Sławomir Górawski
github.com/sgorawski
lato 2020

1

Zakładamy że istnieje DFA A rozpoznający język L, który ma mniej niż 1024 stany.
Bierzemy dwa dowolne różne słowa w1, w2 z alfabetu {0, 1} dł. 10 takie, że po ich przejściu A kończy w tym samym stanie q1. (Słów jest 2^10, a stanów mniej, zasada szufladkowa). Słowa muszą się różnić na jakiejś pozycji i. Następnie dopisujemy do każdego z tych słów (10 - i) zer. Po przejściu wydłużonej tych słów A skończy w stanie q2 (akceptującym bądź nie), takim samym dla w1 i w2. Sprzeczność, gdyż w1 należy do L, a w2 nie (BSO).

2

Niech L – język z zadania.
Bierzemy zbiór A3 = { w ∈ {a, b, c}* | |w| ≤ 3 }. |A3| = 16, pokażemy, że tyle stanów musi mieć DFA A rozpoznający L. Załóżmy, że ma mniej, bierzemy w1, w2 ∈ A3 takie że po ich przejściu A kończy w tym samym stanie q1. Słowa muszą się różnić na jakiejś pozycji i ≤ 2. Rozważamy przez przypadki, że zawsze możemy dopisać taki ciąg za w1 i w2, że w1 będzie należeć do L, a w2 nie.

3

Udowodnij, że język L = { a^n b^2n | n ∈ N } nie jest regularny.

Z lematu o pompowaniu.
https://www.wikiwand.com/en/Pumping_lemma_for_regular_languages

Załóżmy, że L jest regularny. Wtedy istnieje takie p ≥ 1, że każde słowo w ∈ L takie że |w| ≥ p może być podzielone na w = xyz, tak że:

- |y| ≥ 1,
- |xy| ≤ p,
- dla każdego n ≥ 0 (x y^n z) należy do L.

Intuicja: p - liczba stanów DFA rozpoznającego L.

Załóżmy, że L jest regularny. Wtedy istnieje takie p, weźmy to p. Słowo w = a^p b^2p należy do L. W takim razie istnieje podział, …, niech xyz = w. Wiemy, że |y| > 1 i |xy| ≤ p, w takim razie x = a^|x|, y = a^|y|. Wtedy dla każdego n ≥ 0 (a^|x| a^n|y| a^|2p - |z|| b^2p) należy do L, co daje sprzeczność dla n ≠ 1.

4

https://cs.stackexchange.com/questions/10013/if-l-is-a-subset-of-0-then-how-can-we-show-that-l-is-regular

35

https://www.wikiwand.com/en/Synchronizing_word

a. Tak, dla A, dla sync(S) konstruujemy DFA który go rozpoznaje, gdzie stany to zbiory stanów z Q (P(Q)), f. przejścia z A tylko po zbiorach, stan początkowy – zbiór wszystkich stanów S, stany akceptujące – każdy singleton z P(Q).
b. Nie, bo w vw możemy “wypaść” poza S i wtedy to, że w ∈ sync(S) nic nam nie da. Samo wv’ raczej tak. Przydałby się jakiś kontrprzykład.
c. Tak, to co było wcześniej nie ma znaczenia, a później idziemy deterministycznie.

36

a. http://www.math.uni.wroc.pl/~kisiel/auto/volkov-surv.pdf
   Jeżeli dla danego 2-elementowego S sync(S) niepusty, to istnieje w nim w t. że |w| < |Q|^2. Konstruujemy pomocniczy automat, którego stany to 1- lub 2-elementowe zbiory stanów z A. Jest ich |Q|(|Q| + 1)/2. Stan startowy to S, stany akceptujące to singletony. Z założenia wiemy, że istnieje jakaś ścieżka między stanem startowym a jednym z akceptujących, więc musi być jakaś o długości nie większej niż |Q|^2.
b. https://pdfs.semanticscholar.org/7d28/e902a741176f75db63731e74373d9d33e159.pdf
   Jeżeli sync(Q) jest niepusty, to istnieje w nim w takie że |w| < |Q|^3. Pokażemy, jak skonstruować takie w. Intuicja – bierzemy dwa stany z Q, i idziemy z nich do jednego ścieżką v, gdzie |v| < |Q|^2, potem z |Q| - 1 takich ścieżek sklejamy w.
   - Q’ := Q
   - w := ε
   - dopóki |Q’| > 1:
     - weź dowolne q1, q2 z Q’
     - znajdź takie v, że v ∈ sync({q1, q2}) ∧ |v| < |Q|^2 (jak w a.)
     - w := wv
     - Q’ := Q’ ∪ d(q1, v) // albo q2, wszystko jedno

37

To może być wprost automat z papera Cernego (link w 36 a.). Ale pewnie można lepiej, bo u nas S ma być tylko jeden dla danego A.

38

Chyba tak, argument analogiczny jak w 36 b.

39

a. Analogicznie jak 36 a.?
b. Zbiór potęgowy stanów A, pokazujemy że istnieje ścieżka ze zbioru wszystkich stanów do singletonu, w takim razie musi być taka krótsza niż 2^|Q|

82

A ⊆ N × N jest rekurencyjny, jeśli istnieje całkowita funkcja rekurencyjna f: N × N → N taka że:

f(a, b) = {
1 : (a, b) ∈ A,
0 : wpp
}

Jeśli nie możemy zdefiniować funkcji rekurencyjnych dla N × N → N, to możemy przekształcić pary N × N na liczby z N funkcją Cantora (po przekątnych), albo na 2^a 3^b itp.

T: B = { n | istnieje m takie że (n, m) ∈ A } jest r.e.

Weźmy dowolny A ⊆ N × N rekurencyjny, niech ϕ_A – program rozstrzygający A. Konstruujemy program sprawdzający przynależność n do B:

- wczytaj n
- dla każdego m ∈ N:
  - jeśli ϕ_A(n, m) = 1, zwróć 1

Jeśli n ∈ B, to istnieje jakieś m takie że (n, m) ∈ A, program w końcu je znajdzie.
Jeśli nie, to nigdy się nie zakończy.

83

T: Dla danego B r.e. istnieje A ⊆ N × N rekurencyjny.

