ROCZNIKI POLSKIEGO TOWARZYSTWA MATEMATYCZNEGO
Seria II: WIADOMOŚCI MATEMATYCZNE XL (2004)

Leszek Pacholski (Wrocław)

Pierwszy problem milenijny*): czy NP = P?

Z siedmiu problemów milenijnych pierwszy dotyczy zagadnienia z teorii złożoności obliczeniowej, dziedziny, która jeszcze niedawno nie istniała. Podstawowe pojęcia tej teorii pojawiły się w połowie lat sześćdziesiątych dwudziestego wieku, a samo pojęcie NP-zupełności, kluczowe dla tego problemu, zostało wprowadzone w roku 1971. W tym samym roku udowodniono też pierwsze twierdzenia o istnieniu problemów NP-zupełnych.

Prawdopodobnie przeciętny uczony matematyk nie zna biegle wszystkich pojęć potrzebnych do sformułowania problemów milenijnych. Sądzę, że – z wyjątkiem pierwszego problemu – jeśli któregoś pojęcia nie zna, to dlatego, że należy ono do zaawansowanej i specjalistycznej matematyki i nie mieści się w minimach programowych studiów matematycznych. Jednak nawet te, mało znane, pojęcia mieszczą się w pewnej kulturze matematycznej, z którą w czasie studiów i późniejszej pracy naukowej każdy matematyk się zetknął. Każdy wie, co to jest szereg potęgowy, równanie różniczkowe cząstkowe, sfera czy homeomorfizm.

Pojęcia potrzebne do sformułowania pierwszego problemu milenijnego „Czy P = NP?" są elementarne i wykładane na pierwszych wykładach teoretycznych z informatyki oraz na wykładach podstaw informatyki dla matematyków. Niestety, znaczna część twórczych matematyków tych pojęć nie zna, a że nie są one zbudowane na bazie klasycznych pojęć matematyki, matematycy często podchodzą do nich z niechęcią i nieufnością.

Problem „czy NP = P?", a dokładniej, jeden z bardzo wielu problemów równoważnych, można sformułować w bardzo prosty sposób. Na przykład „Czy istnieją liczby naturalne k i c oraz algorytm, który dla dowolnego

*) Od Redakcji: Kolejność i tytuły problemów milenijnych, które przyjęliśmy za anonsem przytoczonym w WM 38(2002), str.61–62, w kolejnych wersjach tego anonsu ulegają zmianom. Także w opracowaniach tych problemów, jakie pojawiają się w druku (K. Devlin, *The Millenium Problems*, Basic Books, New York 2002) – kolejność i tytuły wyglądają różnie. Uznając kolejność i dokładne brzmienie tytułów za mniej istotne, stosujemy się do pierwotnej ich wersji.

grafu planarnego o n wierzchołkach wykonuje nie więcej niż $n^k + c$ kroków i sprawdza, czy graf ten można pokolorować trzema kolorami?". Sformułowanie tego problemu jest proste. Jednocześnie wydaje się, że gdyby taki algorytm przedstawić, to sprawdzenie jego poprawności i oszacowanie liczby kroków nie powinno być trudne. Matematycy znają wiele algorytmów, co do których nie mają wątpliwości, że są one poprawne i potrafią szacować liczbę kroków wykonanych przez te algorytmy. Klasyczna matematyka nie dostarcza jednak języka, pozwalającego w sposób ścisły sformułować i udowodnić fakt, że nie istnieje algorytm spełniający wyżej sformułowane żądania.

Obliczenie. Jednym z podstawowym pojęć informatyki jest obliczenie. *Obliczenie* jest to skończony lub nieskończony ciąg kolejnych przekształceń skończonego układu elementarnych obiektów. Zasady tego przekształcania zadane są przez skończony układ instrukcji, zwany *algorytmem*.

Zaczniemy od kilku przykładów. Z pewnych względów, które później wyjaśnię, nie będę teraz mówił o znanych wszystkim matematykom algorytmach liczbowych, takich jak algorytm Euklidesa. Informatycy budują algorytmy operujące na skomplikowanych obiektach, zwanych *strukturami danych*, jednak wprowadzanie podstawowych pojęć lepiej jest zacząć od algorytmów operujących na obiektach nieskomplikowanych, na przykład na tekstach, czyli skończonych ciągach liter z ustalonego, skończonego alfabetu. Alfabet będziemy oznaczali dużymi literami greckimi Σ, Ξ, ewentualnie z indeksami. Dla dowolnego alfabetu Σ przez Σ^* będziemy oznaczali zbiór wszystkich *słów* (czyli skończonych ciągów) utworzonych z liter tego alfabetu. Algorytmy będą polegały na przekształcaniu tekstu i będą skończonymi zbiorami (lub ciągami) instrukcji pozwalających zastąpić jeden skończony ciąg przez inny. Instrukcje będziemy zapisywać w postaci $\alpha \to \beta$, co oznacza możliwość zastąpienia ciągu α przez ciąg β. I tak na przykład algorytm $\{a \to r,\ c \to i,\ h \to n,\ s \to T,\ y \to g,\ z \to u\}$ zastosowany do słowa *szachy* da, jak łatwo sprawdzić, słowo *Turing*.

Rozważmy inny przykład obliczenia.

Przykład 1. Niech Σ będzie dowolnym skończonym alfabetem, niech $\Sigma' = \{a' : a \in \Sigma\}$ i niech $\Xi = \Sigma \cup \Sigma' \cup \{a'' : a \in \Sigma\} \cup \{\#\}$, gdzie $\#$ jest symbolem, który nie należy do Σ. Rozważmy zbiór instrukcji (1) $a\# \to a'\#a$, (2) $ba' \to b'a'b''$, (3) $a''b' \to b'a''$, i (4) $b''\# \to \#b$, gdzie a, b są dowolnymi elementami Σ. Łatwo zauważyć, że po zastosowaniu pierwszej instrukcji do słowa $wa\#$, gdzie w jest słowem zapisanym w alfabecie Σ i $a \in \Sigma$, słowo $wa\#$ przekształci się w słowo $wa'\#a$. Następnie stosując jeden raz instrukcję (2), potem wielokrotnie instrukcję (3) i w końcu instrukcję (4) możemy zastąpić ostatnią literę b w słowie w, która należy do Σ, w odpowiadającą jej literę $b' \in \Xi$, a następnie przenieść b za znacznik $\#$. W ten sposób, dla dowolnego $w \in \Sigma$, można słowo $w\#$ przekształcić w słowo $w'\#w$, gdzie w'

jest otrzymane z w przez zastąpienie liter z Σ przez odpowiadające im litery z Σ'.

Powyższy algorytm jest *niedeterministyczny*. Oznacza to, że dla zadanego słowa w może być wiele różnych obliczeń zgodnych z zadanym zbiorem instrukcji. Na przykład słowo $abc\#$ można przekształcić w jednym kroku w słowo $abc'\#c$, które z kolei można w jednym kroku przekształcić tylko w słowo $ab'c'b''\#c$. To ostatnie słowo można natomiast przekształcić w dwa słowa: $a'b'a''c'b''\#c$ i $ab'c'\#bc$. Można sprawdzić, że dla słowa $w \in \Sigma^*$ długości $n+1$ można przy pomocy naszego algorytmu słowo $w\#$ przekształcić w słowo $w_1 \in \Xi^*$, które można w jednym kroku przekształcić na n różnych sposobów.

Modele obliczeń. Może się wydawać, że wprowadzone powyżej pojęcia algorytmu i obliczenia są zbyt restrykcyjne i nie obejmują wszystkich możliwych do wyobrażenia przykładów. Tak nie jest. Wszystkie wymyślone do tej pory formalne pojęcia algorytmu i obliczenia okazały się równoważne. Najstarszym modelem obliczeń jest jednotaśmowa maszyna Turinga. Posiada nieskończoną taśmę podzieloną na komórki pamięci. W każdej komórce można przechowywać jeden z symboli należących do pewnego ustalonego alfabetu skończonego. Do dyspozycji dany jest też rozłączny z alfabetem skończony zbiór stanów. W każdym kroku pracy maszyna znajduje się w jednym ze stanów i czyta jedną z komórek pamięci. Wszystkie komórki taśmy, z wyjątkiem skończenie wielu, są puste. Jeden krok pracy maszyny polega na ewentualnym zastąpieniu litery znajdującej się w komórce przez inną literę, zapisanie litery do komórki pustej lub wymazanie litery z komórki pamięci. Jednocześnie maszyna może przesunąć taśmę o jedną komórkę w przód lub w tył i przygotować się do następnego kroku. Praca maszyny kontrolowana jest przez skończony program – skończony zbiór instrukcji. Każda z instrukcji poleca, w zależności od czytanego znaku i stanu maszyny, zapisanie w obserwowanej komórce pamięci odpowiedniej litery lub opróżnienie komórki oraz przesunięcie taśmy o jedną komórkę w przód lub w tył.

