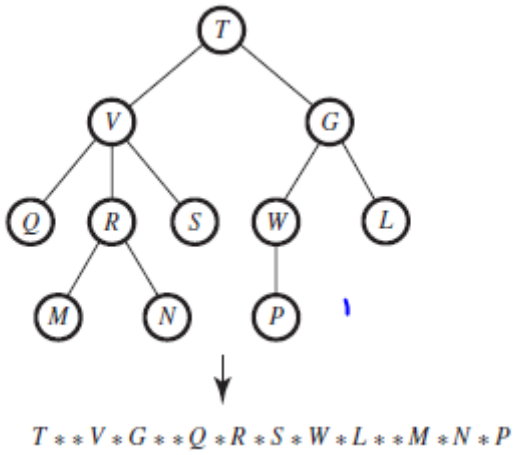
Uniwersalność algorytmów i ich wytrzymałość

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.

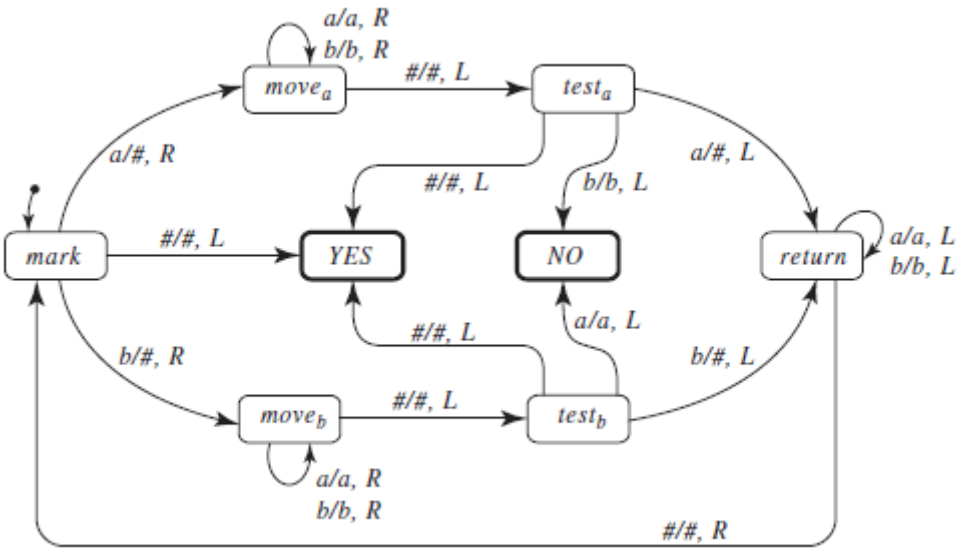
Uproszczenie podstawowych operacji

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

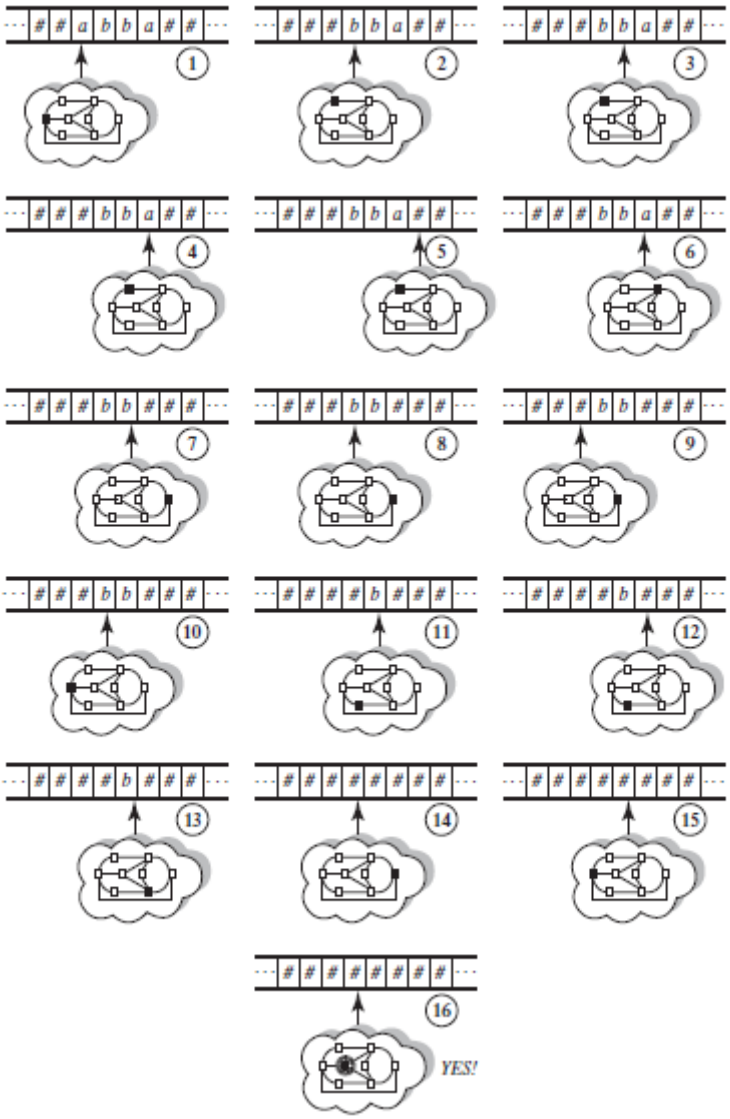


Krawędź prowadząca ze stanu s do stanu t nazywana jest przejściem i jest oznaczona kodem postaci a/b, L lub a/b, R , gdzie a i b są symbolami. Część etykiety nazywana jest wyzwalaczem przejścia i oznacza literę odczytaną z taśmy. Część b to akcja i oznacza literę zapisaną na taśmie. Wreszcie część L 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.