Weźmy dowolny B r.e., niech ϕ_B – program rozstrzygający B.
Niech A = { (n, m) | n ∈ B, ϕ_B(n) zatrzymuje się po m krokach }.
Program rozstrzygający A :

- wczytaj (n, m)
- dla _ od 0 do m:
  - wykonaj 1 krok ϕ_B(n)
  - jeśli otrzymano wartość, zwróć ją
- zwróć 0

85

T: A = { n | |Dom(ϕ_n)| ≥ 7 } jest r.e.

Idea: chcemy sprawdzić, czy istnieje przynajmniej 7 argumentów, dla których ϕ_n się zatrzymuje. Nie możemy jednocześnie sprawdzić wszystkich z A, ale możemy robić coś w stylu Cantora – iść przekątnymi po parach (x, m), gdzie x to argument, zaś m to liczba sprawdzonych kroków dla wszystkich argumentów nie większych niż x.

Pokażemy program rozstrzygający A:

- wczytaj n
- znajdź ϕ_n
- known_domain := ∅
- dla x ∈ N:
    - dla i od 0 do x, i ∉ known_domain:
        - wykonaj 1 krok ϕ_n(i)
        - jeśli otrzymano wartość:
            - known_domain ∪= {i}
    - jeśli |known_domain| >= 7:
        - zwróć 1

86

A, B, C, D – zbiory r.e. takie że każda liczba naturalna należy do dokładnie dwóch z nich.

T: A, B, C, D rekurencyjne.

BSO weźmy A, ϕ_A – program semirozstrzygający A. Pokażemy program rozstrzygający A:

- wczytaj n
- n_in_sets := ∅
- dopóki |n_in_sets| < 2:
    - wykonaj 1 krok ϕ_A
    - jeśli otrzymano wartość, zwróć ją
    - dla X ∈ {B, C, D}, x ∉ n_in_sets:
        - wykonaj 1 krok ϕ_X
        - jeśli otrzymano 1:
            - n_in_sets ∪= {X}
- zwróć 0

87

ϕ – niemalejąca, całkowita funkcja rekurencyjna

T: Zbiór wartości ϕ jest rekurencyjny.

Program rozstrzygający ϕ[N]:
- wczytaj n
- dla i ∈ N:
    - x := ϕ(i)
    - jeśli x = n, zwróć 1
    - jeśli x > n, zwróć 0

Czy pozostaje to prawdą bez założenia o całkowitości?
Nie. Kontrprzykład: f(n) = ϕ_n(n), wtedy f[N] = K.

88

T: Każdy niepusty zbiór r.e. jest zbiorem wartości pewnej całkowitej funkcji rekurencyjnej.

Weźmy dowolny niepusty A r.e. i jego ϕ_A. Skonstruujemy f taką że f[N] = A.

Idea: dla danego n będziemy przeszukiwać jego otoczenie, z każdą iteracją rozszerzając je i wykonując jeden krok ϕ_A dla każdego elementu w tym otoczeniu, aż któryś ϕ_A zwróci 1 i znajdziemy element z A, który możemy zwrócić. Zawsze znajdziemy, bo A niepusty.

Program:
- wczytaj n
- lower, upper := n, n
- powtarzaj:
    - dla i od lower do upper:
        - wykonaj 1 krok ϕ_A(i)
        - jeśli otrzymano 1, zwróć i
    - lower -= 1
    - upper += 1

91

https://math.stackexchange.com/questions/1682675/example-of-a-recursive-set-s-and-a-total-recursive-function-f-such-that-fs
https://www.cs.toronto.edu/~sacook/csc438h/notes/page71.pdf
f – funkcja rekurencyjna całkowita

1

T: Jeśli A jest rekurencyjny, to f[A] też.

No raczej nie, wtedy jeśli A byłby rekurencyjny i $A \leq_{rek} B$, to B też.
Kontrprzykład, korzystający z 82 / 83 – weźmy A = N × N i g: N × N → N tak że g(n, m) zwraca numer ϕ najbliższego n, które zatrzymuje się po m krokach.

Wtedy f[A] = K, K nie jest rekurencyjny.

2

T: Jeśli A jest rekurencyjny, to $f^{-1}[A]$ też.

Tak, odpowiada to temu, że jeśli $A \leq_{rek} B$ i B rekurencyjny, to A też.
Niech g – program rozstrzygający $f^{-1}[A]$, g(n) = ϕ_A(f(n)).

3

T: Jeśli A jest r.e., to f[A] też.

Tak. Weźmy g: N → N, tak że g(x) będzie sprawdzać kolejne a ∈ A, szukając takiego, dla którego f(a) = x. (Oczywiście musimy ograniczać ilość kroków). Jeżeli nie wiemy, które a są w A, to sprawdzamy wszystkie liczby naturalne i nasz warunek to ϕ_A(a) = 1 oraz f(a) = x.

4

T: Jeśli A jest r.e., to $f^{-1}[A]$ też.
Tak, jak 2.

Co zmieni się, jeśli założymy, że f jest funkcją częściową?
2 nie będzie prawdziwe, reszta bez zmian (?).

95

Chcemy w prosty sposób odwzorować wszystkie elementy z A i B.
Niech $C = \{ (a, 0) \mid a \in A \} \cup \{ (b, 1) \mid b \in B \}$.

1

T: $A \leq_{rek} C$ i $B \leq_{rek} C$

r_AC, r_BC: $N \to N$
r_AC(n) = (n, 0)
r_BC(n) = (n, 1)

2

T: Jeśli D jest taki że $A \leq_{rek} D$ i $B \leq_{rek} D$, to $C \leq_{rek} D$.

Weźmy r_AD i r_BD: $N \to N$ – redukcje z założenia. Wtedy:

```
r_CD(n, m) = {
        r_AD(n) : m = 0,
        r_BD(n) : m = 1
}
```

96

1

T: $K \leq_{rek} K'$.

Nie. Załóżmy nie wprost, że tak, niech r – redukcja.
Wiemy, że:

1. $x \in K \Rightarrow r(x) \in K'$,
2. $x \notin K \Rightarrow r(x) \notin K'$.

Można to zapisać równoważnie, korzystając z faktu, że $x \notin K \Leftrightarrow x \in K'$:

1. $x \notin K' \Rightarrow r(x) \notin K$,
2. $x \in K' \Rightarrow r(x) \in K$.