W chwili rozpoczęcia pracy na taśmie maszyny znajduje się słowo wejściowe. Kolejne litery słowa zajmują kolejne komórki, zaczynając od komórki czytanej przez maszynę. Wielkość słowa wejściowego to liczba zajętych przez nie komórek pamięci. Maszyna kończy pracę (zatrzymuje się), jeśli nie może wykonać następnego ruchu. Czasami wprowadza się specjalny stan maszyny, stan zatrzymujący, w którym maszyna kończy pracę. Wynik obliczenia, to ciąg kolejnych symboli na taśmie w chwili zatrzymania się maszyny. Czas pracy maszyny, to liczba kroków od chwili rozpoczęcia do chwili zakończenia pracy maszyny. Wprowadza się także pojęcie pamięci użytej przez maszynę. Jest to liczba wszystkich komórek pamięci, które kiedykolwiek w trakcie pracy maszyny zawierały literę z alfabetu maszyny.

Na początek rozważymy bardzo prosty przykład.

Przykład 2. Niech $\Sigma_1 = \{a,\ c,\ g,\ h,\ i,\ n,\ r,\ s,\ T,\ u,\ y,\ z\}$, będzie alfabetem maszyny M i niech $Q = \{p,\ q\}$ będzie zbiorem jej stanów. Program M będzie zawierał $\{ap \to rp+,\ cp \to ip+,\ hp \to np+,\ sp \to Tp+,\ yp \to gp+,\ zp \to up+,\ \varepsilon p \to \varepsilon q\}$, gdzie $\alpha q_1 \to \beta q_2+$, oznacza instrukcję „jeśli w stanie q_1 widzisz literę α, zapisz β, przejdź do stanu q_2 i przesuń się o jedną komórkę do przodu". Stan p jest stanem początkowym, a q zatrzymującym – M nie może w tym stanie wykonać żadnego ruchu. Symbol ε oznacza komórkę pustą. Łatwo zauważyć, że jeśli w kolejne komórki taśmy maszyny wpiszemy słowo *szachy*, to po siedmiu krokach maszyna zatrzyma się i na taśmie znajdzie się słowo *Turing*.

Działanie maszyny Turinga z powyższego przykładu nie różni się istotnie od działanie opisanego wcześniej algorytmu przekształcania tekstu. Najważniejsza różnica polega na tym, że opisana tu maszyna działa sekwencyjnie i deterministycznie, natomiast algorytm przekształcania tekstu może działać równolegle[1] i niedeterministycznie[2]. Można postawić pytanie, który model jest bardziej naturalny – sekwencyjny i deterministyczny, czy równoległy i niedeterministyczny. Informatycy skłaniają się do poglądu, że ten drugi.

Rozważmy teraz inny, bardziej skomplikowany przykład. Będzie to przykład maszyny Turinga, która dodaje dwie liczby zapisane w systemie binarnym. Algorytm jest skomplikowany, gdyż maszyna ma utrudniony dostęp do informacji. Liczby są zapisane na taśmie jedna po drugiej i głowica maszyny musi wykonać wiele ruchów zarówno po to, by odczytać kolejne bity zadanych liczb, jak i po to, by zapisać wynik. Przykład ten podaję po to, by czytelnik zdał sobie sprawę z tego, jak różne mogą być modele obliczeń. Spodziewam się, że większość czytelników przeczyta tylko tekst opisujący działanie maszyny i nie będzie analizowała kolejnych instrukcji. Opis działania można traktować jako program zapisany w języku programowania wyższego rzędu, natomiast instrukcje to program *skompilowany*, zapisany w języku maszynowym (niskiego rzędu).

Przykład 3. Niech $\Sigma = \{0, 1, \#, \emptyset, e_0, e_1\}$ będzie alfabetem maszyny M. Przygotuj się do pracy i przesuń się na koniec zapisanej części taśmy (tu i w Przykładzie 4 $+$ oznacza przesunięcie o jedną komórkę do przodu, $-$ przesunięcie o jedną komórkę do tyłu, natomiast $\cdot$ oznacza, że maszyna się nie przesuwa):

$\#p_1 \to \#p_1-$,
$\varepsilon p_1 \to e_0 p_2+$,
$ip_2 \to ip_2+$, gdzie $i \in \{0, 1\}$

[1] To znaczy kilka liter można jednocześnie zastąpić innymi literami.
[2] Litery możemy zastąpić przez inne w dowolnej kolejności.

$\#p_2 \to \#p_3+$,
$ip_3 \to ip_3+$, gdzie $i \in \{0, 1, \emptyset\}$,
$\#p_3 \to \#p_4+$,
$ip_4 \to ip_4+$, gdzie $i \in \{0, 1, \emptyset\}$,
$\#p_4 \to \#p_5-$.

Szukaj dotychczas niewykorzystanego bitu drugiej liczby, zapamiętaj ten bit.

$\emptyset p_5 \to \emptyset p_5-$,
$ip_5 \to \emptyset c_1^i-$.

Jeśli wszystkie bity drugiej liczby zostały wykorzystane, zanotuj ten fakt.

$\#p_5 \to \#d_2-$.

Przenoś zapamiętaną wartość na początek taśmy i poszukuj dotychczas niewykorzystanego bitu pierwszej liczby.

$jc_1^i \to jc_1^i-$, gdzie $i, j \in \{0, 1\}$
$\#c_1^i \to \#c_2^i-$, gdzie $i \in \{0, 1\}$,
$\emptyset c_2^i \to \emptyset c_2^i-$, gdzie $i \in \{0, 1\}$,
$\emptyset d_2 \to \emptyset d_2-$.

Do zapamiętanego bitu drugiej liczby dodaj pierwszy dotychczas niewykorzystany bit pierwszej liczby. Jeśli wszystkie bity drugiej liczby zostały już wykorzystane, zanotuj niewykorzystany bit pierwszej liczby.

$jc_2^i \to \emptyset c_3^k-$, gdzie $i, j, k \in \{0, 1\}$ oraz $k = i + j$,
$jd_2 \to \emptyset c_3^j-$, gdzie $j \in \{0, 1\}$.

Jeśli nie znalazłeś niewykorzystanego bitu pierwszej liczby, to przenieś na początek taśmy zapamiętany bit drugiej liczby, lub przygotuj się do zakończenia obliczeń.

$\#c_2^i \to \#c_4^i-$, gdzie $i \in \{0, 1\}$,
$\#d_2 \to \#d_6-$.

Przenoś zapamiętane informacje na początek taśmy:

$jc_3^i \to jc_3^i-$, gdzie $i, j \in \{0, 1\}$,
$jd_6 \to jd_6-$, gdzie $j \in \{0, 1\}$,
$\#c_3^i \to \#c_4^i-$, gdzie $i \in \{0, 1\}$,
$jc_4^i \to jc_4^i-$, gdzie $i, j \in \{0, 1\}$.

Dodaj zapamiętaną wartość do wartości „przeniesionej" z poprzedniego bitu i zapisz bit przeniesienia. Jeśli zostały jeszcze jakieś niewykorzystane bity, przygotuj się do ich wykorzystania:

$e_j c_4^i \to kc_5^l-$, gdzie $i, j, k, l \in \{0, 1\}$, $k \equiv i + j (mod\ 2)$ oraz l jest częścią całkowitą z $(i + j)/2$,
$\varepsilon c_5^i \to eip_2+$, gdzie $i \in \{0, 1\}$.

Jeśli pamiętasz, że wszystkie bity obu liczb zostały wykorzystane, zakończ obliczenie:

$e_j d_6 \to jd_7$, gdzie $j \in \{0, 1\}$.

Obliczenie dla danych m, n zakończy się nie później niż po $(2(2+\log_2 m+\log_2 n))^2$ krokach i mało prawdopodobne wydaje się istotne zmniejszenie niezbędnej liczby kroków wtedy, gdy dysponujemy tylko jedną taśmą. Dużo szybciej można dodać dwie liczby, jeśli dysponujemy *wielotaśmową* maszyną Turinga.

Wielotaśmowa maszyna Turinga działa podobnie jak maszyna jednotaśmowa, z tą różnicą, że dane może zapisywać na kilku taśmach. Jeden krok polega na przeczytaniu danych obserwowanych przez maszynę w aktualnie dostępnych komórkach pamięci i, w zależności od stanu i ciągu obserwowanych liter, zapisanie w obserwowanych komórkach nowych symboli, niezależnym przesunięciu taśm o jedną komórkę i zmianie stanu.

Dla przykładu rozważymy trzytaśmową maszynę Turinga. Na taśmie można zapisywać symbole z alfabetu $\Sigma = \{0, 1, \#\}$. Instrukcja $\alpha\beta\gamma q \to \alpha' + \beta - \gamma \cdot q'$ oznacza „jeśli na pierwszej taśmie widzisz α, na drugiej β, na trzeciej γ i znajdujesz się w stanie q, to na pierwszej taśmie zapisz α', na drugiej β', na trzeciej γ', przesuń pierwszą taśmę do przodu, drugą do tyłu, a trzeciej nie ruszaj i przejdź do stanu q'".

