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.

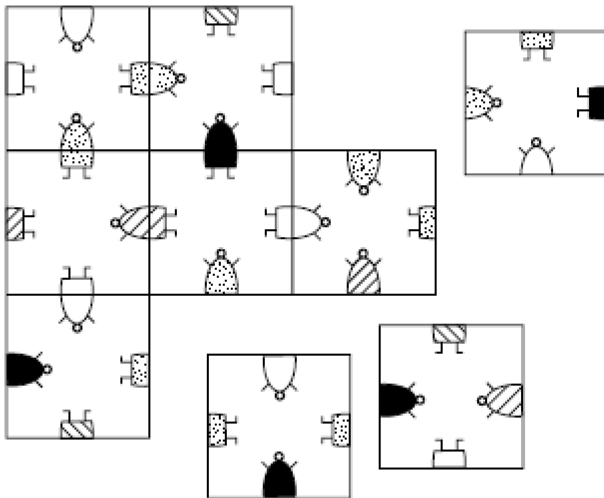
Ponowna wizyta w wieżach Hanoi

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.

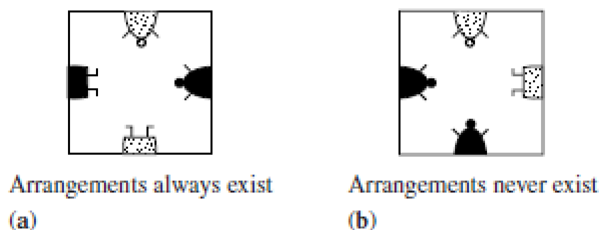
Problem z łamigłówką małp: przykład

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 × 24 × 23 × . . . × 3 × 2 × 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)

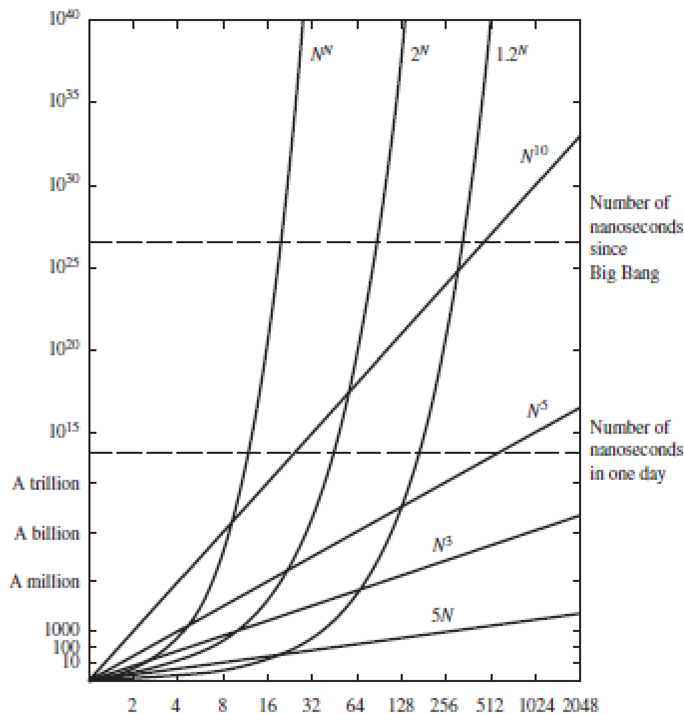
Co więcej, inteligentniejsza wersja algorytmu , który nie uwzględnia rozszerzeń nielegalnych układów częściowych, będzie działał znacznie lepiej w przypadku wielu łamigłówek wejściowych. Jednak nasza gra to analiza najgorszego przypadku. Projektant puzzli faktycznie dąży do zestawów kart, które dopuszczają 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

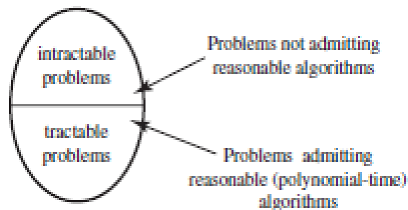


Uderzającą ilustracją różnic między tymi funkcjami są czasy działania algorytmów. Rysunek przedstawia rzeczywiste czasy działania kilku hipotetycznych algorytmów rozwiązywania problemu łamigłówki małp dla różnych wartości N.