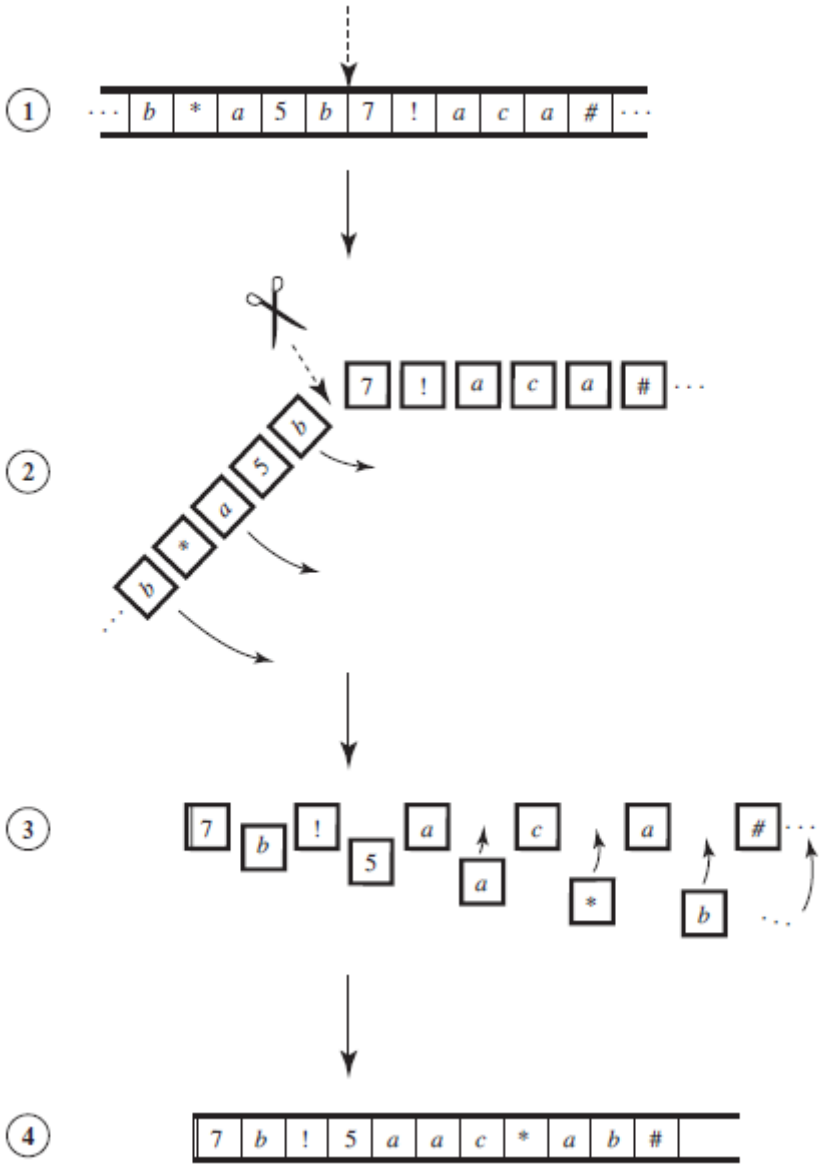
Warianty modelu maszyny Turinga

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

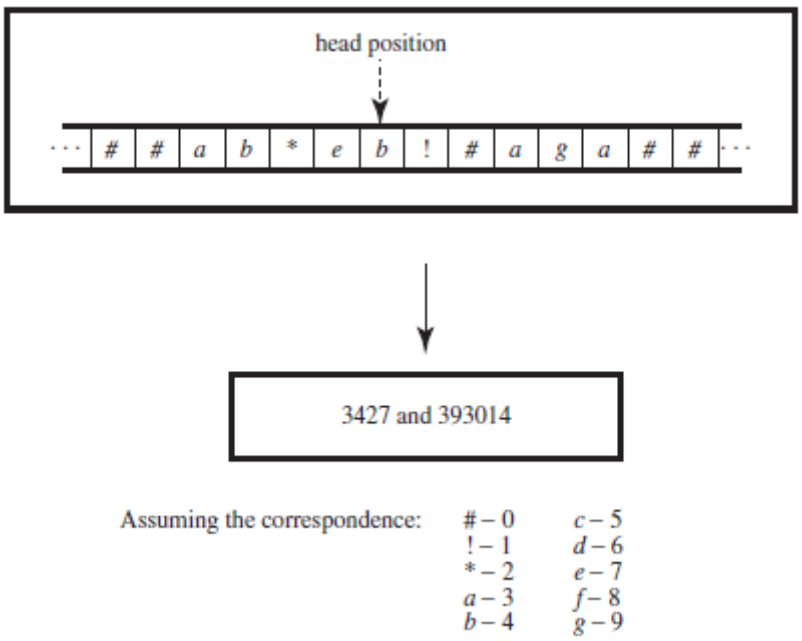
if U = 0 goto B