Jeśli na pierwszej taśmie zapiszemy binarne rozwinięcie liczby m, na drugiej binarne rozwinięcie liczby n, tak jak w przypadku maszyny jednotaśmowej ograniczone z przodu i z tyłu ogranicznikiem $\#$, a na taśmie trzeciej wpiszemy tylko ogranicznik $\#$, to podany poniżej algorytm zapisze sumę $m + m$ na trzeciej taśmie.

Przykład 4. Przygotuj się do pracy:

$\#\#\#q_s \to \# + \# + \# \cdot q_t$,

$ij\#q_t \to ir_i jr_j\# \cdot q_t$, gdzie $i, j \in \Sigma$, $i \neq \#$ lub $j \neq \#$, oraz dla $x \in \{i, j\}$, $r_x = +$ jeśli $x \neq \#$, $r_x = \cdot$ jeśli $x = \#$,

$\#\#\#q_t \to \# - \# - \# - q_0$.

Dodawaj bit po bicie:

$ij\varepsilon q_k \to is_i js_j l - q_{k'}$ gdzie $i, j \in \Sigma$, $i \neq \#$ lub $j \neq \#$, dla $x \in \{i, j\}$, $s_x = -$ jeśli $x \neq \#$, $r_x = \cdot$ jeśli $x = \#$, oraz $l < 2$, $l \equiv i' + j' + k (mod 2)$, $k' = \lfloor \frac{i'+j'+k}{2} \rfloor$, gdzie dla $x \in \{i, j\}$, $x' = x$, jeśli $x \neq \#$ lub $x' = 0$ jeśli $x = \#$.

$\#\#\varepsilon q_i \to \# \cdot \# \cdot i - p$, jeśli $q_i = 1$,

$\#\#\varepsilon q_i \to \# \cdot \# \cdot \# \cdot p_f$, jeśli $q_i = 0$.

Zakończ pracę:

$\#\#\varepsilon p \to \# \cdot \# \cdot \# \cdot p_f$.

Łatwo zauważyć, że obliczenie dla danych m, n zakończy się nie później niż po $2(1+\max(\log_2 m, \log_2 n)$ krokach, a więc znacznie szybciej niż w przypadku maszyny jednotaśmowej. Trudno sobie jednak wyobrazić program dla maszyny Turinga kolorujący wierzchołki grafu lub sprawdzający, czy zadana liczba naturalna jest liczbą pierwszą. Dlatego oprócz maszyn Turinga

rozważa się wiele innych modeli obliczeń pozwalających na łatwiejszy opis skomplikowanych algorytmów operujących na złożonych obiektach. Stosunkowo proste są maszyny o swobodnym dostępie do pamięci (maszyny RAM), operujące na komórkach pamięci, zwanych *rejestrami*, z których każda może zawierać dowolną liczbę naturalną. Programy dla tych maszyn przypominają programy pisane w niskopoziomowych[3] językach programowania, a ich obliczenia pracę prawdziwych komputerów, z tą różnicą, że w prawdziwych komputerach wielkość rejestrów jest ograniczona i nie mogą one przechowywać dowolnych liczb naturalnych, lecz tylko liczby o pewnej, ustalonej liczbie bitów. Bardziej skomplikowane są modele obliczeń, dla których instrukcje przypominają polecenia w wysokopoziomowych[4] językach programowania. Modelami obliczeń są też *systemy przepisywania termów* (wyrażeń algebraicznych) i *systemy przepisywania słów*, takie jak ten, który opisałem wprowadzając pojęcie algorytmu i obliczenia.

Rozważa się nie tylko takie modele obliczeń, które opisują abstrakcyjne maszyny liczące lub imperatywne[5] języki programowania. Ponadto dziedziny, na których operują algorytmy w różnych modelach obliczeń, mogą być różne. Jednak we wszystkich modelach można pomiędzy dziedzinami określić efektywne izomorfizmy – izomorfizmy o tej własności, że elementarne operacje w jednej dziedzinie przechodzą na proste operacje obliczalne w drugiej. Przykładem izomorficznych dziedzin są liczby naturalne i słowa nad ustalonym alfabetem skończonym. Łatwo skończone ciągi liter zakodować przy pomocy liczb i łatwo zbudować algorytm, który dla danych liczb reprezentujących dwa słowa obliczy liczbę reprezentującą zlepienie (złączenie) tych słów. I na odwrót, liczby można reprezentować jako ciągi cyfr, a algorytmy realizujące podstawowe operacje arytmetyczne są algorytmami na słowach reprezentujących liczby.

Dwa pierwsze modele obliczeń, model Churcha i model Turinga, powstały w 1936 roku. Bardzo szybko okazało się, że są one równoważne w tym sensie, że klasy funkcji obliczalnych w tych modelach[6] były identyczne. Równoważne okazały się też wszystkie inne modele, które później zdefiniowano. Dla każdej pary modeli istnieje efektywna procedura, która pozwala przekształcić dowolny algorytm w jednym modelu na algorytm w drugim obliczający tę samą funkcję.

[3] Instrukcje w niskopoziomowych językach programowania opisują elementarne operacje na komórkach pamięci.

[4] Pojedyncze instrukcje w wysokopoziomowych językach oprogramowania mogą opisywać skomplikowane ciągi elementarnych instrukcji. Program napisany w takim języku może przypominać algorytm opisany w języku matematyki.

[5] W językach imperatywnych programy są ciągami poleceń wykonania pewnych operacji. W innych językach program może być definicją funkcji lub innego obiektu.

[6] To znaczy funkcji, które można w tych modelach zdefiniować.

Dla każdego ze znanych modeli można wprowadzić pojęcia złożoności obliczeniowej, a w szczególności pojęcie liczby kroków algorytmu. Dla ułatwienia będę rozważał algorytmy na słowach lub maszyny Turinga. Przez *rozmiar danych* będę rozumiał ich długość, a więc liczbę liter w słowie lub liczbę niepustych komórek na taśmie maszyny Turinga przed rozpoczęciem obliczenia.

Niech $f : \mathbb{N} \to \mathbb{N}$. Mówimy, że algorytm ma złożoność czasową ograniczoną przez funkcję f, jeśli dla danych o rozmiarze n algorytm zakończy pracę po nie więcej niż $f(n)$ krokach. Algorytm jest wielomianowy, jeśli jego złożoność czasowa ograniczona jest przez pewien wielomian.

Wydaje się dość oczywiste, że pojęcie złożoności czasowej w istotny sposób zależy od modelu. Algorytm w jednym modelu może, po przetłumaczeniu do innego modelu, mieć znacznie większą złożoność obliczeniową. Maszyny z dostępem bezpośrednim (RAM) mogą w jednym kroku dodać dwie dowolnie duże liczby naturalne, wielotaśmowa maszyna Turinga zrobi to w czasie ograniczonym przez funkcję liniową, natomiast maszyna jednotaśmowa potrzebuje czasu ograniczonego przez funkcję kwadratową. Okazuje się jednak, że różnice w złożoności pomiędzy różnymi modelami nie są duże. Wspomniane wcześniej efektywne procedury tłumaczące algorytmy jednego modelu na symulujące je algorytmy w drugim, można skonstruować tak, że każdy krok symulowanego algorytmu będzie zastąpiony przez ciąg kroków o długości ograniczonej przez wielomian. Wynika stąd, że klasa algorytmów wielomianowych nie zależy od modelu obliczeń.

Problemy decyzyjne, języki. Algorytmów używa się do znajdowania poszukiwanych obiektów: obwodów Hamiltona, najlepszego rozkładu zajęć, optymalnej trasy przejazdu, wartości funkcji liczbowej oraz do rozstrzygania problemów decyzyjnych takich jak stwierdzenie, czy obwód Hamiltona istnieje, czy można ułożyć plan zajęć spełniający pewne warunki, czy wartość funkcji jest nieujemna i tak dalej. Oczywiście, jeśli potrafimy znaleźć obiekt, to znamy też odpowiedź na problem decyzyjny. Prawdziwe jest także wynikanie odwrotne: jeśli znamy metodę rozstrzygania problemów decyzyjnych, to szybko możemy znaleźć odpowiadające temu problemowi rozwiązanie.

Rozważmy następujący problem, o którym będziemy później jeszcze mówili. Formuła zdaniowa w koniunkcyjnej postaci normalnej jest to koniunkcja pewnych formuł, zwanych klauzulami, z których każda jest alternatywą pewnych zmiennych zdaniowych i ich negacji. Formułą w takiej postaci jest na przykład $(p \vee q \vee \neg r) \wedge (\neg p \vee q \vee r) \wedge (\neg p \vee \neg q \vee \neg r)$.

