psychoanalityk. Tak imponujące, że ludzie byli znani z tego, że faktycznie „zwierzali się" w programie w sposób, w jaki nigdy nie zwierzyliby się człowiekowi! Oto kilka typowych wymian:

Osoba: Moja żona mówi, że często mam depresję.

ELIZA: Przykro mi słyszeć, że masz depresję.

Osoba: Prawda jest taka, że jestem bardzo nieszczęśliwa.

ELIZA: Myślisz, że przyjazd tutaj pomoże ci nie być nieszczęśliwym?

Osoba: Może rozmawiając z tobą dowiem się, dlaczego nienawidzę swojego brata.

ELIZA: Opowiedz mi więcej o swojej rodzinie.

Interesującą rzeczą dotyczącą ELIZY jest fakt, że nie można powiedzieć, że jest inteligentna w żaden rozsądny sposób. W rzeczywistości, pomimo pozornego wyrafinowania, ELIZA jest naprawdę głupia, więc o ile sprawia wrażenie rozważnej terapeutki, jest naprawdę sprytnym oszustem. Czasami po prostu skupia się na konkretnym słowie lub frazie, na które nauczono go zwracać uwagę, i odpowiada, wybierając, praktycznie losowo, jedną z niewielkiej liczby stałych odpowiedzi. Doskonałym tego przykładem jest odpowiedź „Opowiedz mi więcej o swojej rodzinie" wywołana słowem „brat". Innym razem ELIZA po prostu zamienia nadchodzące zdanie w pytanie lub nieco pozbawione treści oświadczenie, jak w wymianie dotyczącej nieszczęścia danej osoby. Aby to zrobić, wykorzystuje prosty

mechanizm do odgadnięcia gruboziarnistej struktury zdań wejściowych. Dziwne rzeczy dzieją się, gdy naprawdę próbujesz przetestować inteligencję ELIZY, zamiast po prostu wylewać swoje problemy. W rzeczywistości, gdybyśmy powiedzieli: „Byłam siostrą w klasztorze w Birmie", a nawet „Podziwiam siostrę Teresę", program może równie dobrze odpowiedzieć taką samą nieistotną odpowiedzią: „Opowiedz mi więcej o swojej rodzinie". Równie zabawna będzie jego odpowiedź na pytanie o programy eksploracji kosmosu. ELIZA oczywiście nie ma szans na zdanie testu Turinga. W rzeczywistości nie było to zamierzone. Motywacją stojącą za ELIZA było pokazanie, że łatwo jest wyglądać na inteligentnego, przynajmniej na krótką chwilę, sympatycznemu obserwatorowi i w wąskiej dziedzinie dyskursu. Bycie naprawdę inteligentnym to jednak zupełnie inna sprawa. Aby jeszcze bardziej docenić różnicę między prawdziwą inteligencją wymaganą do zdania testu Turinga, a płytką, ale zwodniczo zwodniczą naturą zdolności konwersacyjnych ELIZY, oto hipotetyczna wymiana zdań między przesłuchującą test Turinga Alice i naprawdę inteligentnym kandydatem, powiedzmy Bobem:

Alice: Co to jest zupczok?

Bob: Nie mam pojęcia.

Alice: Zupczok to latający wieloryb piszący powieści. Był starannie hodowany w laboratorium przez kilka pokoleń, aby zapewnić, że jego płetwy ewoluują w podobne do skrzydeł rzeczy, które umożliwiają mu latanie. Stopniowo uczono go także czytać i pisać. Posiada gruntowną wiedzę na temat literatury współczesnej i posiada umiejętność pisania powieści kryminalnych do publikacji.

Bob: Jakie to dziwne!

Alice: Czy myślisz, że zupczoki istnieją?

Bob: Nie ma mowy. Nie mogą.

Alicja: Dlaczego?

Bob: Z wielu powodów. Po pierwsze, nasze możliwości inżynierii genetycznej są dalekie od adekwatnych, jeśli chodzi o zamienianie płetw w skrzydła, nie wspominając o naszej niezdolności do spowodowania, aby 10-tonowe istoty z silnikami przeciwstawiały się grawitacji tylko poprzez machanie tymi rzeczami. Po drugie, część dotycząca pisania powieści nawet nie zasługuje na odpowiedź, ponieważ napisanie dobrej historii wymaga znacznie więcej niż umiejętności technicznych czytania i pisania. Cały pomysł wydaje się dość śmieszny. Nie masz nic ciekawszego do omówienia?

Aby przeprowadzić tę rozmowę, Bob, czy to człowiek, czy komputer, musi wykazywać bardzo wyrafinowane zdolności. Musi posiadać dużą wiedzę na konkretne tematy, takie jak wieloryby, latanie, pisanie powieści i grawitacja. Musi być w stanie nauczyć się zupełnie nowego pojęcia, przyjmując definicje i odnosząc je do tego, co już zna. Wreszcie, musi być w stanie wywnioskować z tego wszystkiego, na przykład fakt, że inżynieria genetyczna ma większe znaczenie dla zupczoków niż, powiedzmy, matematyka czy chińska filozofia. (W tym konkretnym przypadku musi też mieć poczucie humoru.)

Silna sztuczna inteligencja i chiński pokój

Załóżmy, że stworzono program komputerowy, który może przejść test Turinga. Zgodnie z silnym twierdzeniem AI, ten program byłby uważany za inteligentny i świadomy. Słynny eksperyment myślowy, zwany chińskim argumentem pokojowym, próbuje obalić tego rodzaju twierdzenia. Oto jak to działa. Rozważ zmodyfikowaną wersję programu, która rozmawia po chińsku, a nie po angielsku. Zasadniczo jest to zestaw precyzyjnych instrukcji dla komputera i można go przetłumaczyć na serię podobnych instrukcji w języku angielskim, aby człowiek mógł je wykonać. (Byłoby to długie i żmudne,