Czyli r działa też w drugą stronę, jako redukcja z K' do K.
Więc ten podpunkt jest równoważny drugiemu.

2

T: K' $\leq_{rek}$ K.

Nie, bo wtedy K byłby rekurencyjny.

Ogólnie w tym zadaniu w nie ma znaczenia, czym jest K i tak samo działa dla dowolnych A, A'.

94

Udowodnij, że zbiór numerów tych programów, które zatrzymują się dla wszystkich argumentów oprócz co najwyżej skończonej liczby, nie jest rekurencyjnie przeliczalny.

Czyli **nie zatrzymują się** dla skończonej liczby argumentów.

Niech A – ten zbiór, załóżmy nie wprost, że A r.e., ϕ_A – program semirozstrzygający przynależność do A.

Wtedy możemy skonstruować f semirozstrzygającą przynależność do K':
(Przypomnienie: K' = N \ K = { n ∈ N | ϕ_n(n) nie zatrzymuje się nigdy }).

- wczytaj n
- t := numer tego programu:
    - wczytaj k
    - wykonaj k kroków ϕ_n(n)
    - jeśli otrzymano wynik, zapętl się
    - zwróć 1
- zwróć ϕ_A(t)

Czy ten program semirozstrzyga przynależność do N \ K?

- Weźmy dowolne n ∈ K'
    - ϕ_n(n) nie zatrzyma się nigdy,
    - dla dowolnego k ϕ_t(k) zwróci 1,
    - ϕ_A(t) zwróci 1.
- Weźmy dowolne n ∉ K' (n ∈ K)
    - istnieje k' takie że ϕ_n(n) zwraca wynik po k' krokach,
    - dla każdego k ≥ k' ϕ_t(m) się zapętli,
    - ϕ_A(t) nie zwróci 1.

Wynika z tego, że K' jest r.e., sprzeczność.

97

T = { (n, m) ∈ N × N | ϕ_n i ϕ_m to ta sama funkcja częściowa }

i

Analogicznie jak w 94, załóżmy nie wprost, że T jest r.e., niech ϕ_T: N × N → N – program semirozstrzygający przynależność do T, pokażemy, że wtedy da się skonstruować program semirozstrzygający przynależność do K'.

Intuicja: niech nasz program dla danego n użyje ϕ_T, żeby sprawdzić, czy "programom" p(_) = ϕ_n(n) i u(_) = ⊥ odpowiada ta sama funkcja częściowa.

- wczytaj n
- p := numer tego programu:
    - wczytaj _
    - zwróć ϕ_n(n)
- zwróć ϕ_T(p, u) // u jest taki sam dla każdego n

Dowód, że ten program semirozstrzyga przynależność do K':

- Dla n ∈ K':
    - p dla każdego wejścia się zapętla,
    - ϕ_T(p, u) zwraca 1.
- Dla n ∉ K' (n ∈ K):
    - p dla każdego wejścia zwraca tą samą liczbę,
    - ϕ_T(p, u) nie zwraca 1.

Wynika z tego, że K' jest r.e., sprzeczność.

ii

T' = (N × N) \ T = { (n, m) | ϕ_n i ϕ_m to różne funkcje częściowe }

Intuicja: jak w i., ale porównujemy p' = 1 ("program" stały) z p:

- wczytaj _
- wykonaj ϕ_n(n)
- zwróć 1

Dowód, że to semirozstrzyga przynależność do K':

- Dla n ∈ K':

    - p dla każdego wejścia się zapętla,
    - ϕ_T(p, p') zwraca 1.
- Dla n ∈ K:
    - p dla każdego wejścia zwraca 1,
    - ϕ_T(p, p') nie zwraca 1.

Wynika z tego, że K' jest r.e., sprzeczność.

98

https://math.stackexchange.com/questions/1949380/is-the-set-of-indices-of-partial-computable-functions-with-finite-domains-r-e
https://www.wikiwand.com/en/Tarski%E2%80%93Kuratowski_algorithm
f: N × N → N
L = { n | f_n – funkcja rekurencyjna niepusta i Dom(f_n) skończona }
Jakie jest najmniejsze i, dla którego zachodzi L ∈ ∑i?

1

T: L ∈ ∑2
Pokażemy, że istnieje B co-r.e. taki że
A = { n ∈ N | istnieje m takie że f(n, m) ∈ B }

Niech B = { f(n, m) | f_n niepusta i zdefiniowana tylko dla i < m }
B jest co-r.e. Dowód:

1. $B \leq_{rek} K'$,
2. B nie jest rekurencyjny.

2

T: L ∉ ∑1
Czyli L nie jest r.e.

Dowód: załóżmy nie wprost, że jest, niech ϕ_L – program semirozstrzygajacy L, wtedy możemy skonstruować program semirozstrzygający przynależność do N \ K.

99

A, B ⊆ N
f: A → B – redukcja z A w B
f[N] = N

T: $B \leq_{rek} A$

Wiemy, że $x \in A \Leftrightarrow f(x) \in B$
Chcemy skonstruować g – funkcję rekurencyjną całkowitą taką że $x \in B \Leftrightarrow g(x) \in A$.
f nie musi mieć funkcji odwrotnej, ale na pewno dla każdego y jesteśmy w stanie znaleźć x takie że y = f(x) w skończonym czasie.

g(x):
- znajdź x' takie że f(x') = x
- zwróć x'

Dowód, że to redukcja:
1. $x \in B$
   Wtedy znajdujemy x' takie że f(x') = x, więc $f(x') \in B$, więc $x' \in A$.
2. $x \notin B$
   Wtedy znajdujemy x' takie że f(x') = x, więc $f(x') \notin B$, więc $x' \notin A$.

100

a. Nie, bo jego dopełnienie nie jest r.e.

Dopełnienie B – N \ B – zbiór takich programów, których dziedzina NIE jest odcinkiem początkowym N.

