Modele Obliczeń

Zagadnienia egzaminacyjne

„Nie wiemy czy P zawiera się w PSPACE."

Popełnione przez

Dziurawy Ponton
Załatany Ponton
Puchaty Pompon

Kraków
Anno Domini 2023

Spis treści

Licencja

Podstawy

0.1 Alfabet

$\Sigma = \{0, 1\}$ – dla ustalenia uwagi, w ogólności może to być dowolny niepusty skończony zbiór

Σ^i – słowa długości *dokładnie* i

$\Sigma^0 = \{\varepsilon\}$ – język jednostkowy (przez ε oznaczamy słowo puste)

$\Sigma^* = \bigcup_{i=0}^{\infty} \Sigma^i$ – wszystkie słowa

$\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$ – wszystkie niepuste słowa

0.2 Słowa

Nie wiemy co to są słowa.

0.3 Język

Definicja 0.3.1 (Język)**. Językiem** nad alfabetem Σ nazywamy dowolny podzbiór Σ^*

Pytanie teoretyczno-abstrakcyjne – czy każdy język ma skończony opis? Otóż nie bardzo, bo programów (opisów) mamy przeliczalnie wiele ($\aleph_0$, bo są to *elementy* Σ^*), a możliwych języków mamy nieprzeliczalnie wiele ($\mathfrak{c}$, bo są to *podzbiory* Σ^*).

Ok, to teraz interesuje nas jakie języki są opisywalne – tj. dla jakich języków istnieje program Π który dla ustalonego języka L oraz słowa z sprawdza, czy $z \in L$.

0.4 Problemy obliczeniowe

Ogólnie mówiąc **problem decyzyjny** to problem pytający się czy jakiś obiekt spełnia określony warunek, tj. zwraca odpowiedź TAK/NIE.

Jak się to ma do języków? Otóż nasze obiekty musimy umieć zakodować binarnie – np. liczby możemy zapisać ich reprezentacją binarną, grafy macierzą incydencji (zapisujemy jej kolejne wiersze), itp.

W takim razie języki to problemy obliczeniowe a pytanie „Czy dany obiekt spełnia warunek?" zamieniamy na „Czy słowo opisujące ten obiekt należy do języka, w którym słowa to kody wszystkich poprawnych instancji?".

0.5 Opis języka

Języki możemy scharakteryzować pod względem złożoności.

Dostajemy coś, co się nazywa **hierarchią Chomsky'ego**

1. języki regularne
   (a) automat skończony
   (b) gramatyka liniowa
   (c) wyrażenie regularne
2. języki bezkontekstowe
   (a) automat ze stosem
   (b) gramatyka bezkontekstowa
3. języki kontekstowe
   (a) gramatyka kontekstowa
4. języki rekurencyjnie przeliczalne
   (a) maszyna Turinga

0.6 Operacje

0.6.1 Na słowach

Słowa możemy traktować jako funkcję $n \mapsto \Sigma$ (gdzie $n \in \mathbb{N}$ – tzn. możemy zapytać taką funkcję, jaka literka jest pod danym indeksem.

Definicja 0.6.1. Konkatenacją nazywamy funkcję

$$\cdot\colon \Sigma^* \times \Sigma^* \mapsto \Sigma^*$$

zdefiniowaną następująco:

$$(v \cdot w)(i) = \begin{cases} v(i) & \text{gdy } i \leq |v| \\ w(i - |v|) & \text{gdy } |v| < i \leq |v| + |w| \end{cases}$$

Mówiąc po ludzku, jest to po operacja sklejenia dwóch słów w jedno. Zazwyczaj będziemy pomijać kropkę i zapisywać po prostu vw.

Własności konkatenacji:

- łączność – $(ab)c = a(bc)$
- element neutralny – $a\varepsilon = \varepsilon a = a$
- $|ab| = |a| + |b|$

Zatem $(\Sigma^*, \cdot, \varepsilon)$ to *monoid*.