ale w zasadzie nadal możliwe, a chińskie części wejścia/wyjścia mogą zawierać „głupie" rzeczy, takie jak „jeśli patrzysz na ten chiński znak i kwadrat numer 159 w Twoim notatniku zawiera 1, napisz 0 w kwadracie 243 i przejdź do następnego znaku w prawo.") Przypuśćmy teraz, że książka zawierająca pełny zestaw instrukcji angielskich jest przekazywana osobie, która rozumie i mówi po angielsku, ale nie po chińsku, a następnie zostaje zamknięta w pomieszczenie ze szczeliną do komunikacji ze światem zewnętrznym. Pytania napisane po chińsku są wkładane przez szczelinę, a osoba zamknięta w pokoju postępuje zgodnie z instrukcjami zawartymi w księdze, aby uzyskać odpowiedź, ponownie po chińsku. Ponieważ dana osoba podąża za oryginalnym programem komputerowym, jej odpowiedzi należy uznać za inteligentne zgodnie z testem Turinga. Najwyraźniej jednak osoba przebywająca w tak zwanym chińskim pokoju nie rozumie treści komunikatów, a jedynie postępuje zgodnie z instrukcjami zawartymi w jego książce. Tak więc chińskiego pokoju i jego użytkownika nie można opisać jako świadomych, czy nawet inteligentnych, a zatem oryginalny program też nie! Napisano wiele artykułów i książek, aby wesprzeć jedną lub drugą stronę tej debaty, ale nie będziemy ich tutaj dalej omawiać. Podczas gdy filozofowie debatują i spierają się, badacze sztucznej inteligencji wciąż próbują opracować naprawdę inteligentne programy. Należy jednak zauważyć, że program, który jest w stanie przejść test Turinga, nie rozwiąże problemu; raczej posłuży raczej do jej zintensyfikowania. Gdyby taki program upierał się, że jest świadomy, czy uwierzyłbyś w to? Jeśli o to chodzi, możesz zadać to samo pytanie także ludziom: czy jesteśmy naprawdę inteligentni, czy po prostu symulujemy inteligencję dzięki programowaniu naszych mózgów?

Grać w gry

Test Turinga wydaje się zapewniać wystarczający warunek, aby komputer posiadał pełną ludzką inteligencję. Jednak większość badaczy sztucznej inteligencji nie postawiła sobie za cel pisania programów, które mogą zdać test Turinga, ponieważ obejmuje wiele rzeczy, które nie są bezpośrednio związane z czystą inteligencją. Na przykład, aby zdać test Turinga, komputer musiałby zostać zaprogramowany tak, aby ukrywał swoje nadludzkie umiejętności, jak widać w pytaniu (4) powyżej. Jest to podobne do tego, co wydarzyło się w innych dziedzinach. Na przykład przez wiele lat ludzie próbowali osiągnąć „sztuczny lot", próbując naśladować ptaki. Sukces osiągnięto dzięki zwróceniu się ku zupełnie innym metodom, a nowoczesnym samolotom przypisujemy zdolność latania, nawet jeśli nikogo nie oszukają, by sądził, że są ptakami. Współczesne badania nad sztuczną inteligencją można z grubsza podzielić na analizę i syntezę. Celem pierwszego jest zrozumienie natury inteligencji; często wiąże się to z badaniem tego, w jaki sposób ludzie uczą się, rozumują, dedukują i tworzą plany oraz powielają te umiejętności w programach komputerowych. Ten kierunek badań ma zatem wiele wspólnego z dziedzinami takimi jak psychologia i neurobiologia, oprócz matematyki i algorytmiki. Drugie podejście ma na celu zbudowanie „inteligentnych" programów komputerowych, które mogą wykonywać przydatne zadania. To, jak podobne są do ludzkiej inteligencji, jest mniej ważne. Początkowe prace w latach 50. i 60. miały na celu zbudowanie ogólnych metod rozwiązywania różnych problemów szczegółowych. Stwierdzono, że jest to znacznie trudniejsze niż początkowo sądzono, a później prace skierowano do węższych dziedzin, aby wykorzystać specjalistyczną wiedzę związaną z określonymi tematami. Granie w gry to jeden z wyspecjalizowanych obszarów, w których badania nad sztuczną inteligencją osiągnęły znaczące wyniki. Wydaje się właściwe rozpoczęcie dyskusji od krótkiego biuletynu informacyjnego.

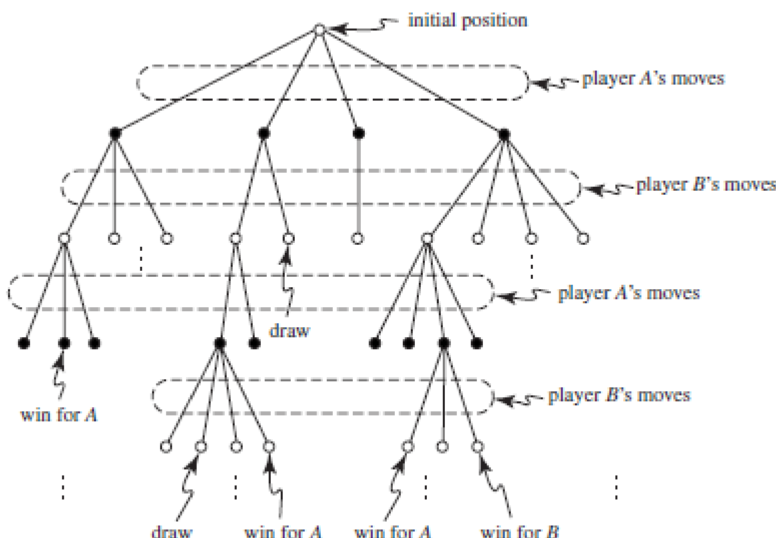
* W 1979 roku program komputerowy pokonał mistrza świata w tryktraku. (To nie uczyniło programu nowym mistrzem, ponieważ gra nie była rozgrywana w oficjalnym turnieju, ale mimo wszystko wygrana była wygraną).

* W 1994 roku mistrz świata w warcaby Marion Tinsley zrezygnował z tytułu na rzecz programu komputerowego Chinook. Ten program wygrał mistrzostwa Man-Machine Checkers Championship,

które odbywają się między najlepszym graczem człowiekiem a najlepszym graczem programu. (Ten specjalny tytuł został stworzony, aby można było mieć zarówno mistrza świata ludzi, jak i komputerowego mistrza świata.)

* W 1997 roku mistrz świata w szachach, Gary Kasparow, przegrał z komputerem Deep Blue od 3,5 do 2,5 w sześciu grach. Ten mecz nastąpił po wygranym 4-2 przez Kasparowa w 1996 roku. Na początku 2003 roku Kasparow zebrał rundę i zremisował 3-3 z Deep Junior, programem, który zdobył tytuł mistrza świata w szachach komputerowych w 2002 roku.

