Nieobliczalność i nierozstrzygalność

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

Zasady gry

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

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

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

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

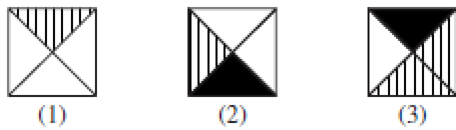
…

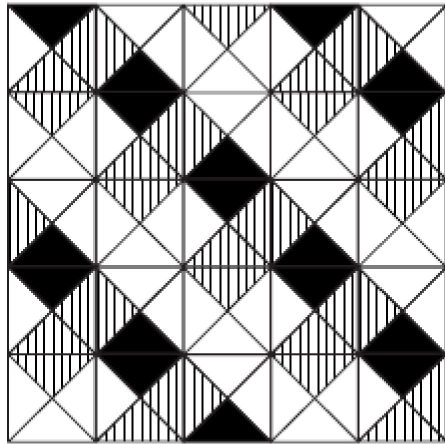
(K) jeśli wejście to I_K, to wypisz „tak” i zatrzymaj się.

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

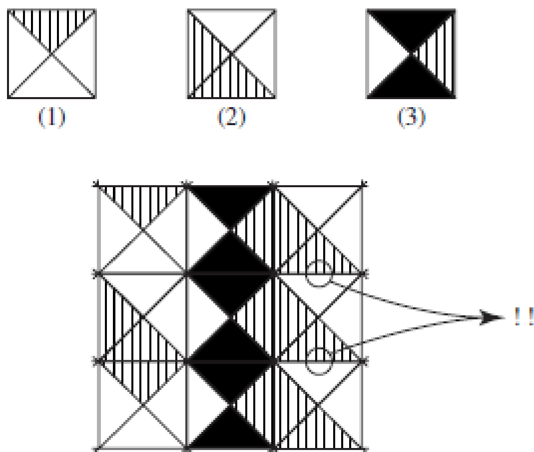
Problem kafelków: przykład

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





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



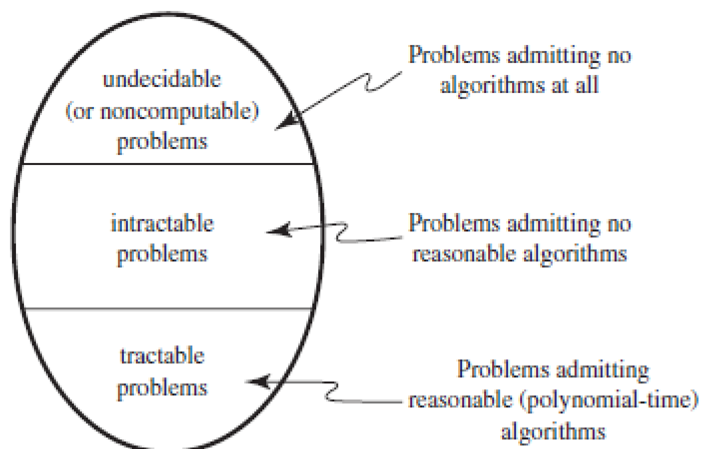
Problem polega na tym, aby w jakiś sposób zmechanizować lub „algorytmizować" rozumowanie stosowane do generowania tych odpowiedzi. I tu pojawia się ciekawostka: tego rozumowania nie da się zmechanizować. Nie ma i nigdy nie będzie algorytmu rozwiązywania problemu kafelkowania! Dokładniej, dla każdego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

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

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

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

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

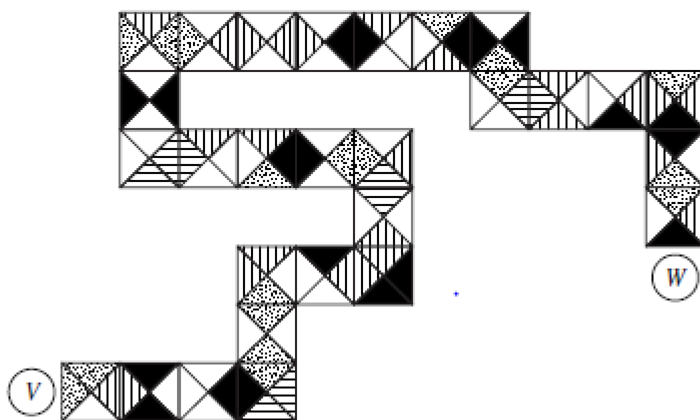


Interesujące jest to, że poniższy, nieco inaczej wyglądający problem, jest w rzeczywistości równoważny z tym, który właśnie opisałem. W tej wersji, zamiast wymagać, aby T był w stanie ułożyć sąsiadujące obszary o dowolnej wielkości, wymagamy, aby T był w stanie ułożyć sąsiadująco całą siatkę liczb całkowitych;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