Pokażemy, że $N \setminus K \leq_{rek} N \setminus B$

Niech r – redukcja, która dla n zwraca numer takiego programu.
- wczytaj m
- wykonaj m kroków ϕ_n(n)
- jeśli otrzymano wynik, zapętl się
- zwróć 1

Dowód, że to redukcja (n ∈ K ⇔ r(n) ∈ B):

- Dla n ∈ K:
  - ϕ_n(n) zwróci wynik w m krokach lub więcej;
  - r(n) zwraca wynik dla argumentów [1, m);
  - r(n) ∈ B.
- Dla n ∉ K:
  - ϕ_n(n) nie zatrzyma się nigdy;
  - r(n) zawsze zwraca wynik;
  - dziedziną r(n) jest N, czyli nie jest to odcinek początkowy;
  - r(n) ∉ B.

b. Tak

Niech A = { (n, m, k) | dziedziną ϕ_n jest [1, m] i dla każdego argumentu z tej dziedziny ϕ_n zatrzymuje się w k krokach }.

Pokażemy, że A jest co-r.e. Równoważnie N \ A jest r.e.
Program n należy do N \ A, jeśli nie zwraca wartości dla któregoś z [1, m] w k krokach lub zwraca wartość dla jakiegoś m' > m.

Program semirozstrzygajacy przynależność do N \ A:

- wczytaj n, m, k
- dla i od 1 do m:
  - wykonaj k kroków ϕ_n(i)
- jeśli nie otrzymano wyniku dla któregokolwiek i, zwróć 1
- dla i, j w porządku Cantora dla [m + 1, \inf) × [0, \inf):
  - wykonaj j kroków ϕ_n(i)
  - jeśli otrzymano wynik, zwróć 1

101

Zbiór na pewno jest rekurencyjny, jeśli jest skończony albo jego dopełnienie jest skończone.

102

Rozwiązanie jak w 99

1. Tak, idziemy od 0 w górę i w końcu coś znajdziemy.
2. Tak, idziemy po przekątnych (argument i ilość kroków) i w końcu coś znajdziemy.

103

D ⊆ P(N)

Wskazówka: mając daną redukcję f, $A = f^{-1}[B]$ (przeciwobraz zbioru B przez f) jest tylko jeden.

$R \Longrightarrow L$

Jeśli istnieje $B \subseteq N$ taki że dla każdego $A \in D$ zachodzi $A \leq_{rek} B$, to D jest przeliczalny.
Ze wskazówki zbiorów A jest tyle, co redukcji w B.
Tych jest przeliczalnie wiele (bo odpowiadają im programy), więc D jest przeliczalny.

$L \Longrightarrow R$

D jest przeliczalny $\Longrightarrow$ Istnieje bijekcja $b: N \to D$.
Niech Cantor – bijekcja $N \times N \to N$.

Wtedy niech $B = \{ Cantor^{-1}(b^{-1}(A), a) \mid a \in A, A \in D \}$.

Dla danego A redukcją jest $r(a) = Cantor^{-1}(b^{-1}(A), a)$.

104

a. Tak,

Niech r będzie redukcją, która dla n zwraca numer takiego programu:
- wczytaj _
- znajdź (przekątniowo) jakiś argument, dla którego M_n zwróci wynik
- zwróć 1

Jeżeli $n \in Nemp$, to r(n) zwróci wynik dla każdego $n \in N$ (ten sam).
Jeżeli nie, to M_n nie zatrzyma się dla żadnego argumentu, więc r(n) się zapętli.

b. Nie, bo Nemp jest r.e., a Tot nie.

Program semirozstrzygający Nemp – uruchamiamy przekątniowo M_n i zwracamy 1 gdy dostaniemy jakiś wynik.

Dowód, że Tot nie jest r.e. – pokażemy, że $N \setminus K \leq_{rek} Tot$.

Niech r będzie redukcją, która dla danego n zwraca numer następującego programu:
- wczytaj m
- uruchom m kroków ϕ_n(n)
- jeśli otrzymano wynik, zapętl się
- zwróć 1

105

Nie. Załóżmy nie wprost, że tak.
Niech f = { (n, k) | ϕ_n(n) zwraca wynik w k krokach }

Niech f' – całkowita funkcja rekurencyjna zawierająca f.

Korzystając z f' możemy napisać program rozstrzygający K:

- wczytaj n
- k := f(n)
- wykonaj k kroków ϕ_n(n)
- jeśli otrzymano wynik, zwróć 1, wpp zwróć 0

K nie jest rekurencyjny, co daje sprzeczność.

https://www-m5.ma.tum.de/foswiki/pub/M5/Allgemeines/MA5116_2012S/lecture.pdf

112

https://www.wikiwand.com/en/Counter_machine

117

Ogólnie proste – robimy jak dla normalnego problemu słów, nie interesuje nas w jakim stanie się zatrzyma, ale czy w ogóle się zatrzyma. Sam fakt skończonej ilości konfiguracji nie wystarczy – możemy mieć ich skończoną ilość, ale cyklicznie przechodzić między nimi w nieskończoność. Dlatego dodajemy licznik ze wskazówki. Problem jest z tym, gdzie go dodać – jeśli na poziomie produkcji Thuego, to to jest dosyć trudne, bo potrzebujemy stanu, symbolu z prawej i symbolu z lewej, więc nie ma chyba gdzie go "wcisnąć". Jeśli na poziomie samej maszyny (jak to by wynikało ze wskazówki), to spoko. Ale tutaj trzeba pokazać, że dla dowolnej maszyny Turinga można "gdzieś na taśmie" dodać licznik kroków, co też jest trudne, bo co jeśli maszyna potrzebuje nieskończonej taśmy w dwie strony, a liczby kroków też nie możemy z góry ograniczyć? Machanie rękami…

118

To jest jak 110, tylko bez stanu. To jest niby deterministyczne…

128

Niech P(∑, Π, w, v) – problem z zadania.

Jakby dodać warunek, że v niepuste, to tak. (Wystarczyłoby przeglądać wszystkie możliwe transformacje BFSem wg rosnącej długości).

Nie mamy tego warunku, ale chyba bez niego problem też jest rozstrzygalny. Wiemy, że problem, czy z danej CFG można wygenerować słowo w, jest rozstrzygalny. Pokażemy redukcję z P do tego problemu.

Konstruujemy CFG odpowiadającą P.