	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 $N\log_2 N$, 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?

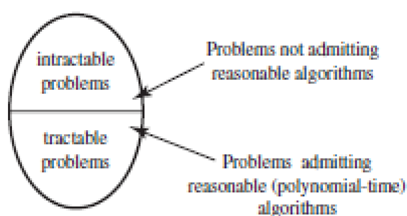
3. Czy ludzie nie próbowali szukać dolnego ograniczenia dla problemu w czasie wykładniczym, abyśmy mogli mieć dowód na to, że nie istnieje żaden rozsądny algorytm?

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.

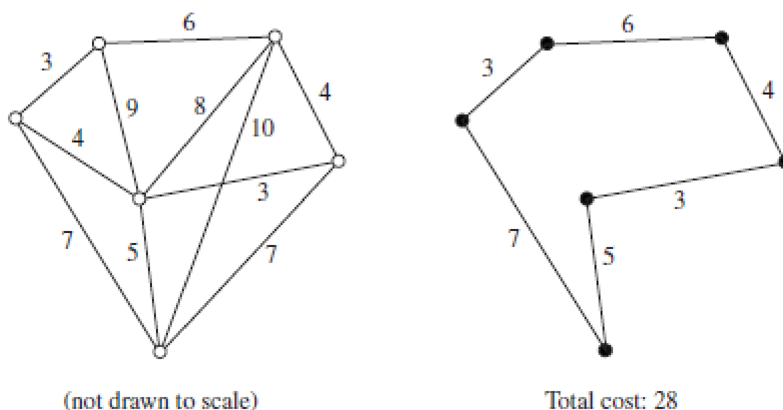
Problemy z układem dwuwymiarowym

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.

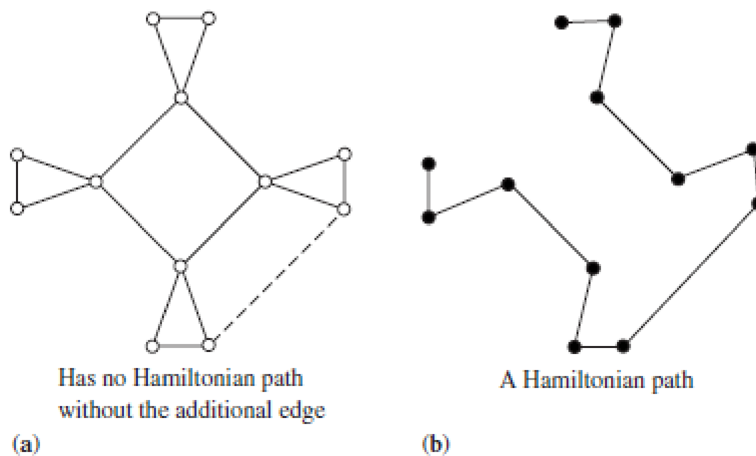
Problemy ze znajdowaniem ścieżki

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.

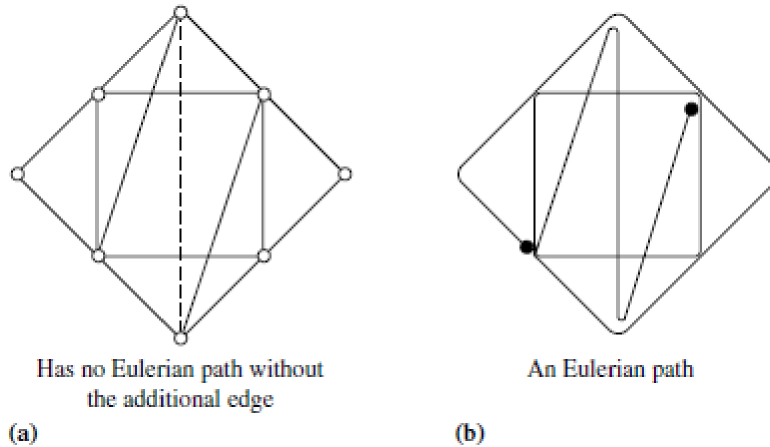


W wersji tak/nie dane wejściowe zawierają liczbę K poza wykresem miasta G, a problem decyzyjny pyta po prostu, czy istnieje wycieczka po G o całkowitym koszcie nie większym niż K. Tak więc dla wykresu z rysunku, odpowiedź brzmiałaby „tak", jeśli K wynosi 30, ale „nie", jeśli K wynosi 27. Pomimo nieco 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 granicznych, 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:

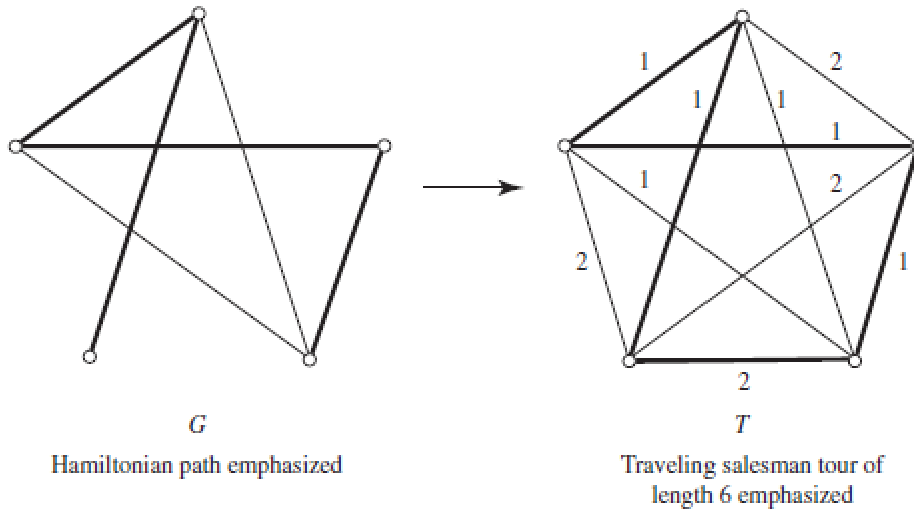
A kiedy wstają, wstają

A kiedy są na dole, są na dole

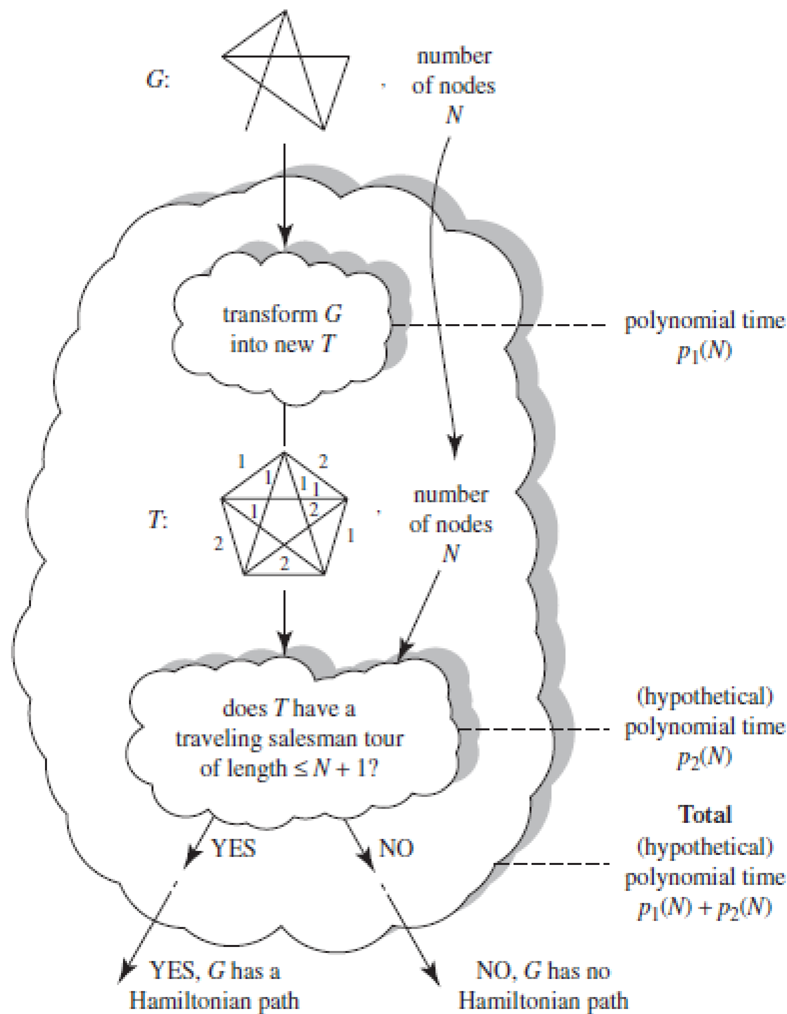
A ponieważ nie mogą być w połowie drogi

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.

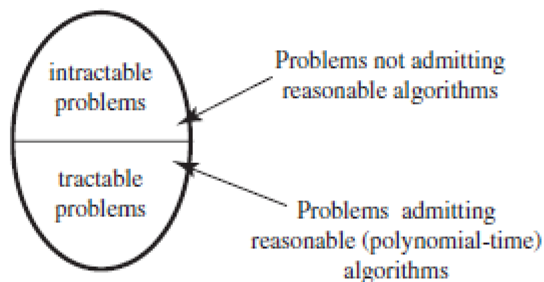
Nieuzasadnione ilości miejsca w pamięci