Przez SAT będziemy oznaczali następujący problem spełnialności formuł: czy dla zadanej formuły w koniunkcyjnej postaci normalnej istnieje wartościowanie zmiennych, przy którym staje się ona prawdziwa. Przypuśćmy teraz, że znamy algorytm rozwiązujący problem decyzyjny SAT,

a chcemy nie tylko stwierdzić, że zadana formuła φ jest spełnialna, ale także znaleźć wartościowanie, przy którym jest spełniona. Niech $v_1, \ldots, v_n$ będą zmiennymi formuły φ. Przez T będziemy oznaczali wartość logiczną *prawda*, a przez F *fałsz*. W pierwszym kroku sprawdzamy, czy φ jest spełnialna. Następnie tworzymy formułę φ_T podstawiając T za zmienną v_1 i korzystając z algorytmu dla SAT sprawdzamy, czy jest ona spełnialna. Jeśli tak, kładziemy $\varphi_1 = \varphi_T$ i $v_1 = T$, jeśli nie, kładziemy $\varphi_1 = \varphi_F$ i $v_1 = F$. Następnie postępujemy podobnie z kolejnymi zmiennymi, podstawiamy T w φ_1 za v_2 otrzymując $\varphi_{1,T}$. Jeśli $\varphi_{1,T}$ jest spełnialna, kładziemy $\varphi_2 = \varphi_{1,T}$ i $v_2 = T$, jeśli nie, kładziemy $\varphi_2 = \varphi_{1,F}$ oraz $v_2 = F$. Postępując w ten sposób otrzymujemy wartościowanie zmiennych $v_1, \ldots, v_n$ spełniające φ. Cała procedura poszukiwania wartościowania wymagała $(n+1)$-krotnego zastosowania algorytmu sprawdzającego spełnialność, a więc gdyby istniał algorytm pracujący w czasie wielomianowym i sprawdzający spełnialność, istniałby też wielomianowy algorytm znajdujący wartościowanie spełniające zadaną formułę. Jak się zapewne czytelnik domyśla, takiego algorytmu dotychczas nie znaleziono. Wszystkie znane algorytmy rozwiązujące ten problem pracują w czasie wykładniczym.

Przez P będziemy oznaczali klasę problemów, dla których istnieje wielomianowy algorytm decyzyjny. Problemy decyzyjne na ogół opisuje się w terminach języków formalnych lub podzbiorów pewnego ustalonego uniwersum. Na przykład problem istnienia obwodu Hamiltona przedstawia się jako zbiór H grafów skończonych, w których taki obwód istnieje, a problem SAT, to zbiór tych formuł w koniunkcyjnej postaci normalnej, które są spełnialne. Przy tym zakłada się, że każdy obiekt zakodowany jest w postaci skończonego ciągu symboli z ustalonego skończonego alfabetu Σ. Zbiór wszystkich skończonych słów utworzonych z liter alfabetu Σ oznacza się przez Σ^*. Język nad alfabetem Σ, to dowolny podzbiór Σ^*. Maszyna Turinga M akceptuje słowo w, jeśli M uruchomiona ze słowem w na taśmie (na wyróżnionej taśmie w przypadku maszyny wielotaśmowej) zatrzyma się w wyróżnionym stanie, zwanym *stanem akceptującym*. Zauważmy, że fakt, że M nie zaakceptowała w może oznaczać, że M zatrzymała się w stanie, który nie jest akceptujący lub że obliczenie się nigdy nie kończy (M się nie zatrzyma).

Instancja problemu, jest to element uniwersum, a więc w naszych przykładach graf skończony G lub formuła φ, a rozstrzygnięcie problemu polega na stwierdzeniu, czy $G \in$H lub $\varphi \in$SAT. Używanie takiego języka może się wydawać sztuczne i niepotrzebne w przypadku klas problemów, które są, tak jak klasa P, zamknięte na negację. Jednak nie wszystkie klasy problemów rozważane w tym artykule takie są.

Niedeterminizm. Analizując Przykład 1 wspomniałem, że podany tam algorytm jest *niedeterministyczny*. Algorytm jest deterministyczny, jeśli

w każdym kroku obliczenia można wykonać co najwyżej jeden ruch. Dla maszyny Turinga oznacza to, że każde dwie różne instrukcje w programie mają różne słowa przed symbolem $\rightarrow$. W niektórych modelach obliczeń, na przykład opisujących imperatywne języki programowania, determinizm jest bardzo naturalny i rozważanie obliczeń niedeterministycznych wymaga specjalnych konstrukcji. W innych bardzo naturalny jest niedeterminizm. W systemach przepisywania nawet jednoznaczność instrukcji przepisywania nie daje gwarancji, że oparte na nich obliczenia będą deterministycze, gdyż można przekształcać różne fragmenty słowa.

Trudno sobie wyobrazić używanie algorytmów niedeterministycznych do obliczania wartości funkcji liczbowych. Jest oczywiste, że różne obliczenia zgodne z algorytmem niedeterministycznym mogą prowadzić do innych wyników. Czasem jednak obliczenia niedeterministyczne mogą być pożyteczne. Za obliczenie niedeterministyczne można uważać dowód matematyczny[7]. Interesuje nas, czy jakieś zdanie jest twierdzeniem i przekonamy się o tym, jeśli znajdziemy jakikolwiek dowód tego zdania. A dowód (formalny), to ciąg przekształceń zgodnych z pewnymi zasadami, które, w przypadku dowodów formalnych, można ułożyć w listę instrukcji przepisywania. Podobnie, gdy chcemy stwierdzić, czy graf można pokolorować trzema kolorami lub czy istnieje w grafie obwód Hamiltona, wystarczy znaleźć jedno dobre kolorowanie lub jeden obwód Hamiltona. Można to zrobić przeszukując wszystkie możliwości lub w sposób niedeterministyczny. Także, gdy chcemy stwierdzić, czy formuła zdaniowa jest spełnialna, można przeszukać wszystkie możliwe wartościowania zmiennych zdaniowych lub zgadnąć dobre wartościowanie.

Mówimy, że problem decyzyjny L jest w klasie NP, jeśli istnieje algorytm niedeterministyczny i wielomian p takie, że dla każdej pozytywnej instancji x tego problemu, czyli dla danych x takich, że $x \in L$, istnieje obliczenie tego algorytmu z danymi początkowymi x, które w czasie ograniczonym przez $p(n)$, gdzie n jest rozmiarem x, daje odpowiedź pozytywną, natomiast jeśli $x \notin L$, żadne obliczenie, albo równoważnie żadne obliczenie wielomianowe, nie da odpowiedzi pozytywnej. Na przykład, problem SAT jest w NP, ponieważ można łatwo napisać algorytm wielomianowy, który w sposób niedeterministyczny generuje wartościowanie zmiennych, a następnie sprawdza, czy przy tym wartościowaniu zadana formuła ma wartość logiczną T. Jeśli formuła jest spełnialna, jedno z możliwych obliczeń wygeneruje wartościowanie spełniające zadaną formułę, natomiast jeśli nie jest, żadne z wygenerowanych wartościowań jej nie spełni.

Zauważmy, że definicja klasy NP nie jest symetryczna w tym sensie, że negacja problemu z klasy NP nie musi należeć do NP. Klasa P jest zamknięta na negację, bo jeśli istnieje wielomianowy algorytm deterministyczny dla

[7] Mówiąc o dowodach, jako obiektach badań, mam na myśli dowody formalne.

problemu L, to aby otrzymać wielomianowy algorytm dla dopełnienia L wystarczy po zakończeniu obliczenia odwrócić wartość logiczną odpowiedzi. Odwrócenie wartości logicznej odpowiedzi w przypadku algorytmu niedeterministycznego nie doprowadzi do zdefiniowania dopełnienia klasy L, gdyż zanegowanie definicji akceptacji daje zdanie *żadne obliczenie wielomianowe nie daje pozytywnej odpowiedzi*.

Znamy już wszystkie definicje niezbędne do tego, żeby zrozumieć problem: *czy NP = P*. Pytanie jest proste: *czy dla każdego problemu należącego do klasy NP można znaleźć deterministyczny algorytm wielomianowy rozstrzygający ten problem?*. Trudno jednak zrozumieć, jak pytanie, które mówi coś o wszystkich problemach należących do klasy NP, może być równoważne z pytaniem dotyczącym jednego problemu.

Problemy NP-zupełne. W praktyce matematycznej często redukuje się jeden problem do drugiego. Pojęcie redukcji odgrywa ważną rolę także w logice i teorii złożoności obliczeniowej. Przypuśćmy, że mamy dane dwa problemy decyzyjne L_1 i L_2 określone dla instancji ze zbiorów U_1 i U_2. Redukcją problemu L_1 do L_2 nazywamy funkcję obliczalną $f : U_1 \to U_2$ taką, że $x \in L_1 \Leftrightarrow f(x) \in L_2$. Redukcja ta jest wielomianowa, jeśli f jest obliczalna przez deterministyczny algorytm wielomianowy. Zauważmy, że jeśli problem L_1 redukuje się do problemu L_2, który jest rozstrzygalny, to problem L_1 też jest rozstrzygalny. Podobnie, jeśli problem L_1 wielomianowo redukuje się do problemu L_2, który jest w P (NP), to problem L_1 też jest w P (NP). Pojęcie redukcji (redukcji wielomianowej) wprowadza w zbiorze problemów relację częściowego (quasi) porządku, pozwala bowiem klasyfikować problemy ze względu na ich trudność: problem L_1 jest łatwiejszy od problemu L_2, jeśli się do niego redukuje.