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

Bezgraniczność może wprowadzać w błąd

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



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

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

Korespondencja słów i równoważność składniowa

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

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

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

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

(b) Admits no correspondence

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

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

Problemy z wyjściami nie są lepsze

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

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

Algorytmiczna weryfikacja programu

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

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

Problem z zatrzymaniem

Rozważ następujący algorytm A:

(1) podczas gdy X ≠1 wykonaj następujące czynności: X ←X – 2;

(2) zatrzymaj się.

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

(1) podczas gdy X ≠1 wykonaj następujące czynności:

(1.1) jeśli X jest parzyste zrób X ← X/2;

(1.2) w przeciwnym razie (X jest nieparzyste) wykonaj X ← 3X + 1;

(2) zatrzymaj się.

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

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

Nic w obliczeniach nie jest obliczalne!

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

Udowodnienie nierozstrzygalności

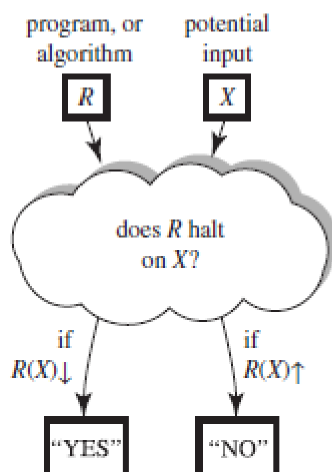
Jak udowodnić, że problem P jest nierozstrzygnięty? Jak ustalić fakt, że nie istnieje żaden algorytm do rozwiązania P, bez względu na to, jak sprytny jest projektant algorytmu? Sytuacja ta jest podobna do opisanej w Części 7 dla problemów NP-zupełnych. Najpierw musi być jeden problem początkowy, którego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

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

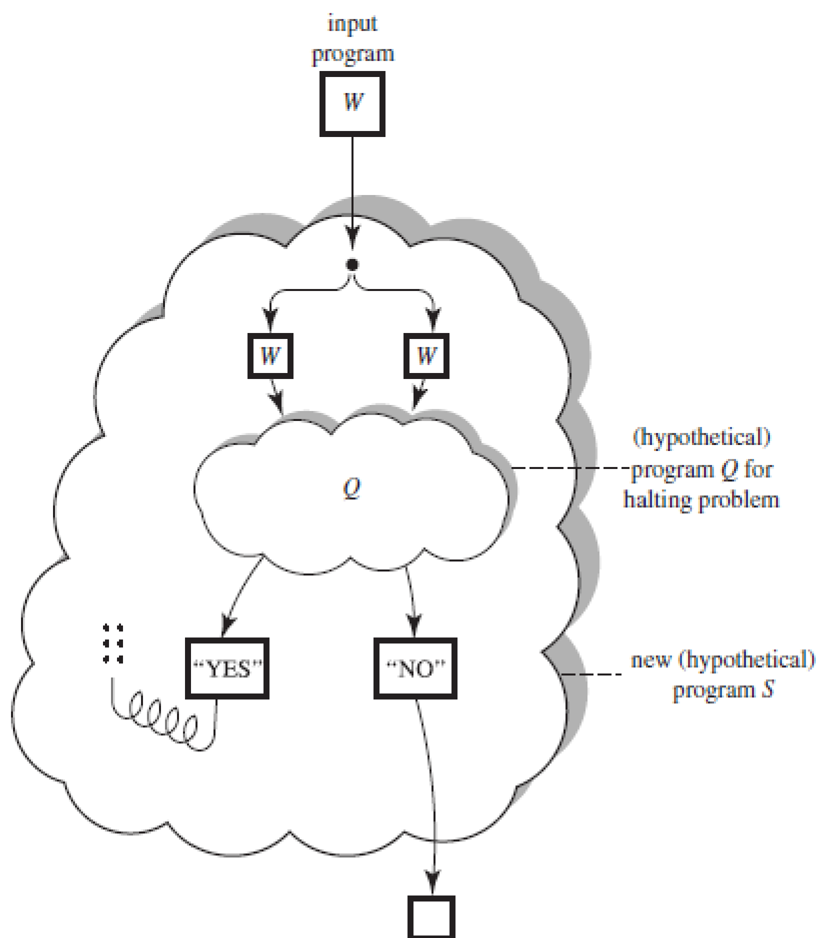
Inne redukcje są znacznie bardziej subtelne. Co, u licha, mają problemy z kafelkowaniem, które mają wspólnego z zakończeniem algorytmicznym? Jak zredukować węże domina do dwukierunkowych formacji słów? Pomimo widocznych różnic, wszystkie te problemy są ze sobą ściśle powiązane, ponieważ są wzajemnie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