1. Niech w = w_1w_2w_3… Zaczynamy od produkcji:
   S → W_1W_2W_3…
2. Dla każdego a_i ∈ ∑ dodajemy produkcję:
   A_i → a_i (terminal)
3. Dla każdego (w, v) ∈ Π, gdzie v = v_1v_2v_3… dodajemy produkcję:
   W → V_1V_2V_3…

Nasza CFG – krotka z rozszerzoną ∑, S, nowymi produkcjami i czymś jeszcze.
Wtedy P jest równoważny zapytaniu, czy z tej CFG da się wygenerować v.
Konwertujemy CFG do postaci normalnej Chomsky'ego i sprawdzamy wszystkie produkcje do 2|v| - 1 kroków.

129

Nie. Niech problem w zadaniu – P. Pokażemy, że semiThue $\leq_{rek}$ P.
Weźmy dowolny problem semiThue(∑, Π, w, v).
Skonstruujemy z Π nowe produkcje dla Π_P, być może rozszerzając ∑ o nowe symbole.

1. Dla każdego w → v ∈ Π, jeśli |w| > 1 i |v| > 1, rozszerzamy ∑ o nowy symbol WV i dodajemy zasady do Π_P:
   w → WV
   WV → v
2. Dla każdego a → v ∈ Π_P, jeśli |v| > 2, niech v = v_1v_2… Rozszerzamy ∑ o nowy symbol V[i:] i dodajemy produkcje do Π_P:
   a → V[1:]
   V[1:] → v_1V[2:]
   V[2:] → v_2V[3:]
   …
3. Dla każdego v → a ∈ Π_P, jeśli |v| > 2, niech v = v_1v_2…v_k. Dodajemy produkcje:
   v_k-1v_k → V[k-1:]
   v_k-2v_k-1 → V[k-2:]
   …
   V[1:] → a
   (Być może tu jest kolizja z 2., wtedy "zwijamy" od początku używając V[:i]).

semiThue nie jest rozstrzygalny, więc P też nie.

130

Niech problem w zadaniu – P. Tutaj możemy pokazać, że $K \leq_{rek} P$, jeżeli pokażemy, że da się tutaj zakodować działanie maszyny Turinga (przejścia między konfiguracjami). Mamy $\sum = \{0, 1\}$, wychodzimy od 1111, więc możemy wpisywać lub kasować zera pomiędzy tymi jedynkami. Korzystając z zad. 112, moglibyśmy zakodować 2 liczniki i stan maszyny unarnie w 3 miejscach. Czemu nie 4 liczniki? Bo musimy wiedzieć, gdzie co się zaczyna i kończy, np. produkcja 001111 → 0111 zadziałałaby też dla 0001111, a tego nie chcemy.

Wtedy nasze produkcje są postaci:

1111 → 11q_011
1a1q1b1 → 1a'1q'1b'1

Dla q_1, a, b → q_2, dec(a), inc(x)(b): ($x \in \{2, 3, 5, 7\}$)

01q_11 → 1q_210^x

I chcemy sprawdzić, czy da się przez takie produkcje wygenerować 11q_F11 (puste liczniki, chyba zawsze łatwo je wyzerować).

114

Niech P114 – problem z zadania, tj: "Dla CFG G, H czy $L_G \cap L_H \neq \varnothing$"?

T: $PCP \leq_{rek} P114$

Dają nam $\sum$ oraz g, h – listy słów z $\sum^*$. Chcemy rozstrzygnąć PCP przy użyciu P114.
Musimy skonstruować 2 CFG, które będą generowały odpowiednie konkatenacje słów z g, h.

G → g[0]G0 | g[1]G1 | … | g[n]Gn | #
H → h[0]H0 | h[1]H1 | … | h[n]Hn | #

Wtedy istnieje w generowane przez obie gramatyki, jeśli zgadza się rezultat (konkatenacja słów) oraz (odwrócony) ciąg indeksów, przy użyciu którego został uzyskany, czyli równoważne PCP.

Czy P114 jest r.e.? Tak, przeszukujemy BFS-em grafy produkcji G i H, w końcu znajdziemy jakieś wspólne słowo, jeśli takie istnieje.

115

Niech P115 – problem z zadania, tj. "Dla $\sum$ i CFG G, czy $L_G = \sum^*$"?

T: $PCP \leq_{rek} P114 \leq_{rek} P115$

Dają nam G i H. Chcemy rozstrzygnąć, czy $L_G \cap L_H = \varnothing$.
Tworzymy $L_K = L_{G'} \cup L_{H'} = (L_G \cap L_H)'$, sprawdzamy, czy $L_K = \sum^*$.
Tak się da, bo gramatyki z redukcji PCP $\leq_{rek}$ P114 są deterministyczne.
Dopełnienie deterministycznego CFL jest deterministycznym CFL.
Suma CFL jest CFL.

116

Niech P116 – problem z zadania, tj. “Dla CFG G, H czy $L_G = L_H$”?

T: P115 $\leq_{rek}$ P116

Dają nam $\sum$ i G. Chcemy rozstrzygnąć, czy $L_G = \sum^*$.
Tworzymy H t. że $L_H = \sum^*$, sprawdzamy, czy $L_G = L_H$.

120

T: K $\leq_{rek}$ P120

Dają nam TM, chcemy rozstrzygnąć, czy się zatrzyma.

Niech czerwony – kolor specjalny, biały – blank, reszta kolorów – zbiór $\sum \cup Q \times \sum$.
Zamiast nieestetycznych chyba lepiej definiować estetyczne ułożenia, tj. dozwolone.

Estetyczne ułożenia kafelków:

Dla każdych q, a jeśli δ(q, a) = (q', a', L):

x	(q, a)
(q', x)	a'

Dla każdych q, a jeśli δ(q, a) = (q', a', R):

(q, a)	x
a'	(q', x)

Dodatkowo, żeby móc zacząć od czerwonego kafelka i "wrócić" z qF do czerwonego kafelka:

(q0, B)	

oraz

(qF, B)	

(Zakładamy, że mamy TM, która potrafi skończyć w qF z pustą taśmą).

Być może przyda się też coś takiego, żeby kwadrat mógł być kwadratowy, ale to raczej detal:

Jeżeli istnieje kwadrat o skończonym boku, który da się tak pokryć, to TM się zatrzymuje.

121

T1: kcomp może być dowolnie duże.
(Dla każdego k istnieje n takie że kcomp(n) > k).