Łatwo zauważyć, że wszystkie problemy w P są wielomianowo równoważne, to znaczy każdy problem w P redukuje się do każdego innego problemu w P w czasie wielomianowym. W klasie NP tak być nie musi. Okazuje się jednak, jak udowodnił S. Cook [8] w roku 1971, że w klasie NP są problemy najtrudniejsze. Problem należący do klasy NP jest NP-zupełny, jeśli redukuje się do niego każdy inny problem z klasy NP. Dowód Cooka istnienia problemów NP-zupełnych polegał na pokazaniu, że z zadanego programu M maszyny Turinga, wielomianu p i danych x, można w czasie wielomianowym względem x zbudować formułę w koniunkcyjnej postaci normalnej, która będzie spełnialna wtedy i tylko wtedy, gdy maszyna M zaakceptuje dane x w czasie $p(n)$, gdzie n jest rozmiarem danych x.

Wkrótce, mniej więcej w tym samym czasie, R. Karp [18] wykazał, że istnieje wiele problemów NP-zupełnych. Lista znanych problemów NP-zupełnych jest obecnie bardzo długa, co więcej, ze wszystkich znanych natu-

ralnych problemów należących do klasy NP wszystkie, z wyjątkiem jednego[8], należą do P lub są NP-zupełne. Zauważmy, że jeśli dla jakiegokolwiek z problemów NP-zupełnych znajdziemy deterministyczny algorytm wielomianowy, to udowodnimy, że NP = P.

Fakt, że prawie wszystkie znane problemy należą do klasy P lub są NP-zupełne może sugerować, że klasa NP jest sumą P i klasy problemów NP-zupełnych. Tak być jednak nie musi. W 1975 R. Ladner [21] udowodnił, że jeśli $P \neq NP$, to istnieje wiele problemów pośrednich, takich, które nie są w P i nie są NP-zupełne. Z drugiej strony fakt, że od wprowadzenia pojęcia NP-zupełności nie znaleziono żadnych problemów pośrednich, może sugerować, iż takich problemów naturalnych nie ma. Niestety, trudno udowodnić takie twierdzenie bez sprecyzowania, co to znaczy „naturalny". Pewnym dość dobrym przybliżeniem pojęcia „problem naturalny" jest pojęcie *problem spełniania więzów*[9]. W 1978 roku T. Schaefer [28] udowodnił twierdzenie o dychotomii. Wykazał, że podklasy problemu SAT są albo w P, albo są NP-zupełne. Od kilku lat trwają próby uogólnienia tego twierdzenia. W 2002 roku A. Bulatov [6] wykazał, że twierdzenie o dychotomii zachodzi dla problemów spełniania więzów nad zbiorem 3 elementowym (SAT jest problemem nad zbiorem 2 elementowym).

Siła niedeterminizmu. Można zadać pytanie, jaka jest moc niedeterminizmu przy innych ograniczeniach na wielkość zasobów wykorzystywanych przy obliczeniach. Jeśli rozważamy maszyny Turinga, których czas pracy i długość użytej taśmy są nieograniczone, to klasy języków rozpoznawanych przez maszyny deterministyczne i niedeterministyczne są takie same. Także dla najsłabszego modelu obliczeń – automatów skończonych, czyli maszyn Turinga, które mogą tylko przeczytać słowo wejściowe, a nie mogą niczego na taśmie zapisać – klasy języków niedeterministycznych i deterministycznych są takie same[10]. *Automaty stosowe*, to ograniczone maszyny Turinga znacznie silniejsze od automatów skończonych. Oprócz taśmy wejściowej, którą można tylko przeczytać[11], mogą używać tylko jednej taśmy roboczej. Dodatkowo mają dostęp tylko do litery, która znajduje się w ostatniej zapisanej komórce (na szczycie stosu). Żeby przeczytać literę znajdującą się przed nią (w głębi stosu), muszą wymazać wszystkie litery znajdujące się

[8] Jest to problem izomorfizmu grafów. Do niedawna nie wiadomo było też, czy problem pierwszości jest w P.

[9] Ang. *constraint satisfaction problem*.

[10] Czas pracy maszyny deterministycznej i niedeterministycznej jest taki sam i jest równy liczbie liter w słowie wejściowym, ale liczba stanów automatu deterministycznego rozpoznającego ten sam język, co automat niederministyczny, może być wykładniczo większa.

[11] Jeden raz, od początku, do końca słowa wejściowego.

za nią (przed nią), tracąc przy tym informację zapisaną przy pomocy tych liter. Niedeterministyczne automaty stosowe są mocniejsze od deterministycznych, rozpoznają palindromy, których automaty deterministyczne nie potrafią rozpoznawać.

Jeśli zamiast ograniczać czas pracy maszyny Turinga, ograniczy się używaną pamięć (liczbę użytych komórek taśmy), to okazuje się, że niedeterminizm nie zwiększa mocy obliczeniowej. W 1970 roku W. Savitch [27] wykazał, że niedeterministyczną maszynę Turinga, używającą $f(n)$ komórek taśmy dla danych rozmiaru n, można symulować przy pomocy maszyny deterministycznej używającej $O(f^2(n))$ komórek taśmy. Stąd klasa NPSPACE języków rozpoznawanych przez niedeterministyczne maszyny Turinga, których taśma jest wielomianowo ograniczona, jest identyczna z klasą PSPACE języków rozpoznawanych przez deterministyczne maszyny Turinga z wielomianowo ograniczoną taśmą.

W 1988 roku N. Immerman [15] i R. Szelepcsényi [32] wykazali niezależnie, że jeśli $S(n) \geq \log(n)$, to klasa problemów akceptowalnych przez niedeterministyczne maszyny Turinga z taśmą ograniczoną przez S, jest zamknięta na dopełnienie. Ciekawe, że dwóch autorów doszło do tego samego wyniku z różnych powodów. Immermanowi wynik ten był potrzebny do rozwijanej przez niego teorii opisującej związki definiowalności w strukturach skończonych ze złożonością obliczeniową, a Szelepcsényi rozwiązał znany, otwarty od wielu lat, problem dotyczący gramatyk kontekstowych.

Teoria modeli skończonych. Podejmowano wiele prób rozstrzygnięcia problemu „czy NP = P". Próby te można podzielić na dwie kategorie: błędne próby udowodnienia, że NP = P, które do niczego nie doprowadziły i próby udowodnienie, że NP $\neq$ P, które doprowadziły do powstania kilku ciekawych teorii. W 1974 roku N. Jones i A. Selman [17] zauważyli związek pomiędzy klasami modeli skończonych elementarnych teorii logicznych i teorią złożoności obliczeniowej, a R. Fagin [11] udowodnił, że klasa NP jest tożsama z klasą zbiorów modeli skończonych[12] definiowalnych przez zdania egzystencjalne logiki drugiego rzędu, to znaczy zamknięte[13] formuły zdaniowe postaci $\exists R_1 \exists R_2, \ldots, \exists R_n \varphi(R_1, R_2, \ldots, R_n, Q_1, \ldots, Q_m)$, gdzie $R_1, R_2, \ldots, R_n, Q_1, \ldots, Q_m$ są symbolami relacji, a φ jest dowolną formułą pierwszego rzędu, w której występują relacje $R_1, R_2, \ldots, R_n$ i $Q_1, \ldots, \ldots, Q_m$.

W tym miejscu potrzebne jest wyjaśnienie. Klasę NP wprowadziłem jako klasę języków, czyli zbiorów słów. Zapewne nie dla wszystkich będzie jasne, co to znaczy, że definiowalny zbiór modeli skończonych należy do klasy NP.

[12] W skończonym języku relacyjnym, bez symboli funkcji i stałych.

[13] Bez zmiennych wolnych.

Najprościej powiedzieć, że każdy skończony obiekt, to dla informatyka ciąg bitów, a wobec tego zbiór obiektów skończonych, to język nad dwuliterowym alfabetem. Struktura relacyjna $\mathcal{M}$, to skończony zbiór M, na którym określone są relacje $R_1, \ldots, R_n$. Jeśli R jest relacją k-arną M, to R można zakodować jako jej funkcję charakterystyczną, czyli ciąg m^k bitów, gdzie m jest mocą M. Strukturę $\mathcal{M}$ zakodujemy, na przykład, wypisując kolejno kody relacji $R_1, \ldots, R_n$, oddzielone specjalnym symbolem $\#$[14].

Wieloletni, wciąż nie zakończony program badawczy związany z twierdzeniem Fagina opierał się na przekonaniu, że *oczywiście* klasa zbiorów modeli skończonych definiowalnych przy pomocy egzystencjalnych formuł drugiego rzędu nie jest zamknięta na dopełnienie. Realizacja programu badawczego polegała na poszukiwaniu problemów NP-zupełnych, których dopełnienie nie da się opisać zdaniem egzystencjalnej logiki drugiego rzędu. Niestety, mimo że jest wiele problemów w NP, które w sposób naturalny definiuje się w języku egzystencjalnej logiki drugiego rzędu, a których dopełnienie *na pewno* nie jest definiowalne tym języku, nie znaleziono metod, które pozwoliłyby to udowodnić. Metody takie, używające technik gier Ehrenfeuchta – Fraissego, opracowano dla ograniczonych klas formuł – egzystencjalnych formuł drugiego rzędu z kwantyfikatorami monadycznymi, to znaczy formuł postaci $\exists X_1 \exists X_2, \ldots, \exists X_n \varphi(X_1, \ldots, X_n, Q_1, \ldots, Q_m)$, gdzie $X_1, \ldots, X_n$ są symbolami zbiorów.