Definicja 0.6.2 (Potęga słowa)**.**

$$w^n = \begin{cases} 0 & \text{gdy } n = 0 \\ w^{n-1}w & \text{gdy } n > 0 \end{cases}$$

0.6.2 Na językach:

- suma $L_1 \cup L_2$
- przecięcie $L_1 \cap L_2$
- katenacja
$$L_1 \cdot L_2 = \{w_1 w_2 : w_1 \in L_1 \wedge w_2 \in L_2\}$$
Widzimy, że dla dowolnego $L \subseteq \Sigma^*$:
$$L\varnothing = \varnothing L = \varnothing$$
$$L\{\varepsilon\} = \{\varepsilon\}L = L$$
- potęga
$$L^n = \begin{cases} \{\varepsilon\} & \text{gdy } n = 0 \\ L^{n-1} \cdot L & \text{gdy } n > 0 \end{cases}$$
- gwiazdka Kleenego
$$L^* = \bigcup_{n=0}^{\infty} L^n$$

Dla rozróżnienia sklejanie słów będziemy nazywać *konkatenacją* a sklejanie języków *katenacją*.

0.7 Algebra Kleenego

Napis $a \leq b$ jest skrótem dla $a + b = b$.

Definicja 0.7.1 (Algebra Kleenego)**. Algebrą Kleenego** nazywamy tuplę $\mathbf{A} = (A, +, \cdot, ^*, 0, 1)$, która spełnia następujące warunki:

1. $(a + b) + c = a + (b + c)$
2. $a + b = b + a$
3. $a + a = a$
4. $a + 0 = a$
5. $(a \cdot b) \cdot c = a \cdot (b \cdot c)$
6. $a \cdot 1 = 1 \cdot a = a$
7. $a \cdot 0 = 0 \cdot a = 0$
8. $a \cdot (b + c) = a \cdot b + a \cdot c$
9. $(a + b)c = ac + bc$
10. $1 + aa^* \leq a^*$
11. $1 + a^*a \leq a^*$
12. $ax + b \leq x \implies a^*b \leq x$
13. $xa + b \leq x \implies ba^* \leq x$

Tak się składa, że

$$(\mathcal{P}(\Sigma^*), \cup, \cdot, ^*, \varnothing, \{\varepsilon\})$$

jest algebrą Kleenego.

Lemat 0.7.1 (Lemat Ardena)**.** Niech $A, B, X \subseteq \Sigma^*$ i załóżmy, że

$$A \cdot X \cup B = X$$

Wtedy język $X = A^*B$ jest rozwiązaniem tego równania, przy czym

(i) jest rozwiązaniem najmniejszym w sensie inkluzji

(ii) jeśli $\varepsilon \notin A$, to jest jedynym rozwiązaniem

Dowód. Pokażmy najpierw, że A^*B spełnia równanie.

$$A(A^*B) \cup B = (AA^*)B \cup B = (AA^* \cup \{\varepsilon\})B = A^*B$$

Przyjrzyjmy się teraz dodatkowym wnioskom.

(i) Weźmy dowolne rozwiązanie X.

Najpierw pokażemy indukcyjnie że dla każdego k mamy $A^kB \subseteq X$.

1. Baza indukcji – przyjmujemy $k = 0$:

$$A^0B = \{\varepsilon\}B = B \subseteq X$$

Ostatnie zawieranie wynika bezpośrednio z równania: $X = AX \cup B$.

2. Krok indukcyjny – zakładamy, że $A^kB \subseteq X$ i dowodzimy, że również $A^{k+1}B \subseteq X$. Wiemy, że $A^{k+1}B = AA^kB$. Ponieważ $A^kB \subseteq X$, to $AA^kB \subseteq AX \subseteq X$, gdzie ostatnią inkluzję ponownie bierzemy bezpośrednio z równania.

Co nam to daje? To, że

$$\bigcup_{k \in \mathbb{N}} A^kB \subseteq X$$

bo skoro każdy z sumowanych zbiorów zawiera się w X, to ich suma też. Zwróćmy teraz uwagę, że

$$\bigcup_{k \in \mathbb{N}} A^kB = A^*B$$

i mamy $A^*B \subseteq X$, tak, jak chcieliśmy.