T2: kcomp nie jest rekurencyjna.

Korzystamy z T1. Załóżmy nie wprost, że kcomp jest rekurencyjna. Niech LI(f) – liczba znaków programu (obliczającego) f.

Napiszemy program ϕ:

- LTOT := LI(kcomp) + LI(ϕ)
- dla każdego n ∈ N:
    - jeśli kcomp(n) > LTOT:
        - wypisz n

ϕ() zwróci n, do którego wypisania potrzeba programu o większej niż LTOT liczbie znaków, jednocześnie samemu mając dokładnie LTOT znaków, co daje sprzeczność.

136

Dają nam instancję PCP(∑, w, v), gdzie w, v to listy słów.
Chcemy skonstruować Żuczka Kleksiorka, który akceptuje słowo będące odwzorowaniem rozwiązania PCP (lista indeksów + być może dodatkowe informacje).

Słowo do zaakceptowania:
efekt końcowy w # indeksy # efekt końcowy v

Idea:
Czytamy indeks i, wchodzimy do stanu, w którym wiemy, jakie jest w[i], przechodzimy do efektu końcowego w, sprawdzamy, czy początek niezakleksowanej części się zgadza, jak tak, to zakleksowujemy dalsze w[i] (jak coś pójdzie nie tak, to fail), przechodzimy do stanu, w którym znamy v[i] oraz efektu końcowego v, postępujemy analogicznie, jeśli wszystko wyszło ok to wracamy i zakleksowujemy obecny indeks. Jeśli rozwiązanie PCP jest OK, to cała taśma zostanie zakleksowana i możemy przejść do qF.

122

a. H10prim $\leq_{rek}$ H10
b. H10 $\leq_{rek}$ H10prim

Dają nam instancję H10prim. Każdą liczbę całkowitą da się przedstawić jako różnicę dwóch liczb naturalnych, więc:

a1x1^e1x2^e2…xk^ek + … = 0
a1(x1’ - x1’’)^e1(x2’ - x2’’)^e2…(xk’ - xk’’)^ek + … = 0

Dają nam instancję H10. Każdą liczbę naturalną da się przedstawić jako sumę kwadratów 4 liczb całkowitych (Lagrange), więc:

a1x1^e1x2^e2…xk^ek + … = 0
a1(x1a^2 + x1b^2 + x1c^2 + x1d^2)^e1… + … = 0

123

T: H10bis jest rozstrzygalny.

Mamy dwie maszyny Turinga M1 i M2, niech M1 sprawdza po kolei rozwiązania, a M2 próbuje pokazać, że nie ma rozwiązania – któraś się w końcu zatrzyma.
https://math.stackexchange.com/questions/181380/second-degree-diophantine-equations
https://www.wikiwand.com/en/Hasse_principle

139

T: Jeżeli A jest w NP, to istnieje NTM_A “zgadująca” rozwiązanie A

Jeżeli A jest w NP, to da się rozstrzygnąć poprawność rozwiązania w czasie wielomianowym. Niech TM_A – maszyna Turinga, która wczytuje n, m, gdzie n – instancja A, m – rozwiązanie, po czym sprawdza poprawność. m musi być wielomianowe, no bo w przeciwnym razie samo wczytywanie zajęłoby zbyt długo więc istnieje p, Działanie musi być wielomianowe, bo problem weryfikacji A jest w P, więc istnieje q, którym da się ograniczyć czas działania TM_A.

Działanie NTM_A:
- niedeterministycznie “zgadnij” m
- wczytaj n
- odpal TM_A dla (n, m)

140

T: A’ jest w NP (A’ – rzut A z zadania)

Niech TM_A – maszyna Turinga rozstrzygająca A w PTIME.
Korzystamy z 139 – skonstruujemy NTM rozpoznającą A’ w PTIME.

Działanie:
- wczytaj n
- niedeterministycznie “zgadnij” $m \leq p(|n|)$
- odpal TM_A dla (n, m)

141

T: Dla danego B w NP istnieje A, które jest w P
Niech NTM_B – niedeterministyczna maszyna Turinga rozstrzygająca B w PTIME.

$A = \{ (n, m) \mid n \in B \wedge m$ – przebieg NTM_B rozstrzygający n $\}$

Skonstruujemy maszynę Turinga rozstrzygającą A w PTIME.

Działanie:
- wczytaj n, m
- zasymuluj przebieg NTM_B dla n

142

T: 5SAT $\leq_P$ 3SAT

Weźmy instancję problemu 5SAT – formułę w CNF ϕ. Jest ona koniunkcją klauzul postaci:

a1 ∨ a2 ∨ a3 ∨ a4 ∨ a5

Każdą taką klauzulę jesteśmy w stanie zastąpić 3 klauzulami postaci:

(a1 ∨ a2 ∨ z1) ∧ (~z1 ∨ a3 ∨ z2) ∧ (~z2 ∨ a4 ∨ a5)

Więc dla instancji 5SAT ϕ o n klauzulach tworzymy instancję 3SAT o 3n klauzulach przy pomocy 2n nowych zmiennych.

143

T: 3SAT $\leq_P$ STASI

https://www.nitt.edu/home/academics/departments/cse/faculty/kvi/NPC%20DOMINATING%20SET.pdf

144

T1: H $\leq_P$ Hd

Mamy instancję H G, budujemy instancję Hd G', gdzie zastępujemy każdą krawędź G dwoma krawędziami skierowanymi.

T2: Hd $\leq_P$ H

Mamy instancję Hd G, konstruujemy instancję H G':

- Dla każdego v ∈ V_G dajemy do V_G' v_in, v_mid, v_out.
- Dla każdej (v, v') ∈ E_G dajemy do E_G' {v_out, v'_in}.

Dowód, że to redukcja – w 1 stronę łatwe, w drugą:
G' ma cykl Hamiltona ⇒ G ma skierowany cykl Hamiltona.
Załóżmy, że G' ma cykl Hamiltona, weźmy ten cykl. Każdy v_mid musi się pojawić w cyklu, a jako że ma on stopień 2, musi być pomiędzy wierzchołkami v_in i v_out. Z kolei v_in ma krawędzie inne niż {v_mid, v_in} tylko do wierzchołków typu _out (i odwrotnie dla v_out), więc znaleziony cykl musi wyglądać na jeden ze sposobów:

1. v1_in – v1_mid – v1_out – v2_in – … – vn_mid – vn_out
2. v1_out – v1_mid – v1_in – v2_out – … – vn_mid – vn_in

Załóżmy 1., odpowiada to cyklowi w grafie skierowanym G v1 → v2 → … → vn.