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.

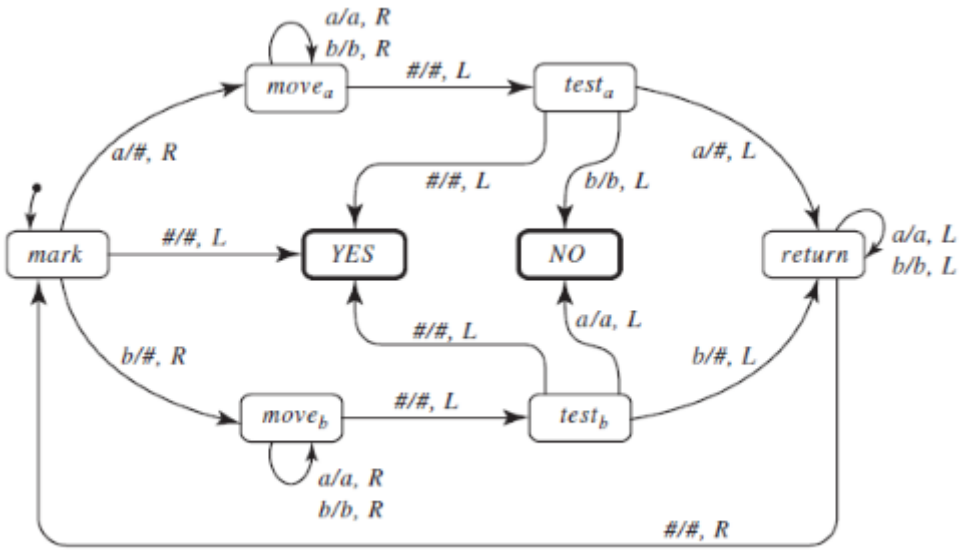
Symulacje jako redukcje

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.