Innym problemem jest charakteryzacja klasy P w terminach logiki. Okazało się, że nie jest to proste. Problem bierze się stąd, że obliczenia w większości standardowych modeli przeprowadza się na obiektach uporządkowanych[15], natomiast elementy struktury skończonej nie muszą być uporządkowane. Dla uporządkowanych struktur relacyjnych N. Immerman [14] i M. Vardi [33] znaleźli charakteryzację klasy P w terminach logiki: klasa P jest identyczna z klasą zbiorów skończonych struktur liniowo uporządkowanych definiowalnych w logice pierwszego rzędu[16] z dodanym operatorem najmniejszego punktu stałego. Logiki opisującej klasę P w strukturach bez porządku prawdopodobnie nie ma, ale dotychczas nie udało się tego udowodnić.

Badania nad teorią modeli skończonych, przynajmniej te inspirowane przez problem „czy NP = P?", straciły w ostatnich latach dynamikę. Prac jest niewiele i wydaje się, że dalszy postęp wymaga znalezienia istotnie nowych metod. Osobom zainteresowanym problemami teorii modeli skończonych można polecić książkę N. Immermana [16].

[14] Nie będzie to kod binarny, proste ćwiczenie znalezienia kodu binarnego pozostawiamy czytelnikowi.

[15] Dane dla maszyny Turinga wprowadza się w kolejnych komórkach taśmy wejściowej, w *ponumerowanych* rejestrach maszyny RAM znajdują się liczny naturalne, na których określona jest relacja dobrego porządku liniowego.

[16] Z kwantyfikatorami wiążącymi elementy struktury.

Maszyny z wyrocznią. W teorii obliczeń i w teorii złożoności obliczeniowej, jednym z najczęściej używanych narzędzi do dowodzenia twierdzeń orzekających, że jakieś dwie klasy obiektów są różne, jest metoda przekątniowa. W ten sposób dowodzi się, że istnieją zbiory rekurencyjnie przeliczalne, które nie są rekurencyjne, i tak dowodzi się, że klasa EXPTIME, problemów rozstrzygalnych w czasie wykładniczym przez deterministyczne maszyny Turinga, zawiera problemy, które nie należą do P. Metoda jest prosta. Buduje się maszynę Turinga uniwersalną dla danej klasy maszyn. Maszyna ta ma dwa wejścia: jedno do wprowadzania kodu konkretnej maszyny, a drugie do zadawania danych wejściowych[17]. Przy pomocy maszyny uniwersalnej buduje się problem w klasie większej, na przykład w klasie EXPTIME, kodujący wszystkie problemy z danej klasy, na przykład wszystkie problemy w klasie P. I teraz, jeśli konstrukcja była udana, łatwo metodą przekątniową pokazać, że problem uniwersalny nie jest w klasie mniejszej. Ta metoda niestety zawodzi, gdy chce się udowodnić, że klasy P i NP są różne. Co więcej, można udowodnić, że metoda przekątniowa nie pozwala na wykazanie, że $\mathrm{NP} \neq \mathrm{P}$.

Maszyna Turinga z wyrocznią jest to maszyna M z dodatkową taśmą *wyroczni*. Na tej taśmie maszyna w czasie pracy zapisuje dowolne ciągi symboli do momentu, gdy znajdzie się w specjalnym stanie pytającym $q_?$. Dalsza praca maszyny zależy teraz od tego, czy napis x na taśmie wyroczni należy do języka A zwanego *wyrocznią*. Jeśli $x \in A$, to maszyna M przechodzi w jednym kroku to stanu q_{tak}, jeśli $x \notin A$, przechodzi do stanu q_{nie}. Zbiór instrukcji maszyny nie zależy od A, ale oczywiście od A zależy zbiór słów akceptowanych przez M. Niech P^A (NP^A) oznacza zbiór języków rozpoznawalnych przez deterministyczne (odpowiednio, niedeterministyczne), wielomianowo ograniczone maszyny Turinga z wyrocznią A. T. Baker, J. Gill i R. Solovay [3] wykazali, że jeśli A jest językiem zupełnym w PSPACE, czyli w klasie języków rozpoznawalnych przez maszyny używające taśmy wielomianowo ograniczonej, to P^A=NP^A. Z drugiej strony, każdy dowód oparty na symulacji jednej maszyny przez drugą – a takie jest rozumowanie przekątniowe wykorzystujące istnienie maszyny uniwersalnej, symulującej prace wszystkich maszyn w danej klasie – pozostaje słuszny także dla maszyn z wyrocznią.

Baker, Gill i Solovay [3] zbudowali też wyrocznię B, dla której $\mathrm{P}^B \neq \mathrm{NP}^B$, oraz wyrocznię C taką, że $\mathrm{P}^C \neq \mathrm{NP}^C$, ale co-$\mathrm{NP}^C$=$\mathrm{NP}^C$, gdzie co-$\mathrm{NP}^C$ oznacza klasę dopełnień języków klasy NP^C.

Prace nad maszynami z wyrocznią prowadzono przez wiele lat wykazując, że na wiele pytań można otrzymywać różne odpowiedzi w zależności od wyroczni. Postawiono też pytanie, co dzieje się, jeśli wyrocznia zostanie

[17] Taką maszyną jest zwykły komputer, który może wykonywać różne programy.

wybrana w sposób losowy. C. Bennet i J. Gill [4] wykazali, że z prawdopodobieństwem 1 zachodzi $\mathrm{P}^A \neq \mathrm{NP}^A$ i sformułowali Hipotezę Wyroczni Losowej[18], zgodnie z którą dowolne dwie klasy K_1 i K_2 złożoności obliczeniowej są równe wtedy i tylko wtedy, gdy dla losowej wyroczni A równość $K_1^A = K_2^A$ zachodzi z prawdopodobieństwem 1. Hipoteza ta była nieco naiwna, gdyż korzystając z technik prac [3] i [4] można zbudować wyrocznię A taką, że dla losowej wyroczni B nierówność $(\mathrm{NP}^A)^B \neq (\mathrm{P}^A)^B$ zachodzi z prawdopodobieństwem 1, co oznacza, że Hipoteza Wyroczni Losowej jest fałszywa dla klas NP^A i P^A. Można argumentować, że klasy te są sztuczne i że ich istnienie nie obala Hipotezy Wyroczni Losowej. Hipoteza została jednak ostatecznie obalona. W 1990 roku A. Shamir [29], bazując na wcześniejszych wynikach, które uzyskali C. Lund, L. Fortnow, H. Karloff i N. Nissan [22], udowodnił, że klasa IP problemów, dla których istnieją dowody interaktywne[19], jest równa klasie PSPACE. Z drugiej strony (patrz [7]) okazało się, że dla losowej wyroczni A, nierówność $IP^A \neq PSPACE^A$ zachodzi z prawdopodobieństwem 1.

Izomorfizm zbiorów zupełnych. W 1977 roku L. Berman and J. Hartmanis [5] zauważyli, że wszystkie znane zbiory NP-zupełne są *wielomianowo* izomorficzne, to znaczy, że istnieje między nimi funkcja obliczalna w czasie wielomianowym, która jest wzajemnie jednoznaczna i taka, że funkcja do niej odwrotna jest też obliczalna w czasie wielomianowym (przez maszynę deterministyczną). Berman i Hartmanis postawili hipotezę, że wszystkie zbiory NP-zupełne są izomorficzne. Przez pewien czas Hartmanis uważał, że hipoteza ta może pozwolić na wykazanie, że $\mathrm{NP} \neq \mathrm{P}$. Oczywiście, jeśli $\mathrm{NP} = \mathrm{P}$, to wszystkie zbiory w P, w tym także zbiory skończone, są NP-zupełne i dlatego nie mogą być wszystkie izomorficzne.

Gęstością języka $L \subseteq \Sigma^*$ nazywamy funkcję f taką, że $f(n) = |\{x \in L : |x| = n\}|$. Łatwo zauważyć, że jeśli języki L_1 i L_2 są wielomianowo izomorficzne, to gęstość jednego z nich jest ograniczona przez wielomian od gęstości drugiego. Zbiory *rzadkie*, to zbiory, których gęstość jest ograniczona przez wielomian. Ponieważ wszystkie znane zbiory NP-zupełne mają gęstość wykładniczą, pierwszym krokiem do udowodnienia hipotezy Bermana-Hartmanisa mogłoby być udowodnienie, że nie ma zbiorów rzadkich NP-zupełnych. W 1982 roku S. Mahaney [23] wykazał, że jeśli istnieją rzadkie zbiory zupełne, to $\mathrm{NP} = \mathrm{P}$. W tym samym roku R. Karp i R. Lipton [19] wykazali, że jeśli istnieje rzadka wyrocznia $A \in \mathrm{NP}$ taka, że $\mathrm{NP} \subseteq \mathrm{P}^A$, to

[18] *Random Oracle Hypothesis.*

[19] Dowód interaktywny dla języka L to pewna gra losowa, w której uczestniczą dwaj gracze: *rzecznik* i *weryfikator*. Dla zadanego słowa x zadaniem rzecznika jest przekonanie weryfikatora, że $x \in L$.

hierarchia wielomianowa zapada się do drugiego piętra, to znaczy wszystkie piętra hierarchii wielomianowej poczynając od drugiego, są sobie równe.