Udowodnienie nierozstrzygalności problemu zatrzymania

Udowodnimy teraz, że problem zatrzymania, opisany na rysunku



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



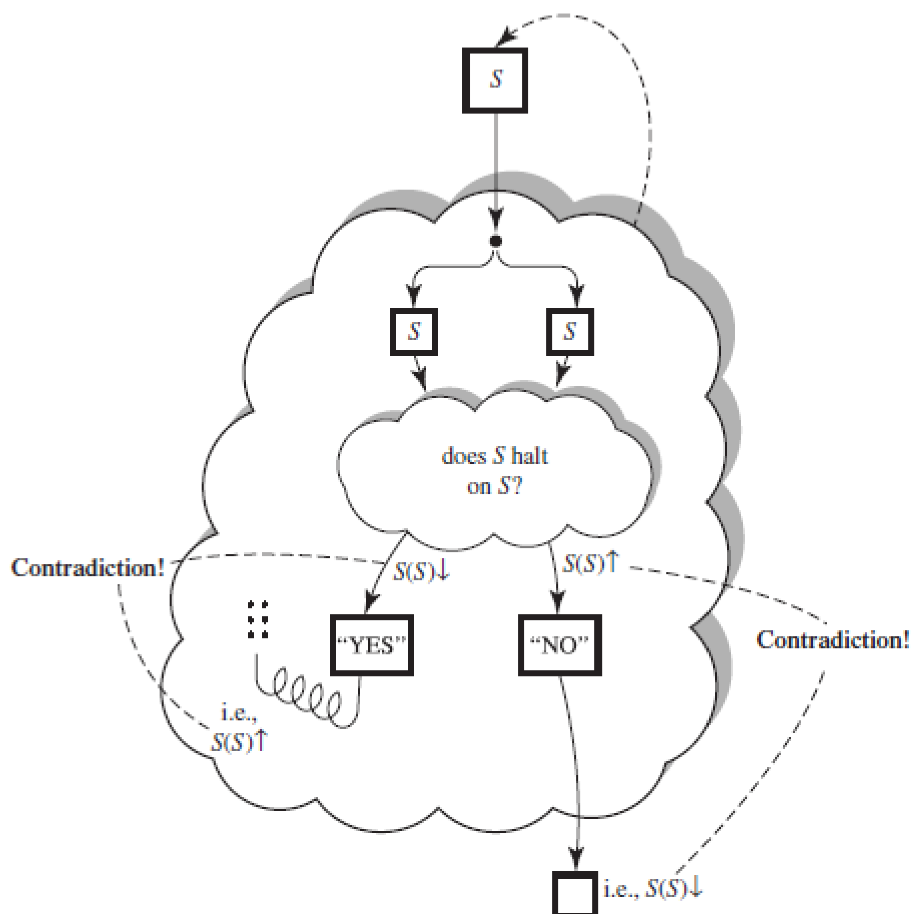
Ten program ma jedno wejście, które jest legalnym programem W w L. Po odczytaniu wejścia S tworzy jego kolejną kopię. To kopiowanie jest oczywiście możliwe w dowolnym języku programowania wysokiego poziomu, przy wystarczającej ilości pamięci. Przypominając, że (zakładając, że istnieje) program Q oczekuje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