Chociaż programy mogą teraz grać w doskonałe gry w tryktrak, warcaby i szachy, nadal nie są doskonałe. Istnieją inne gry, takie jak Go, w których komputery wcale nie radzą sobie tak dobrze. Dlaczego programy nie zawsze mogą grać w idealną grę? Dlaczego komputer nie może wykonać wszystkich możliwych ruchów i zawsze wykonać najlepszy? Odpowiedź tkwi w drzewach łownych, o których wspomniano w Części 2.



Na przykład w kółko i krzyżyk (kółko i krzyżyk) nie ma trudności. Pierwszy gracz ma dziewięć możliwych ruchów, na które jego przeciwnik może odpowiedzieć na jeden z ośmiu sposobów, na który pierwszy może odpowiedzieć na jeden z siedmiu itd. Drzewo gry składa się zatem z korzenia z dziewięcioma potomstwem, z których każdy ma osiem potomstwo i tak dalej. Niektóre węzły w tym drzewie są terminalami, co oznacza, że reprezentują albo wygraną jednego z graczy, albo pełną planszę bez wygranej. W każdym razie dowolna sekwencja dziewięciu ruchów prowadzi do węzła końcowego. Drzewo ma zatem maksymalną głębokość 9, z maksymalnym stopniem zewnętrznym 9 u nasady . W sumie jest nie więcej niż 9!, czyli 362,880 możliwości sprawdzenia; stąd program, który można łatwo napisać, aby wydajnie grać perfekcyjnie w kółko i krzyżyk. Z drugiej strony w szachach historia jest zupełnie inna. Białe mają 20 możliwych pierwszych ruchów, a średnia liczba możliwych następnych ruchów z dowolnej pozycji w szachach wynosi około 35. Tak więc odchylenie drzewa wynosi średnio około 35. Głębokość drzewka to liczba ruchów (dwukrotność liczby rund gry), która z łatwością może osiągnąć 80 lub 100. Oznacza to, że liczba możliwości sprawdzenia w typowej grze może wynosić 35100. W rozdziale 7 przedstawiamy widzieliśmy kilka takich liczb i pamiętamy, że 35100 to wiele, wiele rzędów wielkości większa niż liczba protonów we wszechświecie, czyli liczba nanosekund od Wielkiego Wybuchu. W konsekwencji, nawet jeśli zignorujemy księgowość i pamięć zaangażowaną w brutalną podróż przez wszystkie możliwe ruchy i założymy, że każdy z nich można przetestować w ciągu

jednej nanosekundy, żaden program nigdy nie zagra w perfekcyjne szachy. Liczby dla warcabów nie są aż tak duże, ale idealne warcaby również nie wchodzą w rachubę. Liczby dla Go są nawet wyższe niż dla szachów; na przykład liczba możliwych ruchów w każdej pozycji wynosi zwykle około 200, a nie 35, a gra trwa około 300 ruchów! Jak zatem działają dobre programy szachowe? Cóż, rzeczywiście przeprowadzają masowe przeszukiwanie dużych części drzewa gry, ale używają także heurystyk lub reguł kciuka. W kontekście gier heurystyka jest używana do pomocy w podjęciu decyzji, które części drzewa gry będą brane pod uwagę przy próbie wybrania dobrego ruchu. Nietypowe wyszukiwanie heurystyczne wykorzystuje intuicyjne reguły wprowadzone do programu przez programistę, aby zignorować pewne części drzewa gry. Na przykład można zdecydować, że jeśli w ciągu ostatnich pięciu ruchów nic się nie zmieniło w czterokwadratowym sąsiedztwie pewnego pionka, to ten pionek nie zostanie przesunięty, a zatem wyszukiwanie może zignorować wszystkie części drzewa znajdujące się poniżej odpowiedni węzeł. Taka zasada może okazać się bardzo wnikliwa — z pewnością skutkuje rozważeniem mniejszych drzew — ale oczywiście może to kosztować grę; Gary Kasparow mógł awansować tego samego pionka, aby wygrać w trzech ruchach. Jest to bardzo prosty przykład, a heurystyki zawarte w prawdziwych programach do gry w szachy są zwykle znacznie bardziej wyrafinowane. Niemniej jednak są to heurystyki, a ich użycie zwiększa prawdopodobieństwo, że możemy przegapić najlepszy ruch.

Więcej o heurystyce

Dobrym sposobem na wyjaśnienie natury wyszukiwania heurystycznego jest rozważenie osoby, która zgubiła soczewkę kontaktową. Jedną z możliwości jest przeprowadzenie wyszukiwania na ślepo, pochylając się i obmacując obiektyw w sposób przypadkowy. Inną możliwością jest przeszukanie systematyczne, polegające na ciągłym poszerzaniu przeszukiwanego obszaru, w sposób metodyczny i zorganizowany (powiedzmy, w coraz większych kręgach wokół punktu centralnego). To poszukiwanie w końcu się powiedzie, ale może być bardzo czasochłonne. Trzecią możliwością jest poszukiwanie analityczne, w którym obliczane są dokładne równania matematyczne rządzące opadaniem soczewki, biorąc pod uwagę wiatr, grawitację, tarcie powietrza i dokładną topografię. , napięcie i fakturę powierzchni. To również, jeśli zostanie zrobione poprawnie, gwarantuje sukces, ale z oczywistych powodów jest to niepraktyczne. W przeciwieństwie do tych metod, większość z nas podchodziłaby do problemu za pomocą wyszukiwania heurystycznego. Najpierw określilibyśmy przybliżony kierunek upadku i zgadywali, jaką odległość soczewka mogła pokonać, spadając, a następnie ograniczylibyśmy poszukiwania do wynikowego obszaru. Tutaj oczywiście wyszukiwanie może się nie powieść, ale wydaje się, że jest całkiem spora szansa, że się uda. (Istnieje oczywiście piąte podejście, leniwe wyszukiwanie, które wymaga znalezienia najbliższego optyka i wykonania nowego obiektywu…) Główną wadą heurystyk jest to, że nie gwarantują sukcesu. Zawsze jest możliwe, że w konkretnym przypadku zawiedzie praktyczna reguła. Jeśli chodzi o korzyści, oprócz możliwości drastycznego skrócenia czasu pracy, heurystyki są zwykle podatne na ulepszenia lub wymianę, gdy lepiej poznajemy dany problem i sposoby, w jakie ludzie go radzą. Jednak w kontekście algorytmiki najważniejszą cechą heurystyki jest to, że jej wykonanie nie podlega analizie. Możemy zdecydować się na heurystykę szachową, włączyć ją do naszego programu gry w szachy i od tej pory będziemy mogli oceniać jej wykonanie tylko przez obserwację. Może rozegrać 100 doskonałych gier, zanim odkryjemy, że ma poważną słabość, która kiedyś znana przeciwnikowi dramatycznie pogarsza jego grę. W pewnym sensie stosowanie heurystyki przypomina trochę rzucanie monetami, ponieważ niekoniecznie uwzględniamy wszystkie możliwości i w konsekwencji możemy przegapić dobre rozwiązanie. W rozdziale 11 zobaczyliśmy, jak piły można ulepszyć, podążając za kaprysami losowych rzutów monetą. Innymi słowy, przestrzeń poszukiwań wszystkich możliwości została zmniejszona, a niektóre kierunki pozostały niezbadane. Byliśmy więc skłonni nazwać liczbę „pierwszą”, chociaż nie sprawdziliśmy wszystkich możliwych świadków jej niepierwotności. Również tam sukces nie jest gwarantowany; stąd kuszące