Sieci logiczne. W połowie lat osiemdziesiątych dwudziestego wieku pojawiły się nadzieje na rozwiązanie problemu NP = P, a przynajmniej na istotny postęp w badaniach na ten temat. Stało się tak za sprawą serii prac Ajtaia, Håstada, Razborova, Smolensky'ego i innych na temat ograniczeń dolnych na wielkość sieci logicznych obliczających pewne funkcje.

Sieć logiczna (*Boolean circuit*), jest to skończony graf skierowany bez cykli, którego wierzchołki są etykietowane zmiennymi logicznymi lub symetrycznymi funkcjami logicznymi[20]. Mówimy, że wierzchołek u jest poprzednikiem wierzchołka v, jeśli z wierzchołka u prowadzi krawędź do v. Ponieważ sieć jest skończonym grafem acyklicznym, przechodnie domknięcie relacji poprzednika określa na niej relację częściowego dobrego porządku, względem którego można, przez indukcję, definiować funkcje.

Wierzchołki *wejściowe*, czyli takie, do których nie prowadzą żadne krawędzie, są etykietowane zmiennymi. Pozostałe wierzchołki (*bramki logiczne*) są etykietowane funkcjami, przy czym liczba argumentów funkcji etykietującej dany wierzchołek jest równa liczbie krawędzi wchodzących do tego wierzchołka. Wierzchołki, z których nie wychodzą żadne krawędzie, to wierzchołki *wyjściowe* (na ogół rozważa się sieci, które mają jeden wierzchołek wyjściowy)[21]. Sieć logiczna oblicza funkcję logiczną. W każdym wierzchołku sieci, dla zadanego ciągu wartości zmiennych wejściowych, przez indukcję definiuje się wartość sieci w tym wierzchołku. Wartością funkcji obliczanej przez sieć jest wartość sieci w wierzchołku wyjściowym. Niech $x_1, \ldots, x_n$ będą zmiennymi etykietującymi wierzchołki wejściowe sieci. Dla zadanego wartościowania $\sigma : \{x_1, \ldots, x_n\} \to \{0, 1\}$ zmiennych określamy wartość $\sigma^*(v)$ sieci w wierzchołku v. Jeśli wierzchołek v jest etykietowany zmienną x_i, to $\sigma^*(v) = \sigma(x_i)$. Jeśli wierzchołek v jest etykietowany funkcją logiczną ϕ o k argumentach, to v ma k poprzedników $v_1, \ldots, v_k$ i kładziemy $\sigma^*(v) = \phi(\sigma^*(v_1), \ldots, \sigma^*(v_k))$.

Jeśli $x \in \{0, 1\}^*$, to przez $\vec{x}$ oznaczamy ciąg $x_1, \ldots, x_n$ bitów, taki że $x_i = 1$ wtedy i tylko wtedy, gdy w słowie x na i-tym miejscu stoi cyfra 1. Niech $L \subseteq \{0, 1\}^*$ będzie dowolnym językiem (problemem) i niech $L_n = L \cap \{0, 1\}^n$. L_n można utożsamiać z n-argumentową funkcją logiczną $\phi_{L,n}$, taką że $\phi_{L,n}(\vec{x}) = 1$ wtedy i tylko wtedy, gdy $x \in L$. W ten sposób język L będzie opisany przez ciąg funkcji logicznych, zawierającym, dla każdego n, dokładnie jedną n-argumentową funkcję $\phi_{L,n}$.

[20] Funkcja jest symetryczna, jeśli jej wartość nie zmienia się przy permutacji argumentów.

[21] Można oczywiście rozważać funkcje logiczne, których wartościami są wektory wartości logicznych.

Rozmiar sieci, to liczba jej wierzchołków. Dla danej funkcji logicznej $f : \{0,1\}^n \to \{0,1\}$ przez $S(f)$ oznaczamy rozmiar najmniejszej sieci obliczającej f, a dla języka L, jego sieciowa złożoność $NC(L)$, to funkcja określona wzorem $NC(L)(n) = S(\phi_{L,n})$. Łatwo udowodnić, że jeśli $L \in \mathrm{P}$, to $NC(L)$ jest ograniczona przez pewien wielomian. Stąd jedna z możliwych metod udowodnienia, że $\mathrm{NP} \neq \mathrm{P}$, mogłaby polegać na znalezieniu ponadwielomianowego ograniczenia na $NC(L)$ jakiegoś NP-zupełnego języka L. Warto zauważyć, że klasa problemów, które mają wielomianową złożoność sieciową, jest znacznie większa od P. Wynika to stąd, że funkcje logiczne o różnej liczbie zmiennych są obliczane przez różne sieci, natomiast maszyna Turinga akceptuje dane wszystkich możliwych rozmiarów. W szczególności łatwo wykazać, że w klasie problemów o wielomianowej złożoności sieciowej są języki, które nie są nawet rekurencyjnie przeliczalne, a więc nie są rozpoznawane przez maszyny Turinga bez żadnych ograniczeń na czas obliczeń.

Nadzieje na uzyskanie ciekawych wyników oparte były na przekonaniu, że kombinatoryczna analiza sieci logicznych powinna być znacznie łatwiejsza od analizy obliczeń maszyny Turinga. Już w 1949 roku C. Shannon [30] zauważył, że prawie wszystkie funkcje logiczne zależne od n zmiennych mają złożoność sieciową równą co najmniej $\frac{2^n}{n}$.

Niestety, nie udało się znaleźć żadnych silnych ograniczeń dolnych dla dowolnych sieci logicznych. Takie ograniczenia udowodniono tylko dla ograniczonych klas sieci: sieci o ograniczonej głębokości i sieci monotonicznych.

Głębokość sieci logicznej, to maksymalna długość ciągu $v_1, \ldots, v_k$ takiego, że v_i jest poprzednikiem v_{i+1}. W 1984 roku M. Furst, J. Saxe i M. Sipser [12] oraz M. Ajtai [1] wykazali, że dla żadnego d nie ma wielomianowych sieci logicznych głębokości d obliczających funkcję parzystości[22]. Wkrótce potem J. Håstad poprawił ten wynik dowodząc, że nie ma sieci głębokości d rozmiaru $2^{cn^{1/d}}$ obliczających funkcję parzystości dla n zmiennych. Badania nad sieciami o ograniczonej głębokości były prowadzone jeszcze przez kilka lat. W tym niespecjalistycznym tekście trudno te wyniki nawet wymienić, chciałbym jednak wspomnieć o metodzie algebraicznej, która pozwoliła R. Smolensky'emu [31] udowodnić, że jeśli p i q są różnymi liczbami pierwszymi, to nie ma sieci wielomianowych ograniczonej głębokości z bramkami $\vee, \wedge, \neg$ i MOD_q obliczających MOD_p, czyli sprawdzających, czy liczba zmiennych przyjmujących wartość 1 przystaje do 0 modulo p.

Sieci monotoniczne nie zawierają negacji ani zanegowanych zmiennych. Rozważamy funkcję $K_{n,k}$, która zależy od $\binom{n}{2}$ zmiennych reprezentujących istnienie lub brak krawędzi pomiędzy n wierzchołkami grafu. Funkcja ta ma wartość 1, jeśli w grafie reprezentowanym przez zmienne istnieje *klika*

[22] Funkcja taka przyjmuje wartość 1, jeśli parzysta liczba zmiennych wejściowych ma wartość 1.

rozmiaru k, to znaczy zbiór k wierzchołków, z których każde dwa są połączone krawędzią. W 1985 roku A. Razborov [25] wykazał, że każda sieć monotoniczna obliczająca $K_{n,k}$ ma ponadwielomianowy rozmiar.

Systemy dowodowe dla rachunku zdań. Systemów dowodowych dla rachunku zdań jest bardzo wiele. To, co poniżej napiszę, służy wyjaśnieniu o co chodzi w badaniach nad rachunkami zdaniowymi związanych z problemem NP = P, a nie jest wykładem tych systemów. Pełny wykład musiałby być albo bardzo długi albo całkowicie nieczytelny. Z konieczności też to, co napiszę, będzie tylko przybliżeniem prawdy. W szczególności rozważa się niestandardowe systemy dowodowe, które mogą nie spełniać podanych poniżej definicji.