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

…

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

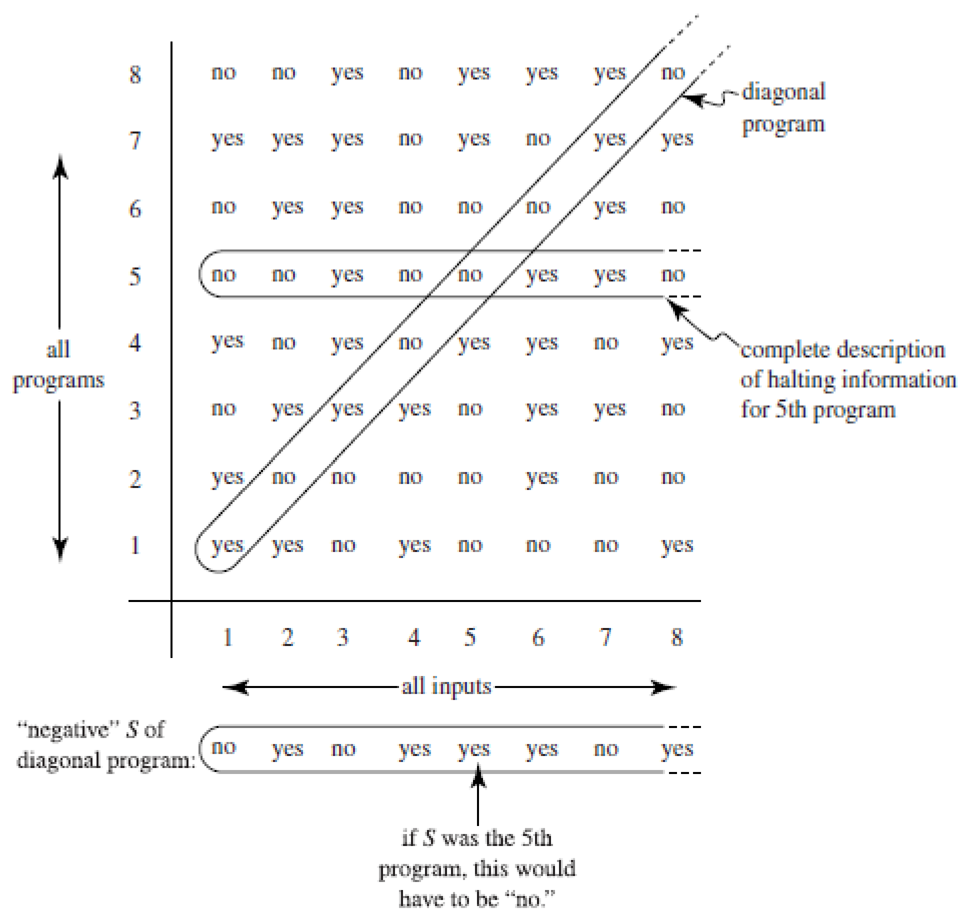
gdzie OUT jest zmienną zawierającą wyjście Q. Na tym kończy się konstrukcja (dziwnie wyglądającego) programu S, który, co należy podkreślić, jest programem legalnym w języku L, zakładając oczywiście, że Q jest. Teraz chcemy pokazać, że z S jest coś nie tak. Samo założenie, że S można skonstruować, jest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



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

Metoda Diagonalizacji

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



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

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

Certyfikaty skończone dla nierozstrzygalnych problemów

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

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

Problemy z certyfikatami dwukierunkowymi są rozstrzygane

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

Problemy, które są jeszcze mniej rozstrzygające!

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

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

Wysoce nierozstrzygnięte problemy

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

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

Cztery podstawowe poziomy zachowań algorytmicznych

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

Rodzaj problemu: status algorytmu

Ograniczona o stałej szerokości: praktyczna

ograniczony: nieusuwalny

nieograniczony : nierozstrzygalny

cykliczne : wysoce nierozstrzygalne

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

Formalizm logiczny : Mówi o : Złożoność

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

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

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

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

Badanie nierozstrzygalności

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

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