jest, aby postrzegać rzucanie monetą jako ślepą heurystykę, rodzaj pozbawionej intuicji zasady kciuka. Jest jednak zasadnicza różnica. W dziedzinie algorytmów probabilistycznych analiza zastępuje intuicję. Stosując starannie zdefiniowane zestawy możliwych do zignorowania możliwości i stosując randomizację, aby zdecydować, które faktycznie zignorować, jesteśmy w stanie przeanalizować prawdopodobieństwo wystąpienia rygorystycznego sukcesu, składanie precyzyjnych oświadczeń na temat wydajności algorytmu. Z heurystykami zazwyczaj nie możemy.

Chociaż probabilistyczny algorytm testowania pierwszości (a właściwie każdy algorytm, nawet prosta procedura sortowania) zdecydowanie wygląda na inteligentny — w rzeczywistości robi to znacznie lepiej niż przeciętny człowiek — nie uważamy go za naprawdę inteligentny . Chociaż jego konstrukcja mogła wymagać intuicyjnej pomysłowości ze strony projektanta, jego działanie nie opiera się na intuicji i można je wytłumaczyć analitycznie. z kciuka, których wyników nie możemy przewidzieć ani przeanalizować. Jest to zatem jedna z możliwości interpretacji naszego poczucia, że prawdziwa inteligencja jest nieprogramowalna; po prostu zamień nonprogrammable na nonanalyzable. AI charakteryzuje się programami opartymi na regułach, które wydają się pomocne, ale których przydatność nie została dokładnie przeanalizowana. Ta prymitywna próba zdefiniowania sztucznej inteligencji przez jakąś właściwość powstałych programów nie jest całkiem sprawiedliwa. Większość ludzi wolałaby zdefiniować to tematycznie. Co więcej, niektóre kierunki badań nad sztuczną inteligencją stały się ostatnio dość precyzyjne i analityczne. W wielu przypadkach stosowane heurystyki są czymś więcej niż tylko wykształconymi domysłami; w rzeczywistości opierają się na dobrze zdefiniowanych modelach matematycznych i wzorach. Na przykład w skomputeryzowanym widzeniu heurystyka stosowana do wykrywania ruchu i rozumienia stereoskopowych par obrazów jest oparta na skomplikowanej matematyce i formułuje się mocno poparte przypuszczenia dotyczące losowego zachowania się programów wynikowych wejścia.

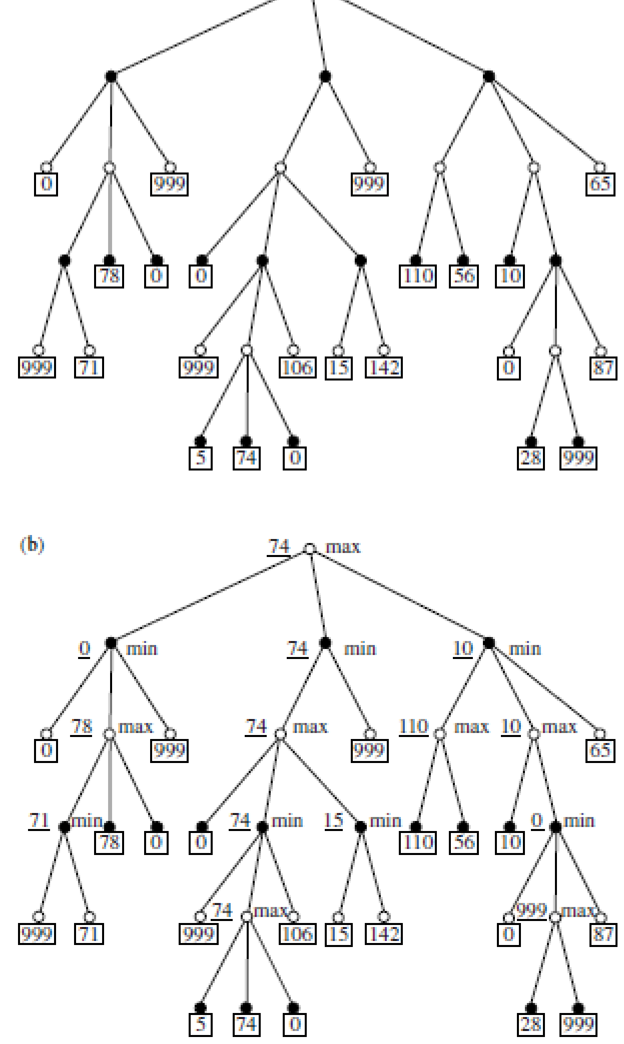
To samo można powiedzieć o planowaniu ruchu w robotyce. W takich przypadkach algorytmy heurystyczne są bliższe algorytmom aproksymacyjnym z Części 7 niż algorytmom probabilistycznym, ponieważ stosowane heurystyki są zwykle domniemane albo w celu zagwarantowania średnio dobrego rozwiązania, albo niezwykle rzadko dają złe rozwiązanie. Możemy zatem nazwać otrzymane algorytmy algorytmami aproksymacji hipotetycznej — brakuje tylko dowodu własności hipotetycznej. Oczywiście po znalezieniu dowodu nie spodziewalibyśmy się, że dyscyplina AI zrezygnuje z algorytmu tylko dlatego, że jego zachowanie zostało dokładnie przeanalizowane