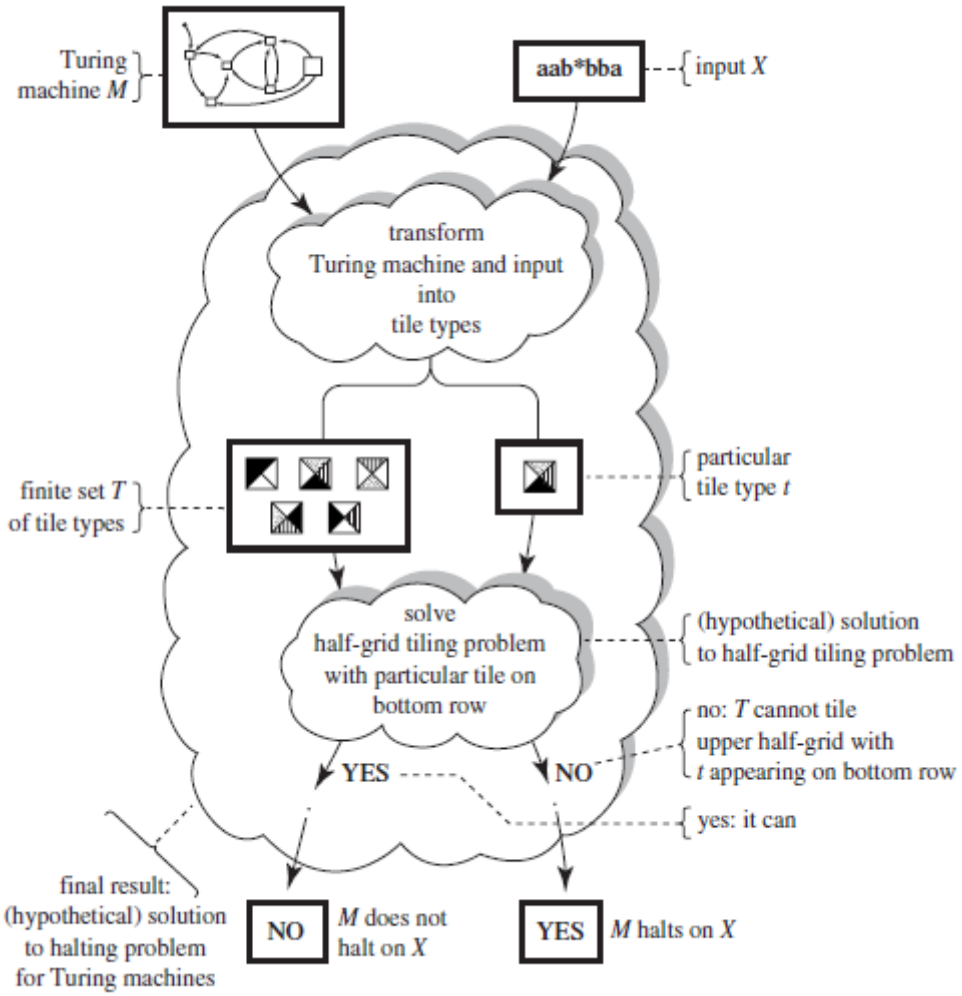
Maszyny Turinga i problem P vs. NP.