145

T: HORNSAT ∈ P

https://www.wikiwand.com/en/Horn-satisfiability

146

T: 2SAT ∈ P

https://www.wikiwand.com/en/2-satisfiability

147

T: 3SAT $\leq_P$ 3SAT_3

Mamy instancję 3SAT ϕ, skonstruujemy instancję 3SAT_3.
Dla każdej zmiennej x, która pojawia się więcej niż 3 razy:

- zastępujemy kolejne wystąpienia nowymi zmiennymi x1, x2, …, xk;
- dodajemy klauzule (~x1 ∨ x2), (~x2 ∨ x3), …, (~xk ∨ x1), co tworzy “krąg implikacji”: x1 ⇒ x2 ⇒ … ⇒ xk ⇒ x1, więc wszystkie muszą być 0 albo 1.

153

T: SAT_2 ∈ P

https://cs.stackexchange.com/questions/86730/show-that-the-sat-problem-for-cnf-formulas-with-at-most-two-occurences-of-each-v

148

T: 3SAT $\leq_P$ HAMCYCLE

http://eaton.math.rpi.edu/faculty/Mitchell/courses/matp6620/notesMATP6620/lecture06/06A_hamiltoniancycle.pdf

149

T: HAMCYCLE $\leq_P$ TSP

Dają nam instancje HAMCYCLE G. Tworzymy instancję TSP – graf pełny o wierzchołkach z G i wagach krawędzi:

w(e) = {
    1 : e ∈ G.E,
    2 : wpp
}

ze stałą k = |V_G|.

150

Pokażemy, że przy pomocy takiego algorytmu dałoby się rozwiązać HAMCYCLE.

Dla danej instancji HAMCYCLE G tworzymy instancję TSP, gdzie:

w(e) = {
    1 : e ∈ E,
    2|V_G| : wpp
}

Jeśli algorytm zwróci cykl długości |V_G|, to G ma cykl Hamiltona.

151

Wielomianowy algorytm aproksymacyjny:

1. Budujemy MST.
2. Przechodzimy wierzchołki MST DFS-em.
3. Łączymy powtarzające się krawędzie
   (tj. dla 1→ 2 → 3 → 2 → 4 → 2 → 5 mamy 1 → 2 → 3 → 4 → 5).

Dowód optymalności:

1. Rozwiązanie TSP bez jednej krawędzi to drzewo spinające, więc MST nie jest gorsze.
2. DFS przechodzi po każdej krawędzi 2 razy, więc mamy 2x gorsze rozwiązanie.
3. Z nierówności trójkąta tutaj nadal mamy 2x gorsze rozwiązanie.

152

Jak w 149, tylko z wagą 1.5 dla e ∉ E.

154

T: 3SAT $\leq_P$ KLIKA_2

1. 3SAT $\leq_P$ KLIKA_3 (Sipser)
2. KLIKA_3 $\leq_P$ KLIKA_2

Problem znalezienia kliki o |V|/3 wierzchołkach jest NP-zupełny. Dla danej instancji dodajemy zbiór X kolejnych |V|/3 wierzchołków połączonych ze wszystkimi innymi. Jeśli znajdziemy klikę o |V|/2 wierzchołkach, to po wyrzuceniu X z rozwiązania zostaje nam klika o |V|/3 wierzchołkach.

158

T: 3COL $\leq_P$ 3COL+1

Niech kolory – {R, G, B}. Dają nam instancję 3COL G. Konstruujemy G', gdzie do każdego wierzchołka z G “doklejamy” klikę K5.

$L \Rightarrow R$

Mamy 3-kolorowanie G. Zachowujemy wszystko jak w oryginale i dodajemy kolorowanie K5 dla każdego wierzchołka – dla wierzchołka o kolorze R musi być 1x R, 2x G, 2x B.

$R \Rightarrow L$

Mamy 3+1-kolorowanie G'. Każdy wierzchołek z "doklejoną" kliką K5 to razem klika K6, więc musi w niej wystąpić 2x każdy kolor. Co za tym idzie, "oryginalne" wierzchołki muszą być pokolorowane poprawnym "zwykłym" 3-kolorowaniem, bo "doklejone" kliki wyczerpują "tolerancję" max 1 wierzchołka w tym samym kolorze.

159

T: $3SAT\_3 \leq_P$ PZPR

Dają nam instancję 3SAT_3 ϕ. Dla każdej zmiennej p z ϕ usuwamy klauzule z ($p \vee \sim p \vee \ldots$), po czym dodajemy do A zbiory:

{p}, {p, C1}, {p, C2}, …, {p, C1, C2, C3}

gdzie C1, C2, C3 – klauzule, w których występuje p w takiej samej postaci,
tj. dla C1 = ($p \vee \ldots$), C2 = ($p \vee \ldots$), C3 = ($\sim p \vee \ldots$):

- {p}, {p, C1}, {p, C2}, {p, C1, C2}
- {p, C3}

$L \Rightarrow R$

Mamy wartościowanie, które spełnia ϕ. Dla każdej klauzuli Ci, która jest spełniona dzięki wartościowaniu p (T, jeśli Ci zawiera p; F, jeśli ~p), dajemy do rozwiązania {p, C1, …}. Jeśli dana klauzula jest spełniona jednocześnie dzięki wartościowaniu kilku zmiennych, wybieramy jedną z nich, np. jeśli Cx = ($p \vee q \vee r$) i p, q, r = T, to możemy wybrać spośród:

- {p, Cx}, {q}, {r}
- {p}, {q , Cx}, {r}
- {p}, {q}, {r, Cx}

W wyniku mamy B, w którym zbiory są rozłączne, każda zmienna występuje raz, każda klauzula występuje raz.