Ocena i wyszukiwanie

Opis heurystyk w skomputeryzowanych szachach również był zbyt uproszczony. W rzeczywistości dzieje się o wiele więcej niż kilka prostych zasad, które powodują, że program ignoruje części drzewa gry. Na przykład musi istnieć sposób oceny jakości pozycji w drzewie. Jako prosty przykład rozważmy węzeł, dla którego zdecydowano zignorować wszystkie z wyjątkiem dwóch możliwych następnych ruchów białych. Załóżmy, że pierwszy z nich może ostatecznie doprowadzić do 10 węzłów końcowych, z których trzy reprezentują wygraną białych, jeden wygraną czarnych, a sześć remisów, a w pozostałych liczbach nie będzie 10, 3, 1 i 6, ale powiedzmy 8, 4, 3 i 1. Jak powinien być program (który gra Białymi) porównać te sytuacje, aby wybrać następny ruch? Problem staje się znacznie bardziej dotkliwy, gdy nawet w rozważanych kierunkach chcemy przerwać poszukiwania na pewnej głębokości. W takim przypadku osiągnięto najbardziej odległe pozycje w wyszukiwaniu nie dają konkretnych informacji o wygranej/przegranej/remisie; mamy tylko informacje, które są ukryte w samej konfiguracji płytki. W takim przypadku funkcja oceny jest znacznie mniej oczywista. Problem oceny sytuacji i przypisywania im wartości liczbowych pomagających w podjęciu decyzji jest jednym z głównych wyzwań programowania heurystycznego i nie ogranicza się tylko do grania w gry. Rozważ program do przeprowadzania zabiegów medycznych i diagnoz. Tutaj również istnieje drzewo, którego węzły

reprezentują kombinacje symptomatycznych problemów i pytań skierowanych do pacjenta, przy czym węzły końcowe reprezentują ostateczne diagnozy. Tutaj też drzewo jest ogromne; ponadto niektóre węzły odpowiadają różnym rodzajom testów medycznych, które niosą ze sobą własne ryzyko. W związku z tym musi nastąpić poszukiwanie heurystyczne, z obserwowalnymi objawami pacjenta i jego odpowiedziami na pytania określające kierunki, które będą realizowane i testy, które zostaną wykonane. Równie trudny jest tu problem ewaluacji, który określa, jak istotny jest dany zestaw możliwości dla pożądanej ostatecznej diagnozy.

Niezależnie od tego, czy w szachach, w medycynie, czy gdzie indziej, po zdefiniowaniu funkcji oceny dla drzewa poszukiwań, nadal pozostaje problem wykorzystania wartości węzłów w drzewie do efektywnego przeszukiwania jego odpowiednich części. Tutaj praca nad sztuczną inteligencją zaowocowała wieloma potężnymi metodami wyszukiwania, które mają wiele zastosowań także w algorytmice nieheurystycznej. Wiele z nich opiera się na podstawowej strategii wyszukiwania zwanej metodą minimax, którą najlepiej wyjaśnić w ramach drzew gry.



Rysunek (a) przedstawia część drzewa gry dla gry dwuosobowej, np. w kółko i krzyżyk lub szachy, wraz z wcześniej ustalonymi wartościami dla każdego węzła końcowego. (Końcowe węzły na rysunku są albo rzeczywistymi pozycjami końcowymi gry, albo pozycjami pośrednimi, poza którymi zdecydowaliśmy się nie szukać.) Korzeń reprezentuje pozycję na planszy, na której gracz Alicja ma iść, a wartości

reprezentują siłę pozycji z punktu widzenia Alicji. Tak więc 999 oznacza wygraną Alicji, a 0 wygraną Boba. Wartości pośrednie odzwierciedlają względne szanse na wygraną Alicji z odpowiednich pozycji, zgodnie z pewną heurystyczną funkcją oceny. Dla Alicji należy wybrać jeden możliwy ruch, który zmaksymalizuje jej zdolność do wygrania. Podstawową ideą jest wielokrotne propagowanie wartości w górę drzewa, zaczynając od dołu. Jeśli bieżący węzeł rodzicielski reprezentuje ruch Alicji, wtedy maksymalna z wartości węzłów potomnych jest dołączona do rodzica, a jeśli reprezentuje ruch Boba, przyjmowana jest wartość minimalna. Uzasadnienie tego jest jasne: zakłada się, że Bob gra w rozsądny sposób, a zatem dołoży wszelkich starań, aby zmaksymalizować własne szanse na wygraną, co oznacza, że będzie starał się zminimalizować szanse Alicji na to. Innymi słowy, Alicja powinna wykonać najlepszy ruch (tzn. ten z maksymalną wartością) przy założeniu, że po nim Bob wykona swój najlepszy ruch (tzn. ten z minimalną wartością funkcji Alicji), po czym Alicja zrobi jak najlepiej w następnym ruchu i tak dalej. Rysunek (b) przedstawia drzewo z wprowadzonymi wartościami min i max. Proces ten można skrócić obserwując, że w niektórych przypadkach nawet częściowa informacja o wartościach potomstwa drugiego pokolenia węzła (czyli jego wnuków) jest wystarczająca do określenia jego wartości końcowej. Na przykład na rysunku poniżej nie ma sensu oceniać skrajnego prawego z pięciu poddrzew drugiej generacji, ponieważ minimalna z trzech skrajnych lewych poddrzew wynosi 42, czyli więcej niż 27, czyli znana wartość po prawej stronie. Minimum 27 i jakakolwiek możliwa wartość korzenia najbardziej prawego drzewa nie będzie większa niż 27, a zatem maksymalizacja, która zostanie przeprowadzona w celu określenia wartości u korzenia, nieuchronnie doprowadzi do wartości 42. Podobny przypadek ma miejsce, gdy dany węzeł reprezentuje ruch Boba, a nie ruch Alicji, ale z maksymalną i minimalną zmianą miejsc. Znaleziono metody przemierzania drzewa podczas procesu propagacji w taki sposób, aby wykorzystać te oszczędności. Jednym z bardziej znanych z nich, który nie zostanie tutaj opisany, jest przycinanie alfa-beta. Podsumowując, wyszukiwanie heurystyczne składa się z:

1. heurystyki zawarte w funkcji wartościowania;

2. zasady określające, jak głęboko będzie wyglądać perspektywa z dowolnej pozycji (zazwyczaj jest to funkcja danej pozycji); oraz

3. sprawna procedura propagacyjna do określania wartości i faktycznego dokonywania wyborów.