(ii) Załóżmy, że $\varepsilon \notin A$ i pokażmy, że jeśli X jest rozwiązaniem to $X \subseteq A^*B$. Z tego otrzymamy już równość, bo inkluzję w drugą stronę udowodniliśmy w poprzednim podpunkcie.

Prowadzimy dowód nie wprost: załóżmy, że istnieją słowa należące do X, ale nie do A^*B. Niech w będzie najkrótszym takim słowem (w przypadku remisu dowolnym z najkrótszych).

Zastanówmy się najpierw: czy może być $w \in B$? Otóż nie, ponieważ $\varepsilon \in A^*$, więc gdyby $w \in B$, to $w = \varepsilon w \in A^*B$, a przecież przed chwilą założyliśmy, że tak nie jest.

Zatem $w \notin B$. Przypomnijmy, że $w \in X = AX \cup B$, bo X jest rozwiązaniem naszego równania. Zatem skoro $w \notin B$, to siłą rzeczy $w \in AX$, bo ma należeć do sumy. Oznacza to, że $w = uv$ dla pewnych $u \in A$, $v \in X$.

Niewątpliwie $u \neq \varepsilon$, bo $\varepsilon \notin A$, na mocy założeń twierdzenia. Zatem długość v jest *silnie mniejsza* od długości w.

Na koniec zastanówmy się, czy może zachodzić $v \in A^*B$. Okazuje się, że nie może. Dlaczego? Bo gdyby tak było, to

$$w = uv \in AA^*B$$

ale

$$AA^*B = A\left(\bigcup_{k=0}^{\infty} A^k\right)B = \left(\bigcup_{k=1}^{\infty} A^k\right)B = \bigcup_{k=1}^{\infty}(A^kB) \subseteq \bigcup_{k=0}^{\infty}(A^kB) = \left(\bigcup_{k=0}^{\infty} A^k\right)B = A^*B$$

Czyli mielibyśmy $w \in A^*B$, a przecież założenie, że tak nie jest, już śni nam się po nocach.

Znaleźliśmy zatem śmieszne słowo v, które należy do X, *nie* należy do A^*B, i jest krótsze od w. Ale chwila, przecież w miało być najkrótszym należącym do X i nienależącym do A^*B! Uzyskana sprzeczność dowodzi tezy.

□

Lemat 0.7.2. $\leq$ jest porządkiem częściowym

Dowód. Porządek częściowy musi spełniać trzy warunki:

1. zwrotność, czyli $a \leq a$ – z warunku (3) dla algebry Kleenego mamy dokładnie $a + a = a$.
2. przechodniość, czyli jeśli $a \leq b$ oraz $b \leq c$ to $c \leq a$

$$a + c = a + (b + c) = (a + b) + c = b + c = c$$

3. antysymetryczność, czyli jeśli $a \leq b$ oraz $b \leq a$ to $a = b$

$$b = a + b = b + a = a$$

□

Rozdział 1

Języki regularne

1.1 Wyrażenia regularne

Definicja 1.1.1. Język REG_Σ definiujemy jako najmniejszy język nad alfabetem $\Sigma' = \Sigma \cup \{+, \cdot, ^*, 0, 1, (,)\}$, który spełnia następujące warunki:

- $0, 1 \in \mathrm{REG}_\Sigma$
- $\forall_{a \in \Sigma} : a \in \mathrm{REG}_\Sigma$
- $\forall_{d_1, d_2 \in \mathrm{REG}_\Sigma} : (d_1 + d_2) \in \mathrm{REG}_\Sigma$
- $\forall_{d_1, d_2 \in \mathrm{REG}_\Sigma} : (d_1 \cdot d_2) \in \mathrm{REG}_\Sigma$
- $\forall_{d \in \mathrm{REG}_\Sigma} : d^* \in \mathrm{REG}_\Sigma$

Definicja 1.1.2. Wyrażenie regularne nad alfabetem Σ definiujemy jako element języka REG_Σ.