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

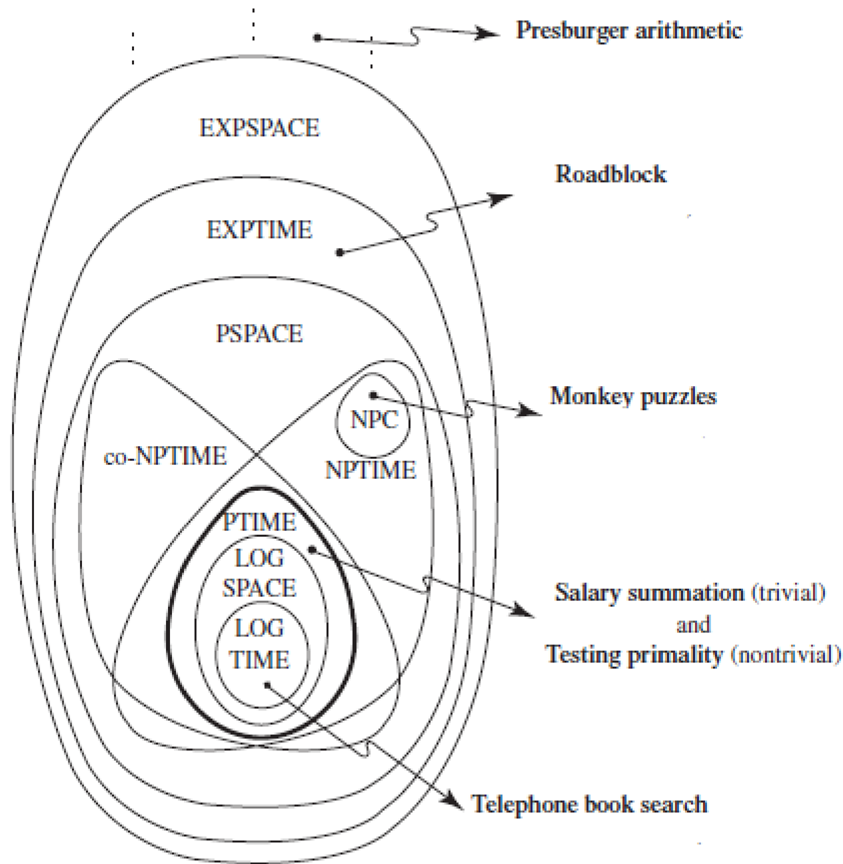
wystarczyłby, aby to wszystko zapisać! Sytuacja jest wyraźnie niewyobrażalnie gorsza w przypadku nieelementarnych ograniczeń przestrzeni.

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.