Reprezentacja wiedzy

Aby osiągnąć inteligencję algorytmiczną, potrzebujemy czegoś więcej niż tylko heurystyk. Musimy znaleźć sposoby na przedstawienie wiedzy, którą manipulują inteligentne algorytmy. Jeśli części kontrolne programu AI są szczególne, oparte na „miękkim” pojęciu heurystyki, to części danych również są szczególne, oparte na „miękkim” pojęciu wiedzy. To, że dwa razy cztery to osiem i że Francja jest w Europie to wiedza, ale to samo dotyczy faktu, że wszystkie żyrafy mają długie szyje, że Alan Turing był genialny i że naukowcy, którzy nie publikują, giną. Ale co to jest „długi”, a co „genialny” i czy „zginąć” znaczy dosłownie? Co więcej, w jaki sposób przedstawiamy takie fakty w naszych umysłach lub w naszych algorytmicznych bazach wiedzy i jak je wykorzystujemy? Żaden program nie może być oznaczony jako inteligentny, czy to ten, który działa w wąskiej dziedzinie, takiej jak szachy, bloki i piramidy, czy też jako uniwersalny kandydat do zdania testu Turinga, chyba że ma odpowiedni mechanizm przechowywania, wyszukiwania i manipulowania wiedza, umiejętności. Problem reprezentacji wiedzy jest rzeczywiście jednym z centralnych zagadnień sztucznej inteligencji. Trudność polega na tym, że wiedza to nie tylko duży zbiór faktów, ale także wiele zawiłych relacji między nimi. Relacje te mają wiele aspektów i atrybutów, które z kolei tworzą inne, wyższego poziomu relacje z innymi elementami wiedzy. Niewiele wiemy o tym, jak sami przechowujemy i manipulujemy ogromnymi ilościami wiedzy zgromadzonymi w ciągu naszego życia. W rzeczywistości istnieją dowody

na to, że ludzka baza wiedzy jest dynamiczna i asocjacyjna i niekoniecznie działa w sposób sugerujący działanie współczesnych komputerów cyfrowych. Wiele modeli wiedzy zostało zaproponowanych do wykorzystania przez inteligentne programy. Niektóre opierają się na czystych koncepcjach informatycznych, takich jak relacyjne lub hierarchiczne bazy danych, a inne na formalizmach logicznych, takich jak rachunek predykatów lub logika modalna. Niektóre języki programowania, takie jak LISP i PROLOG, które są omówione w Części 3, są bardziej odpowiednie do manipulowania wiedzą niż inne. PROLOG, na przykład, jest całkiem trafny, jeśli chodzi o elementy wiedzy dotyczące prostych relacji, takich jak „Jan jest ojcem Marii", a nawet „rodzic dowolnego przodka Marii jest również przodkiem Maryi". Jednak poza małą, dobrze zdefiniowaną domeną dyskursu wymagane relacje stają się znacznie bardziej skomplikowane, a takie modele stają się zdecydowanie nieodpowiednie. Odzyskiwanie tych elementów wiedzy, które są istotne dla jakiejś decyzji, którą program musi podjąć, staje się ogromnym zadaniem. Nie znaleziono zatem „właściwego" modelu algorytmicznej reprezentacji wiedzy

Systemy oparte na wiedzy

Jednym ze sposobów podejścia do problemu są zasady produkcji. Prosta reguła produkcyjna może stwierdzać, że jeśli X jest firmą, a Y nią zarządza, to Y pracuje dla X. Bardziej złożona reguła, prawdopodobnie odnosząca się do programu rozumienia sceny, stwierdzałaby, że za każdym razem, gdy zostaną znalezione trzy linie spotykające się w jednym punkcie wtedy jest możliwe, że reprezentują róg trójwymiarowego sześcianu, chyba że jeden z kątów jest mniejszy niż 45◦, a inny większy niż 90◦. Niektóre programy specjalnego przeznaczenia zostały nazwane systemami opartymi na wiedzy lub systemami eksperckimi2, ponieważ opierają się na regułach stosowanych przez eksperta w celu rozwiązania konkretnego problemu. Typowy system oparty na wiedzy jest konstruowany poprzez wypytanie eksperta o sposoby, w jakie wykorzystuje on wiedzę ekspercką w rozwiązywaniu danego problemu. Pytający (człowiek), czasami nazywany inżynierem wiedzy, próbuje odkryć i sformułować reguły używane przez eksperta, a system oparty na wiedzy następnie wykorzystuje te reguły do kierowania poszukiwaniem rozwiązania danego przypadku problemu. Sercem systemu opartego na wiedzy jest zestaw reguł podany w pewnym formacie oraz odpowiadający mu mechanizm wyszukiwania do znajdowania reguł, które mają zastosowanie. Powstałe systemy nazywane są również systemami produkcji lub systemy oparte na regułach. Oparte na wiedzy systemy o akceptowalnym poziomie wydajności zostały zbudowane do przeprowadzania ograniczonych form diagnozy medycznej, przydzielania zasobów, takich jak bramki lotnisk do przylatujących lotów, planowania i harmonogramowania operacji dla statków kosmicznych oraz planowania logistyki. Takie systemy oszczędzają wiele milionów dolarów korporacjom, które z nich korzystają. W rzeczywistości Amerykańska Agencja Projektów Badań Obronnych (DARPA), przez wiele lat główna agencja finansująca badania nad sztuczną inteligencją w Stanach Zjednoczonych, stwierdziła, że system planowania i harmonogramowania transportu użyty podczas wojny w Zatoce Perskiej w 1991 r. z nawiązką zwrócił całkowitą kwotę DARPA. inwestycja w sztuczną inteligencję przez okres 30 lat! Musimy jednak zdać sobie sprawę, że oprócz polegania na wyszukiwaniu heurystycznym, reguły kontrolujące wyszukiwanie są tworzone przez kwestionowanie ekspertów, którzy nie zawsze działają według sztywnych reguł. Szanse na nieoczekiwane, być może katastrofalne zachowanie w systemie opartym na wiedzy są zatem nie do pominięcia. Niektórzy ludzie ujmują to w ten sposób: „Czy chciałbyś, aby zaopiekował się tobą skomputeryzowany oddział intensywnej terapii, który został zaprogramowany zgodnie z paradygmatem systemu opartego na wiedzy?" W rzadkich przypadkach urządzenie może podać niewłaściwy lek lub w niewłaściwym momencie zamknąć kluczowy zawór. Jego zachowanie rządzi się regułami, które zostały sformułowane podczas wywiadów z lekarzami ekspertami, którzy niekoniecznie muszą działać w nietypowym przypadku według dobrze sformułowanych reguł. (Oczywiście podobne rzeczy miały miejsce w przypadku programów

nieheurystycznych z powodu nieodpowiednich praktyk inżynierii oprogramowania, jak widzieliśmy w przypadku Therac-25 w Rozdziale 5.) Możliwe jest podjęcie prób poddania krytycznych systemów opartych na wiedzy precyzyjnym technikom analitycznym , co umożliwia nam formalną weryfikację, czy określone właściwości bezpieczeństwa mają zastosowanie w przypadku zastosowania zasad leżących u ich podstaw. Ponieważ, jak wyjaśniono w rozdziale 10, właściwości związane z bezpieczeństwem zwykle zakładają, że złe rzeczy się nie wydarzą, wydaje się, że systemy oparte na regułach byłyby podatne na takie podejście.