Warto zaznaczyć, że operujemy jedynie na napisach bez żadnej intepretacji – symbole są jedynie symbolami, które będziemy zaraz interpretować odpowiednio.

Możemy teraz sobie dobrać do tego interpretację $L : \mathrm{REG}_\Sigma \to \mathcal{P}(\Sigma^*)$

- $L(0) = \varnothing$
- $L(1) = \{\varepsilon\}$
- $L(a) = \{a\}$
- $L(\alpha + \beta) = L(\alpha) \cup L(\beta)$
- $L(\alpha \cdot \beta) = L(\alpha) L(\beta)$
- $L(\alpha^*) = L(\alpha)^*$

Stosując tę dosyć naturalną interpretację będziemy opisywać sobie języki.

1.2 Automaty

1.2.1 DFA

Definicja 1.2.1. Deterministyczny Automat Skończony (DFA od deterministic finite automaton) to tupla:

$$A = (Q, \Sigma, \delta, s, F)$$

gdzie

- Q jest skończonym zbiorem stanów
- Σ jest skończonym alfabetem
- $\delta : Q \times \Sigma \to Q$ jest funkcją przejścia
- $s \in Q$ jest stanem startowym
- $F \subseteq Q$ jest zbiorem stanów akceptujących (końcowych)

To jest bardzo abstrakcyjna i formalna definicja, w praktyce automat będziemy reprezentować jako graf skierowany, albo jako tabelkę przejść.

O automacie możemy myśleć jako o maszynie, która rozpoznaje czy zadane słowo należy do jakiegoś języka (związanego z tymże automatem). Zaczynamy w stanie startowym i przechodzimy do kolejnego stanu zjadając przy tym kolejne litery ze słowa.

Aby nieco sformalizować powyższe zdanie wprowadzamy funkcję $\hat{\delta}$, która definiuje w jakim stanie kończymy jeśli zaczynamy w stanie q z danym słowem w:

Definicja 1.2.2. $\hat{\delta} : Q \times \Sigma^* \to Q$

$$\hat{\delta}(q, w) = \begin{cases} q & \text{jeśli } w = \varepsilon \\ \delta(\hat{\delta}(q, x), a) & \text{jeśli } w = xa, a \in \Sigma \end{cases}$$

Fakt, że funkcja $\hat{\delta}$ faktycznie zgadza się z oczekiwaną semantyką można prosto udowodnić stosując indukcję po długości słowa.

Definicja 1.2.3. Językiem akceptowanym przez automat $A = (Q, \Sigma, \delta, s, F)$ nazywamy

$$L(A) = \left\{ w \in \Sigma^* \mid \hat{\delta}(s, w) \in F \right\}$$

Podobnie, słowo w jest akceptowane przez automat A jeśli $w \in L(A)$.

1.2.2 NFA

Deterministyczne automaty są dość proste – jak widzą jakąś literę to po prostu za nią idą.

Ale co gdyby nasz automat miał kilka opcji do wyboru i mógł „zgadywać" gdzie powinien przejść dalej? Wtedy dostajemy automat, który przestaje być deterministyczny, ale nadal można go sensownie zdefiniować.

Definicja 1.2.4. Niedeterministyczny Automat Skończony (NFA – nondeterministic finite automaton) to tupla:

$$A = (Q, \Sigma, \delta, S, F)$$

gdzie

- Q jest skończonym zbiorem stanów
- Σ jest skończonym alfabetem
- $\delta \subseteq Q \times \Sigma \times Q$ jest relacją przejścia
- $S \subseteq Q$ jest zbiorem stanów startowch
- $F \subseteq Q$ jest zbiorem stanów akceptujących (końcowych)

To co się zmieniło względem DFA to:

- δ jest dowolną relacją (czyli niekoniecznie funkcją), co oddaje fakt, że możemy przechodzić do różnych stanów na podstawie tej samej litery
- S jest zbiorem stanów startowych (czyli może być ich więcej niż jeden)

Zdefiniujmy pomocniczą funkcję $\tilde{\delta}$, która mówi nam dokąd możemy się rozgałęzić jeśli jesteśmy w stanach β i widzimy literę a. Jest to w pewnym sensie stworzenie takiej funkcji jaką jest δ w DFA.