W matematyce, poza jej podstawami, nie używa się pojęcia dowodu formalnego, dlatego przypomnienie tego pojęcia wydaje mi się niezbędnie. *Dowód formalny* (dla rachunku zdań), to skończony ciąg zdań (formuł zdaniowych). Każde z nich ma postać należącą do ustalonego skończonego zbioru aksjomatów lub może zostać wyprowadzone ze zdań wcześniejszych przy pomocy jednej z kilku reguł dowodzenia. Z punktu widzenia badań, o których będziemy pisać, istotne są dwie cechy: *wielomianowa weryfikowalność* i *zupełność*. Wielomianowa weryfikowalność oznacza, że dla danego ciągu zdań możemy deterministycznie, w czasie wielomianowym sprawdzić, czy jest on dowodem, natomiast zupełność oznacza, że zdanie jest tautologią wtedy i tylko wtedy, gdy ma dowód, czyli jest elementem ciągu, który jest dowodem. Rozważa się też abstrakcyjne systemy dowodowe, które nie muszą mieć nic wspólnego z klasyczną teorią dowodu: system dowodowy dla języka L, to deterministyczny algorytm, który dla danych ciągów w (problemu) i x (dowodu) w czasie wielomianowym akceptuje lub odrzuca parę $\langle w, x\rangle$, przy czym algorytm ten jest zupełny, to znaczy że $w \in L$ wtedy i tylko wtedy, gdy istnieje x taki, że para $\langle w, x\rangle$ zostanie zaakceptowana.

Celem badań prowadzonych nad systemami dowodowymi dla rachunku zdań jest rozstrzygnięcie problemu, czy istnieje wielomianowy system dowodowy, to znaczy taki, w którym każda tautologia ma dowód, którego długość jest ograniczona przez wielomian od jej rozmiaru. W wyżej podanej abstrakcyjnej definicji oznacza to, że długość ciągu x jest wielomianowa względem długości w. Łatwo zauważyć, że dla SAT istnieje abstrakcyjny wielomianowy system dowodowy, sprawdzający dla danej formuły w i danego wartościowania x, czy w jest spełniona przy tym wartościowaniu.

Jak już wcześniej wspominałem, spełnialność formuł jest problemem NP-zupełnym. Zdania spełnialne, to zdania, których negacja nie jest tautologią, czyli nie posiada dowodu. Stąd wynika, że istnienie wielomianowego systemu dowodowego dla tautologii jest równoważne temu, że NP=co-NP (S. Cook, R. Reckhow [9]).

Mało kto wierzy w możliwość znalezienia wielomianowego systemu dowodowego. Większe są nadzieje na udowodnienie, że takiego systemu nie ma, albo co najmniej, że znane systemy dowodzenia takie nie są. Niestety wciąż daleko jesteśmy od udowodnienia takich twierdzeń. Pierwszy wynik na ten temat uzyskał A. Haken [13], który wykazał, że *metoda rezolucji* nie jest wielomianowym systemem dowodowym, gdyż wyprowadzenie w tym systemie zasady szufladkowej Dirichleta wymaga wykładniczej liczby kroków. Metoda rezolucji wprowadzona przez M. Davisa i H. Putnama [10], jest systemem dowodowym, który dobrze nadaje się do budowy komputerowych systemów automatycznej dedukcji. Stanowi teoretyczną bazę dla języków programowania logicznego takich jak PROLOG [26].

Systemy dowodowe dla rachunku zdań można poklasyfikować ze względu na ich siłę dowodową mierzoną długością dowodów. Metoda rezolucji jest słabym systemem, a rozszerzone systemy Fregego są bardzo silne. Nie znaleziono dla nich ograniczeń dolnych na długość dowodów. Gdzieś w środku skali są systemy Fregego o ograniczonej głębokości. Są to najsilniejsze systemy dowodowe, o których wiadomo, że nie są wielomianowe (M. Ajtai [2], T. Pitassi, P. Beame i R. Impagliazzo [24] oraz J. Krajicek P. Pudlak i A. Woods [20]).

Badania nad systemami dowodowymi nie przyniosły rozwiązania problemu „czy NP = P?" i prawdopodobnie nie przyniosą. Jest to jednak dziedzina w której wciąż pracują doskonali matematycy dowodząc nowych, ważnych twierdzeń i stawiając ciekawe pytania.

Literatura

[1] M. Ajtai, *Σ_1^1-formulae on finite structures*, Annals of Pure and Applied Logic 24 (1983), 1–48.

[2] M. Ajtai, *The complexity of the pigeonhole principle*, Proc. 29th IEEE FOCS (1988), 346–355.

[3] T. Baker, J. Gill, and R. Solovay, *Relativisation of the P=?NP question*, SIAM Journal on Computing 4 (1975), 431–442.

[4] C. Bennet and J. Gill, *Relative to a random oracle A, $P^A \neq NP^A \neq co{-}NP^A$, with probability 1*, SIAM Journal on Computing 10 (1981), 96–113.

[5] L. Berman and J. Hartmanis, *On isomorphism and density pf NP and other complete sets*, SIAM Journal on Computing 6 (1977), 305–322.

[6] A. A. Bulatov, *A dichotomy theorem for constraints on a three-element sets*, Proceedings of the 43rd Symposium on Foundations of Computer Science (2002), 649–658.

[7] R. Chang, B. Chor, O. Goldreich, J. Hartmanis, J. Håstad and D. Ranjan, *The random oracle hypothesis is false* Journal of Computer and System Sciences 49 (1994), 24–39.

[8] S. A. Cook, *The complexity of theorem proving procedures,* Proc. 3rd ACM STOC (1971), 151–158.

[9] S. A. Cook and R. Reckhow, *The relative efficiency of propositional proof systems*, Journal of Symbolic Logic 44 (1979), 36–50.
[10] M. Davis and H. Putnam, *A computing procedure for quantification theory*, Journal of ACM 7 (1960), 201–215.
[11] R. Fagin, *Generalized first-order spectra and polynomial time recognizable sets*, Complexity of Computations, SAIM-AMS Proceedings 7 (1974), 43–73.
[12] M. Furst and J. B. Saxe and M. Sipser, *Parity, circuits and the polynomial time hierarchy*, Mathematical System Theory 17 (1984), 13–27.
[13] A. Haken, *The intractability of resolution*, Theoretical Computer Science 39 (1985), 297–308.
[14] N. Immerman, *Relational queries computable in polynomial time*, Proc. 14th ACM STOC (1982), 147–152. Pełna wersja Information and Control (1986), 86–104.
[15] N. Immerman, *Nondeterministic space is closed under complementation*, SIAM Journal on Computing 17 (1988), 935–938.
[16] N. Immerman, *Descriptive Complexity*, Springer Verlag (1999).
[17] N. D. Jones and A. L. Selman, *Turing machines and the spectra of first-order formulas*, Journal of Symbolic Logic 39 (1974), 139–150.
[18] R. Karp, *Reducibility among combinatorial problems*, Complexity of Computer Computations (1972), 85–103.
[19] R. Karp and R. Lipton, *Turing machines that take advice,* L'Ensegnement Mathématique 28 (1982), 191–209.
[20] J. Krajicek, P. Pudlak, and A. Woods, *Exponential lower bounds to the size of bounded depth Frege proofs of the pigeonhole principle*, Random Structures and Algorithms 7 (1995), 15–39.
[21] R. E. Ladner, *On the structure of polynomial time reducibility*, Journal of ACM 22 (1975), 155–171.
[22] C. Lund, L. Fortnow, H. Karloff, and N. Nisan, *Algebraic methods for interactive proof systems*, Journal of ACM 39 (1992), 859–868.
[23] S. Mahaney, *Sparse complete sets for NP Solution of a conjecture of Berma and Hartmanis*, Journal of Computer and System Science 25 (1982), 130–143.
[24] T. Pitassi, P. Beame, and R. Impagliazzo, *Exponential lower bounds for the pigeonhole principle*, Computationa Complexity 3 (1993), 97–140.
[25] A. Razborov, *Lower bounds on the monotone complexity of some Boolean functions*, Doklady Akademi Nauk SSSR 281 (1985), 798–801.
[26] J. A. Robinson, *A machine oriented logic based on resolution principle*, Journal of ACM 12 (1983), 23–41.
[27] W. Savitch, *Relationship between nondeterministic and deterministic tape classes*, Journal of Computer and System Science 4 (1970), 177–192.
[28] T. J. Schaefer, *The complexity of satisfiability problem*, Proc. 10th ACM Symposium on Theory of Computing (1978), 216–226.
[29] A. Shamir, *IP=PSPACE*, Journal of ACM 39 (1992), 869–877.
[30] C. E. Shannon, *The synthesis of two-terminal switching circuits*, Bell Systems Technical Journal 28 (1949), 59–98.
[31] R. Smolensky, *Algebraic methods in the theoryof lower bounds for Boolean circuit complexity*, Proc. 19th Annual ACM Symposium on Theory of Computing (1987), 77–82.
[32] R. Szelepcsnényi, *The method of forced enumeration for nondeterministic automata*, Acta Informatica 26 (1988), 279–284.
[33] M. Vardi, *Complexity of relational query languages*, Proc. 14th ACM STOC (1982), 137–146.