Wiedza w zakresie uczenia się, planowania i dedukcji

Problem reprezentacji wiedzy jest rzeczywiście fundamentalny dla mechanizacji wszelkiego rodzaju inteligencji i pojawia się ponownie w uczeniu się, planowaniu i dedukcji. Oto kilka typowych przykładów, które dodatkowo ilustrują potrzebę wyrafinowanej reprezentacji wiedzy. O żadnym programie nie można powiedzieć, że jest naprawdę inteligentny, jeśli nigdy się nie nauczy, jest na zawsze skazany na powtarzanie poprzednich błędów i nigdy nie staje się lepszy. Rozważ program w warcaby, który się uczy. Można założyć, że znasz zasady, a następnie po prostu uczysz się, jak unikać błędów, które popełnia, gdy gra więcej gier. Jednak nawet tutaj algorytmiczny punkt widzenia przedstawia poważne problemy reprezentacyjne. Czy po prostu robimy listę pozycji i ruchów, które okazały się złe, i odtąd wielokrotnie je przeglądamy, aby uniknąć ponownego popełniania tych samych błędów? A może staramy się zapamiętać i zaktualizować bardziej ogólne zasady dobrej gry, które posłużą do modyfikacji heurystyki programu? Te pytania stają się tym trudniejsze, gdy temat jest szerszy. Jak uczą się dzieci? Jak przedstawiają wiedzę, która pozwala im rozpoznawać znajome twarze lub syntetyzować wiadomości, których nigdy wcześniej tak naprawdę nie słyszeli? W jaki sposób dorosły zapamiętuje i odzyskuje wiedzę, która pozwala mu nauczyć się pisać wypracowanie, organizować finanse osobiste lub przystosować się do nowego środowiska? Wcześniej wspomnieliśmy o dedukcji jako o jednym z rodzajów inteligencji wymaganych do zdania testu Turinga. Zostały napisane programy, które dość dobrze radzą sobie z dowodzeniem twierdzeń w geometrii szkoły średniej. Potrafią gromadzić istotne fakty i wyciągać z nich nowe. Jak to jest zrobione? Czy te programy po prostu przechowują listę znanych twierdzeń i kilka logicznych reguł wnioskowania (takich jak „jeśli P jest prawdziwe, a P zawsze implikuje Q, to Q też jest prawdziwe")? A może są zaznajomieni z bardziej złożonymi łańcuchami rozumowania, które są nastawione na tworzenie interesujących twierdzeń? Ponownie, pytania te stają się tym trudniejsze, gdy wiedza związana z pożądanymi dedukcjami nie ogranicza się do wąskiej dziedziny. W jaki sposób wiedza pozwala na wyciągnięcie wniosku, że prawdopodobnie jutro będzie padać, wybór odpowiedniej szkoły dla dziecka lub udowodnienie naprawdę głębokiego twierdzenia w topologii algebraicznej?

Większość naszych dotychczasowych dyskusji dotyczyła wewnętrznych działań inteligentnych programów. Potrzeba mówienia, chodzenia lub widzenia była wielokrotnie rezygnowania, koncentrując się zamiast tego na zrozumieniu, uczeniu się i dedukcji. Jednak bardziej ogólny pogląd na inteligentne maszyny wymaga, aby posiadały one również zdolność do fizycznego naśladowania ludzi. W swojej najbardziej ogólnej formie muszą być w stanie rozumieć i syntetyzować ludzką mowę, a także wykonywać zadania fizyczne w duchu klasycznego robota. Również tutaj problem reprezentacji wiedzy jest problemem fundamentalnym. Niektóre systemy AI mogą rozumieć proste widoki bloków i piramid przy użyciu odpowiedniego wizualnego sprzętu sensorycznego, a inne mogą rozumieć wyraźnie mówiony elementarny angielski przy użyciu sprzętu audio. Jak oni to robią? Czy programy wizyjne po prostu znają każdą możliwą kombinację lokalizacji odpowiednich obiektów, czy też rozpoznają różne konfiguracje skrzyżowań linii i wykorzystują reguły do wywnioskowania z nich ogólnego układu? Co się stanie, jeśli zostanie wprowadzony nowy rodzaj obiektu, powiedzmy cylinder? Czy programy do rozpoznawania mowy mają bazę danych zawierającą profile fal dla wszystkich możliwych wymowy

każdego słowa? A może mają wbudowane reguły, które pozwalają im łączyć wypowiadane segmenty słów w całość? Ponownie, jeśli domena jest znacznie szersza, jak te, które napotyka człowiek oglądający nowe otoczenie lub słuchający bogatej i różnorodnej rozmowy, sprawy stają się znacznie bardziej skomplikowane. Jak ludzie rozumieją różnorodność kolorów, linii i kształtów, które składają się na wnętrze salonu? Jak rozpoznają ruch i odległość od obserwowania rzeczy dwojgiem oczu w krótkim czasie i jak rozumieją obce akcenty? Umiejętność planowania to kolejna inteligentna umiejętność. Niektóre roboty, które działają w bardzo ograniczonym otoczeniu, są w stanie zaplanować sekwencję ruchów, która zaprowadzi je do celu. Jak oni to robią? Czy po prostu przeszukują możliwości, heurystycznie lub w inny sposób? A może wykorzystują bardziej subtelną wiedzę, która umożliwia im planowanie w celu osiągnięcia bardziej ogólnych celów? Ponownie, szersze domeny znacznie utrudniają sprawę. W jaki sposób osoba planuje podróż, nakreśla plan zakończenia roku z dodatnim saldem lub obmyśla strategię wygrania wojny? Jeśli inteligencja jest źródłem życia, jak mówi Księga Przysłów, to problem reprezentacji wiedzy jest z pewnością jej kamieniem węgielnym, a stwierdzenie, czy możemy tchnąć życie — że tak powiem — w komputer wymaga znalezienia odpowiedniego rozwiązania do tego problemu.