Definicja 1.2.5. $\tilde{\delta} : \mathcal{P}(Q) \times \Sigma \to \mathcal{P}(Q)$

$$\tilde{\delta}(\beta, a) = \{q \in Q \mid \exists_{q_b \in \beta}(q_b, a, q) \in \delta\}$$

Definiujemy teraz, $\hat{\delta}$ które (trzymając się pewnej analogii z DFA) mówi, gdzie możemy skończyć jeśli zaczynamy w jakimś zbiorze stanów β oraz „przechodząc" słowo w.

Definicja 1.2.6. $\hat{\delta} : \mathcal{P}(Q) \times \Sigma^* \to \mathcal{P}(Q)$

$$\hat{\delta}(\beta, w) = \begin{cases} \beta & \text{jeśli } w = \varepsilon \\ \tilde{\delta}(\hat{\delta}(\beta, x), a) & \text{jeśli } w = xa, a \in \Sigma \end{cases}$$

Ponownie, zgodność definicji z semantyką można wykazać szybką indukcją.

Definiujemy również język akceptowany przez NFA, analogicznie do tego co widzieliśmy w DFA.

Definicja 1.2.7.

$$L(A) = \left\{w \in \Sigma^* \mid \hat{\delta}(S, w) \cap F \neq \varnothing\right\}$$

1.2.3 ε-NFA

Definicja 1.2.8. Definiujemy ε-NFA jako NFA, z tym że:

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times Q$$

Bardziej intuicyjnie – pozwalamy na przechodzenie krawędzią bez konsumowania obecnej litery – definicja funkcji $\tilde{\delta}$ wygląda bardzo podobnie do tej ze „zwykłego" NFA.

Definicja 1.2.9. Funkcję $\tilde{\delta} : \mathcal{P}(Q) \times (\Sigma \cup \varepsilon) \to \mathcal{P}(Q)$ definiujemy następująco:

$$\tilde{\delta}(B, a) = \{q \in Q : \exists_{q_b \in B}(q_b, a, q) \in \delta\}$$

Niestety sprawia to, że przy tej definicji ε-NFA nie jest jako tako NFA, który zawsze musi użyć obecnej litery. To wymusza dodatkowe shenanigansy w definicji, z którymi radizmy sobie wprowadzając pojęcie **epsilon-domknięć**.

Definicja 1.2.10. Definiujemy **ε-domknięcie** zbioru stanów B w automacie A jako zbiór $B^{A,\varepsilon}$ taki, że

$$B^{A,\varepsilon} = \bigcup_{i=0}^{\infty} B_i^{A,\varepsilon}$$

gdzie

$$B_i^{A,\varepsilon} = \begin{cases} B & \text{gdy } i = 0 \\ \tilde{\delta}\left(B_{i-1}^{A,\varepsilon}, \varepsilon\right) & \text{gdy } i > 0 \end{cases}$$

Intuicyjnie, $B^{A,\varepsilon}$ jest to zbiór wszystkich stanów do których jesteśmy w stanie się „dostać" za pomocą samych epsilon przejść z jakiegoś innego zbioru stanów.