$R \Rightarrow L$

Mamy rodzinę zbiorów rozłącznych B, taką że suma B = A. Dla każdego $A_i \in B$ patrzymy, jak wygląda ten zbiór – jeśli jest w nim tylko zmienna p, to jej wartościowanie nie ma znaczenia. Jeśli są w nim jeszcze klauzule, to patrzymy, w jakiej postaci występuje w nich p. Jeśli p, to p = T, jeśli ~p, to p = F.

173

T: NAE-3-SAT $\leq_P$ 173

Redukcja: niech kolorom w grafie odpowiada wartościowanie
Dla każdej zmiennej x mamy wierzchołki x i ~x połączone binarnym wierzchołkiem xalt (żeby nie pokolorować tak samo x i ~x). Dla każdej klauzuli C z 3 literałami dajemy krawędzie do tych literałów i chyba jest ok.

NAE-3-SAT ma rozwiązanie $\Rightarrow$ 173 ma rozwiązanie

Każdej klauzuli dajmy czarny, każdemu xalt biały, wtedy literały mogą być pokolorowane dowolnie, bo każdy z nich będzie miał sąsiadów w obu kolorach, więc ograniczenia są tylko z klauzul i odwrotności – które da się spełnić, jeśli formuła jest spełnialna.

NAE-3-SAT nie ma rozwiązania $\Rightarrow$ 173 nie ma rozwiązania

Dla każdego wartościowania istnieje klauzula, której wyjdzie TTT lub FFF, więc nie da się tak pokolorować grafu, żeby wierzchołki klauzulowe miały sąsiadów w obu kolorach.

180

T: 3COL $\leq_P$ NAE-3-SAT

Weźmy instancję problemu 3COL G.
Niech K = {r, g, b}, k(v) – $v \in G.V$ jest koloru $k \in K$.
Skonstruujemy r(G) – formułę logiczną w 3-CNF.

Wprowadzamy specjalną zmienną t, która dla poprawnego kolorowania ma być T.
Kodujemy kilka rzeczy:

1. Sąsiadujące wierzchołki nie mogą mieć tego samego koloru – dla każdych $\{v, v'\} \in E\_G$:
   $(r(v) \vee r(v') \vee t), (g(v) \vee g(v') \vee t), (b(v) \vee b(v') \vee t)$
2. Jeden wierzchołek może mieć tylko jeden kolor – dla każdego $v \in G.V$:
   $r(v) \vee g(v) \vee b(v)$

3. No i prawie dobrze, tylko teraz możemy mieć naraz 2 kolory. Potrzebujemy:
   (r(v) ∨ g(v) ∨ t), (g(v) ∨ b(v) ∨ t), (r(v) ∨ b(v) ∨ t)

3COL G ma rozwiązanie ⟹ NAE-3-SAT r(G) ma rozwiązanie

Z konstrukcji, t = T, a reszta jak w G.

3COL G nie ma rozwiązania ⟹ NAE-3-SAT r(G) nie ma rozwiązania

Jeśli G nie ma 3-kolorowania, to 2 sąsiednie wierzchołki muszą mieć ten sam kolor.
Wtedy istnieją takie {v, v'} ∈ G.E, że r(v) ∧ r(v') (albo g, albo b).
Wtedy żeby r(v) ∨ r(v') ∨ t była spełniona, to t = F.
Ale wtedy (g(v) ∨ g(v') ∨ t), (b(v) ∨ b(v') ∨ t) nie są spełnione.
Więc r(G) nie jest spełnialna.

181

To prawie jest NAE-3-SAT (wierzchołki – literały, kolory – prawda/fałsz, krawędzie – tam gdzie klauzule). Tylko trzeba jeszcze ograniczyć jakoś to, żeby wierzchołki x i ~x nie dostały takiego samego koloru.

183

Zakładamy, że |x| – dł. zapisu binarnego x.
Niech A = { (x, y) | $f^{-1}(x) < y$ }.

Lemat: f jest bijekcją.
- Z treści jest różnowartościowa.
- Z |n| = |f(n)| dla danego n ∈ [2^k, 2^k+1) musi przybrać każdą wartość z tego przedziału.

Zał. f istnieje, wtedy pokażemy, że NP ∩ co-NP ≠ PTIME.

1. A ∈ NP – dla danych (x, y) możemy niedeterministycznie zgadnąć n takie że f(n) = x, zwracamy n < y.
2. A ∈ co-NP ⇔ A' ∈ NP —||—, zwracamy n ≥ y.
3. A ∉ PTIME – załóżmy, że A ∈ PTIME. Wtedy da się obliczyć f^{-1} w PTIME.
   Program obliczający f^{-1}:
   - wczytaj x // niech |x| = k ⟹ x ∈ [2^k, 2^k+1)
   - binarnie przeszukaj i = [k, k + 1) // O(log(2^k)) – O(|x|)
     - jeśli A(i, x), to lewo, wpp prawo czy jakoś tak

192

(Sipser)

Konwertujemy wyrażenie do NFA, odpalamy niedeterministyczną maszynę Turinga symulującą NFA, która spróbuje zgadnąć, czy jakieś słowo NIE należy do języka:

- wczytaj RE
- skonstruuj NFA M ze zbiorem stanów Q
- dla i od 1 do 2^|Q|:
    - niedeterministycznie zgadnij literę a $\in \sum$
    - wrzuć a do M
    - jeśli a jest w stanie nieakceptującym:
        - zwróć 1 i zakończ
- zwróć 0 i zakończ

193

Sprawdzamy wszystkie wartościowania w 3-arnym drzewie, z którego pamiętamy tylko obecną ścieżkę. Chcemy ustalić, że wartościowanie jest dokładnie jedno, więc kasujemy podwójne.

202

Korzystamy z tożsamości: $(x \Rightarrow z) \land (y \Rightarrow z)$ wtw $(x \lor y) \Rightarrow z$.

P_2(x, y) = $\exists z$ R(x, z) $\land$ R(z, y)
P_k(x, y) = $\exists z$ P_k/2(x, z) $\land$ P_k/2(z, y)
= $\exists z \forall a \forall b$ ((a = x $\land$ b = z) $\Rightarrow$ P_k/2(a, b)) $\land$ ((a = z $\land$ b = y) $\Rightarrow$ P_k/2(a, b))
= $\exists z \forall a \forall b$ ((a = x $\land$ b = z) $\lor$ (a = z $\land$ b = y)) $\Rightarrow$ P_k/2(a, b)