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

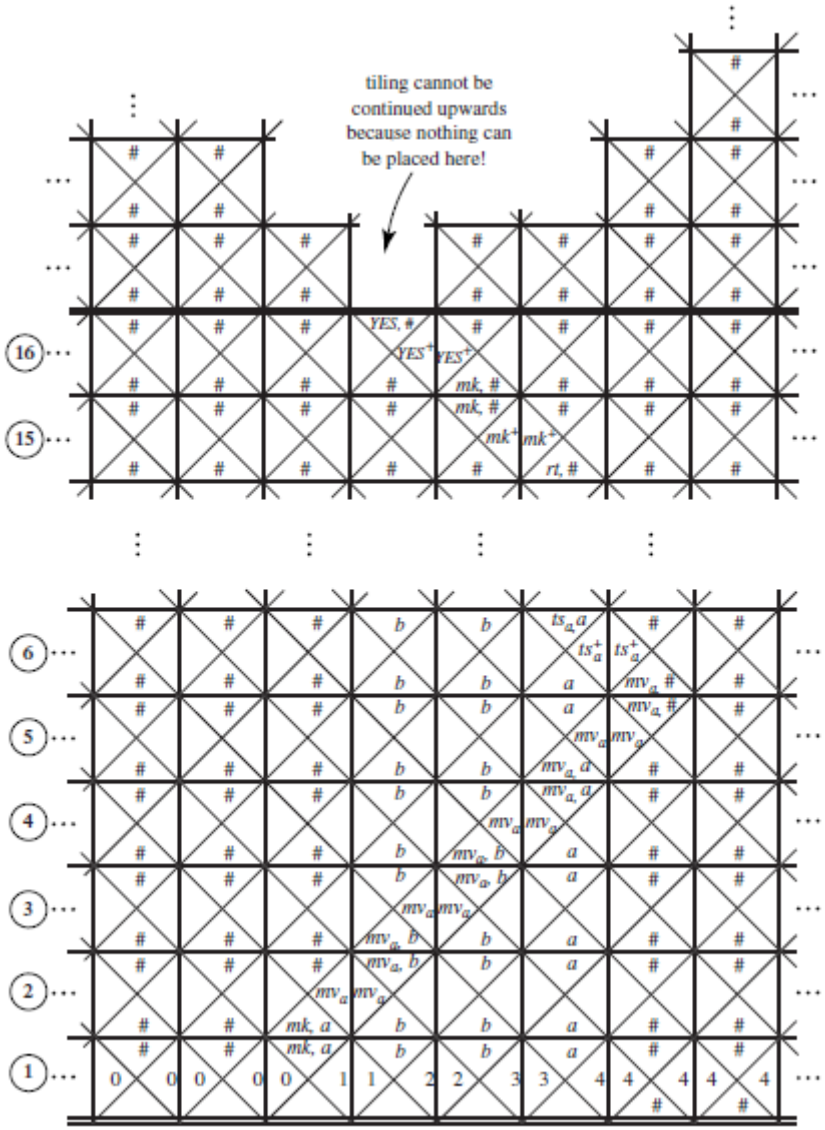
Aby udowodnić górną granicę problemu algorytmicznego - to znaczy znaleźć dobry algorytm - należy użyć najbogatszego i najpotężniejszego dostępnego formalizmu. W rzeczywistości naukowcy zazwyczaj stosują konstrukcje programistyczne bardzo wysokiego poziomu i skomplikowane struktury danych, aby opracowywać 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.

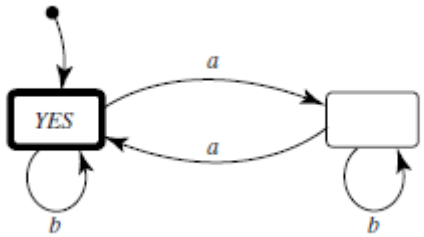
Jednokierunkowe maszyny Turinga lub automaty skończone

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

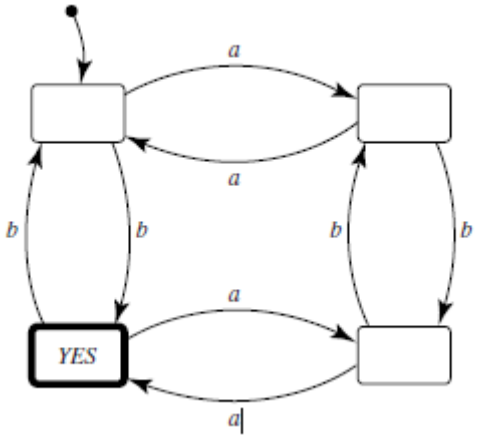
modele obliczeń, a w szczególności na maszyny Turinga, co skutkuje słabszymi klasami problemów, które jednak są bardzo interesujące. Jednym z oczywistych podejść jest ograniczenie wykorzystania zasobów maszyny. Wiemy już, że klasę P uzyskuje się, dopuszczając tylko maszyny Turinga, które zatrzymują się 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