Definicja 1.2.11. Niech $\mathcal{P}^{A,\varepsilon}(Q)$ to zbiór wszystkich ε-domkniętych zbiorów stanów, to znaczy:

$$\mathcal{P}^{A,\varepsilon}(Q) = \left\{B^{A,\varepsilon} : B \in \mathcal{P}(Q)\right\}$$

Definicja 1.2.12. Definiujemy funkcję $\Delta : \mathcal{P}^{A,\varepsilon}(Q) \times \Sigma \to \mathcal{P}^{A,\varepsilon}(Q)$ następująco:

$$\Delta(B, a) = \tilde{\delta}(B, a)^{A,\varepsilon}$$

1.3 Równoważność automatów i wyrażeń regularnych

1.3.1 Równoważność DFA i NFA

Można się zastanawiać, czy NFA jest silniejszy niż DFA (tzn. czy może akceptować więcej języków). Okazuje się, że nie. Co więcej, jesteśmy w stanie przekształcić dowolny NFA w DFA.

Twierdzenie 1.3.1. Niech $L \subseteq \Sigma^*$. Następujące warunki są równoważne:

1. Istnieje DFA A_D, dla którego $L(A_D) = L$
2. Istnieje NFA A_N, dla którego $L(A_N) = L$

Dowód. „$\Longrightarrow$"

Każdy DFA jest NFA, bo każda funkcja jest relacją.

„$\Longleftarrow$"

Niech A_N będzie NFA takim, że:

$$A_N = (Q, \Sigma, \delta, S, F)$$

Korzystamy z faktu, że $\tilde{\delta}$ jest elegancką funkcją i konstruujemy DFA A_D takie, że:

$$A_D = (\mathcal{P}(Q), \Sigma, \tilde{\delta}, S, \mathcal{F})$$

gdzie $\mathcal{F} = \{\beta \in \mathcal{P}(Q) \mid \beta \cap F \neq \varnothing\}$

Intuicyjnie, tworzymy DFA, które w swoich stanach trzyma informację, w jakich stanach NFA mogłoby się znaleźć po przejściu tego samego prefiksu słowa. W związku z powyższym definicja $\mathcal{F}$ jest całkiem intuicyjna, bo chcemy zaakceptować wtedy i tylko wtedy, gdy któreś obliczenie NFA znalazło się w stanie akceptującym.

Dla pełności potrzebujemy pokazać, że $\hat{\delta}_{A_N}(S,u) \cap F \neq \varnothing \iff \hat{\delta}_{A_D}(S,u) \in \mathcal{F}$, ale to mamy tak naprawdę z definicji, bo $\hat{\delta}_{A_D} = \hat{\delta}_{A_N}$

□

1.3.2 Równoważność DFA i ε-NFA

Okazuje się, że możemy bardzo podobny dowód przeprowadzić dla DFA i ε-NFA.

Twierdzenie 1.3.2. Niech $L \subseteq \Sigma^*$. Następujące warunki są równoważne:

1. Istnieje DFA A_D, dla którego $L(A_D) = L$
2. Istnieje ε-NFA A_N, którego $L(A_N) = L$

Dowód. „$\implies$”

Każdy DFA jest ε-NFA, bo każda funkcja jest relacją.

„$\impliedby$”

Niech A_N będzie ε-NFA takim, że:

$$A_N = (Q, \Sigma, \delta, S, F)$$

Ponownie jak w poprzednim przypadku, chcemy w stanach DFA „trzymać” informacje o tym, w jakich stanach może znaleźć się $\varepsilon - NFA$. Jedyne co się zmienia to to, że mamy do czynienia z epsilonami. Jak o tym pomyślimy, to w sumie jednak nie jest to wielki problem, bo wystarczy chodzić po epsilon-domknięciach z użyciem funkcji Δ.

Konstruujemy więc A_D, które jest DFA takim, że:

$$A_D = (\mathcal{P}^{A_N,\varepsilon}(Q), \Sigma, \Delta, S, \mathcal{F})$$

gdzie $\mathcal{F} = \left\{\beta \in \mathcal{P}^{A_N,\varepsilon}(Q) \mid \beta \cap F \neq \varnothing\right\}$

□

1.3.3 DFA $\to$ Wyrażenie regularne

Pokażemy teraz, że dla dowolnego DFA A możemy skonstruować wyrażenie regularne α takie, że $L(A) = L(\alpha)$.