Inteligencja bez reprezentacji wiedzy?

„Klasyczne” podejście do sztucznej inteligencji, oparte na tzw. hipotezie reprezentacji wiedzy, zakłada, że kluczem do inteligencji jest poznanie oparte na wiedzy reprezentowanej wewnętrznie. Hipoteza ta pozwoliła naukowcom skoncentrować się na algorytmach uczenia się, planowania, dedukcji itp., jednocześnie w dużej mierze ignorując środowisko, w którym ma się znajdować inteligentny agent. Nawet badania nad wizją komputerową i robotyką były pomyślane jako mające na celu rozwinięcie mechanizmów, które pozwalają inteligentnemu agentowi postrzegać swoje środowisko i działać w nim, ale nadal być tylko danymi wejściowymi i wyjściowymi dla jakiegoś innego centralnego komponentu poznawczego. Ten podstawowy model został zakwestionowany w połowie lat 80. XX wieku. Twierdzono, że interakcja między percepcją a działaniem może tworzyć złożone zachowania i że nie ma potrzeby stosowania komponentu „poznawczego” opartego na wewnętrznej reprezentacji wiedzy, która ma zdolności rozumowania symbolicznego. Poznanie wyłania się z tej interakcji, ale nie jest w nią zaprogramowane. Z tego punktu widzenia „klasyczny” rozkład inteligencji, jaki obserwuje większość badaczy AI, jest przedwczesny. Inteligencja na poziomie człowieka jest zbyt złożona i wiemy o niej zbyt mało, aby móc skutecznie zidentyfikować jej składniki. Zamiast tego badacze podążający za tym poglądem próbują zbudować kompletne systemy, które oddziałują ze swoim środowiskiem, zaczynając od małych, które są tak inteligentne jak, powiedzmy, owady, i budując w górę. Takie podejście rozkłada inteligentne zachowanie zgodnie z jego działaniami, a nie jego funkcjami. Dlatego inteligentny system nie powinien być postrzegany jako składający się z różnych funkcji prowadzących od percepcji do działania poprzez pośrednie zadania poznawcze, takie jak modelowanie i planowanie. Składa się raczej z różnych umiejętności, z których każda przechodzi od percepcji do działania. Na przykład w robocie takie umiejętności mogą obejmować unikanie kolizji, eksplorację i analizowanie widocznej przestrzeni, wyszukiwanie określonych obiektów (takich jak gniazdka elektryczne do ładowania się...) i tak dalej. Rzeczywiście, wiele robotów (w tym komercyjny autonomiczny odkurzacz domowy) zostało wyprodukowanych w ten sposób. Ich poziom inteligencji można porównać do poziomu niektórych owadów (na co nie można kichać!), ale wciąż jest to dalekie od inteligencji na poziomie człowieka. Czas pokaże, czy to oparte na zachowaniu podejście do sztucznej inteligencji rzeczywiście będzie skutecznie zwiększać skalę w przyszłości.

Perspektywy sztucznej inteligencji

Termin „sztuczna inteligencja” został po raz pierwszy ukuty w związku z konferencją w Dartmouth w 1956 roku. Było to spotkanie czołowych badaczy z różnych dziedzin nauki, którzy próbowali ustalić

program badawczy dla raczkującej dziedziny. Ich nastrój był bardzo optymistyczny i spodziewali się wielkich przełomów w ciągu 10 lat. Najważniejszym rezultatem późniejszych badań było uświadomienie sobie, jak trudne były w rzeczywistości problemy. W konsekwencji nieuchronnie ucierpiała reputacja branży.

Jak sobie radziliśmy przez mniej więcej pół wieku, które minęło? Chociaż nie opracowano jeszcze nic zbliżonego do prawdziwej sztucznej inteligencji, zwrócenie się ku bardziej wyspecjalizowanym problemom przyniosło imponujące sukcesy, zarówno pod względem naukowym, jak i komercyjnym. Na przykład dzisiejsze komputery potrafią rozumieć polecenia głosowe, rozpoznawać twarze i obiekty na zdjęciach oraz tworzyć zadowalające tłumaczenia dokumentów technicznych. Jak wspomniano wcześniej, systemy oparte na wiedzy są wykorzystywane w wielu przedsięwzięciach komercyjnych i inżynieryjnych, często za kulisami. W rzeczywistości wiele rzeczy, o których przywykliśmy czytać w science fiction w gazetach pojawiają się obecnie historie opisujące trwające badania, a czasem nawet istniejące systemy. Obejmują one komputery do noszenia, które są stale połączone z ogólnoświatową siecią informacyjną; rzeczywistość wirtualna i rozszerzona; „inteligentny pył", czyli liczne małe czujniki, które działają razem jako rozproszony komputer; „inteligentne pokoje", które za pośrednictwem ekranów wielkości ścian reagują na polecenia głosowe, gesty rąk i mimikę twarzy; i, bardziej ponuro, monitorowanie przez rząd całej komunikacji internetowej. Pomimo tych osiągnięć sztuczna inteligencja jest dziś równie kontrowersyjna, jak na początku. Stało się jasne, że systemy budowane przy użyciu technik AI mogą być bardzo przydatne w praktykowaniu nawet bez osiągnięcia inteligencji na poziomie człowieka. Niektórzy (w tym wielu badaczy sztucznej inteligencji) uważają, że ostateczny cel prawdziwej inteligencji maszynowej jest nieosiągalny. Inni przewidują przyszłość, w której inteligencja maszyn w końcu przewyższy ludzką inteligencję, być może nawet prowadząc do połączenia ludzi i komputerów. Pewien słynny wynalazca i autor przewidział w 1999 r., że do 2029 r. pojawią się bezpośrednie połączenia neuronowe między ludzkim mózgiem a komputerami, a roszczenia maszyn do bycia świadomymi zostaną w dużej mierze zaakceptowane. Przewiduje się, że do 2099 roku nie będzie już wyraźnego rozróżnienia między ludźmi a komputerami. Pozostawiamy czytelnikowi decyzję, czy jest to możliwe, prawdopodobne, a nawet pożądane . . .