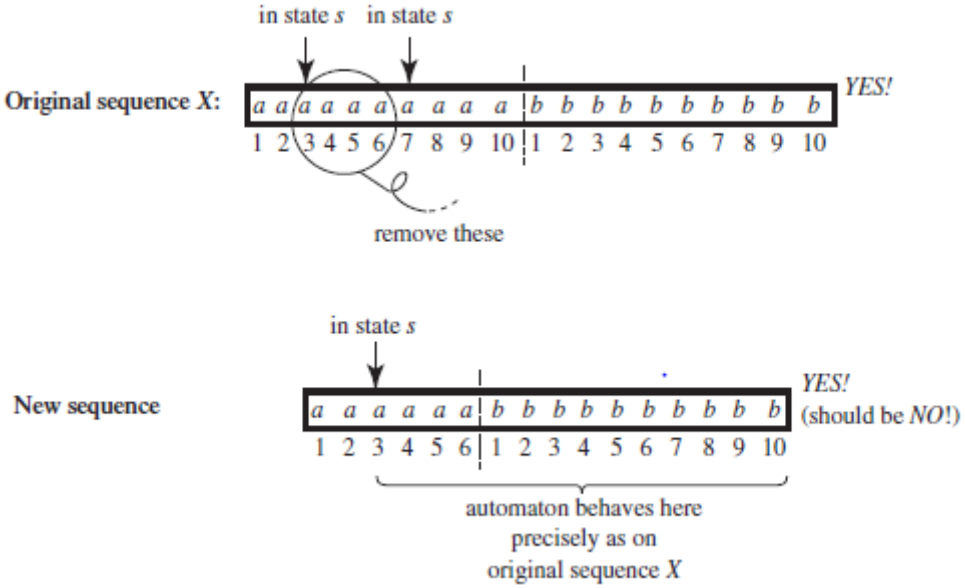


pokazuje automat skończony, który decyduje, czy ciąg wejściowy a i b zawiera parzystą liczbę a. (Co robi automat z rysunku 2?)

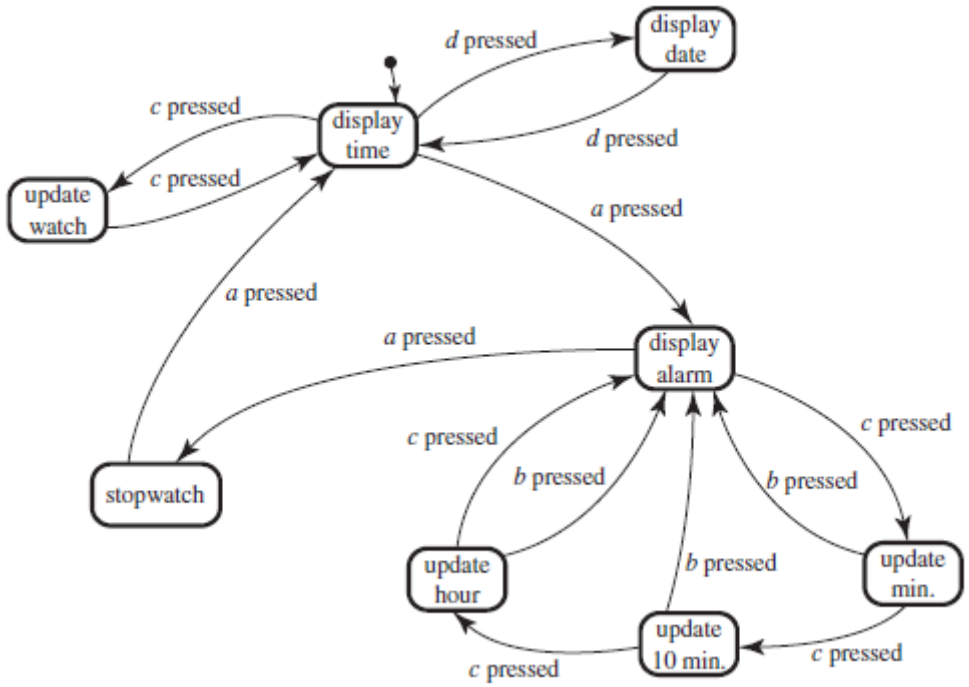


Odpowiadanie na pytania dotyczące parzystości (czyli parzystości i nieparzystości) może na początku wymagać umiejętności liczenia. To nie. Automat z rysunku 1 wykonuje to zadanie nie przez zliczanie liczby a, które widział, ale przez naprzemienne przechodzenie między dwoma stanami, które wskazują aktualny 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.