Czynimy obserwację, że jeśli $w \in L(A)$ to istnieje ścieżka $s \to q_f \in F$ idąca stanami

$$q_1, q_2, \ldots, q_{k+1}$$

taka że $s = q_1$ i $q_{k+1} \in F$.

Definiujemy $\alpha^k_{i,j}$ jako takie wyrażenie regularne, które reprezentuje wszystkie ścieżki prowadzące od q_i do q_j, w której stany pośrednie należą do $\{q_1, \ldots, q_k\}$ (stany numerujemy *arbitralnie* od 1, bo tak będzie wygodniej). Widzimy, że dla każdych i, j:

$$\alpha_{i,j}^0 = \{a \in \Sigma : \delta(q_i, a) = q_j\} \cup \beta_{i,j}$$

gdzie

$$\beta_{i,j} = \begin{cases} \varnothing & \text{gdy } i \neq j \\ \{\varepsilon\} & \end{cases}$$

Zauważmy teraz, że dla dowolnych $k, i, j \leq |Q|, k \geq 1$ jest tak, że:

$$\alpha_{ij}^k = \alpha_{ij}^{k-1} + \alpha_{ik}^{k-1}(\alpha_{kk}^{k-1})^* \alpha_{kj}^{k-1}$$

To może wydawać się być overwhelming, więc to rozbijmy:

1. Pierwszy składnik sumy (α_{ij}^{k-1} mówi nam o sytuacji, kiedy przechodzimy ze stanu q_i do stanu q_j nie przechodząc przez q_k ani stan o „wyższym" numerze w porządku.
2. Drugi składnik sumy mówi o sytuacji, kiedy przechodzimy przez q_k (bo teraz możemy). W takiej sytuacji nasza wybitna podróż dzieli się na 3 segmenty, których wyrażenia regularne są relatywnie proste i które wypada skonkatenować, by mieć opis całej podróży:
   (a) Przejście z q_i do q_k (po raz pierwszy w czasie podróży) – opisywane wyrażeniem α_{ik}^{k-1}
   (b) Chodzenie dowolną liczbę razy (być może 0) z q_k do q_k , po stanach po których nam wolno chodzić – można to łatwo opisać wyrażeniem $(\alpha_{kk}^{k-1})^*$ (gwiazdka oddaje to, że możemy chodzić wiele razy).
   (c) Przejście z q_k do q_j (po raz ostatni czas w podróży, tzn. po tym wyjściu z q_k już tam nie wrócimy) – opisywane wyrażeniem α_{kj}^{k-1}.

Okazuje się w takim razie, że dostaliśmy coś bardzo fajnego: mianowicie, pokazaliśmy że jak mamy DFA możemy sobie konstruować takie wyrażenia regularne dla kolejnych k. Jest to absolutnie wspaniałe na mocy naszej wcześniejszej obserwacji, tzn. tego, że jeśli $w \in L(A)$ to istnieje taka ścieżka w naszym DFA. W takim razie regex który opisuje tę ścieżkę ma postać:

$$\sum_{q_j \in F} \alpha_{ij}^k$$

gdzie k to liczba stanów, a q_i to stan początkowy. Jest to poprawnie zdefiniowane wyrażenie regularne, bo jego składowe są poprawnie zdefiniowane. Formalnie dowód tego, że wszystkie wyrażenia regularne tej postaci są poprawnie zdefiniowane można przeprowadzić pewnie jakąś indukcją; pozostawiamy to jako ćwiczenie dla czytelnika.

1.3.4 Wyrażenie regularne → ε-NFA

Ostatnia rzecz, którą chcemy uzyskać to sposób konwersji wyrażenia regularnego do ε-NFA. To dałoby nam już wspaniały rezultat, bo mielibyśmy, że wyrażenia regularne i wymienione wyżej automaty są wzajemnie równoważne jeśli chodzi o ich „moc". W tym celu możemy wykonać indukcję po długości wyrażenia regularnego, intuicyjnie konstruując nowy automat na podstawie „wcześniejszych".

Na początek wypada powiedzieć co robimy dla „trywialnych" wyrażeń regularnych, by indukcja się spięła. Jeśli wyrażene jest postaci a, dla $a \in \Sigma$, tworzymy trywialnie automat który

akceptuje jedynie słowo takiej postaci. Dla pustego wyrażenia tworzymy automat który nie akceptuje niczego, dla 1 tworzymy (równie trywialnie) automat który zaakceptuje jedynie ε.

Mając z głowy nudne przypadki bazowe, możemy przejść do ciekawszej rzeczy dowodu, czyli konwersji poszczególnych wyrażeń regularnych na automaty. Będziemy się kejsować po tym, na co jesteśmy w stanie „rozbić" regex (załączone ilustracje powinny bardziej rozjaśnić)[1]:

- Jeśli wyrażenie regularne jest postaci $\alpha_1 + \alpha_2$, to z założenia indukcyjnego mamy, że istnieją DFA A_1, A_2 takie, że $L(A_1) = L(\alpha_1)$ i $L(A_2) = L(\alpha_2)$. Możemy stworzyć automat A_3, który składa się z tych samych stanów i przejść co A_1 i A_2, ale ma jeszcze dodatkowy stan q; q jest stanem startowym nowego automatu i ma epsilon przejścia do stanów startowych A_1 i A_2. Zauważam, że jeśli słowo należy do języka akceptowanego przez któryś z tych automatów (nazwijmy go A), to w automacie A_3 istnieje obliczenie, które ze stanu startowego przechodzi epsilon-przejściem do stanu startowego A, po czym to słowo zostanie zaakceptowane. Jeśli żaden z automatów nie akceptował tego słowa, to trywialnie widać że nowy automat również go nie zaakceptuje.
- Jeśli wyrażenie regularne jest postaci $\alpha_1 \cdot \alpha_2$, to, podobnie jak powyżej, muszę wymyślić jakiś sposób na zrobienie nowego automatu mając automaty A_1 i A_2 tak, by ten akceptował konkatenację tych języków. Ponownie więc tworzymy A_3, który ma wszystkie stany i przejścia ze wspomnianych automatów wraz z nowym stanem startowym q. Dodaję ε-przejście z q do stanu startowego z A_1 oraz ε-przejścia ze stanów akceptujących A_1 do stanu startowego A_2. Stanami akceptującymi nowego automatu są **tylko** stany akceptujące z A_2. Widać że to działa ładnie – jeśli jakieś słowo w jest postaci xy, gdzie $x \in L(A_1)$ i $y \in L(A_2)$, to istnieje obliczenie które „przejdzie" x do stanu akceptującego, następnie „przeskoczy" po epsilonie do A_2, który z kolei zaakceptuje y. Jeśli zaś słowo nie należy do konkatenacji takich języków, to znaczy że taki podział nie istnieje, a więc albo w ogóle nie „przejdziemy" do A_2, albo po przejściu do A_2 nie zostaniemy zaakceptowani.
- Jeśli wyrażenie regularne jest postaci $(\alpha)*$, to mamy jakiś automat A, dla którego $L(A) = L(\alpha)$. Chcemy teraz zrobić jakąś „owijkę" na niego tak, by można było zaakceptować dowolne słowo będące „zlepkiem" słów z tego języka (bo to gwiazdka kleenego). Konstrukcja jest całkiem oczywista: robimy automat B, który, standardowo, ma wszystkie stany i przejścia z automatu A. Dodajemy oddzielny stan startowy q; dodajemy z niego ε-przejście do stanu startowego A. Dodajemy jeszcze ε-przejścia ze stanów akceptujących A z powrotem do stanu q. Stanem akceptującym również jest q. Zauważmy, że jeśli jakieś słowo należy do języka generowanego przez $(\alpha)^*$, to istnieje jego podział na słowa które należą do $L(\alpha)$, a więc istnieje obliczenie które będzie sobie akceptować po kolei te „podsłowa", a na samym kończu się zakończy. Jednocześnie, cokolwiek co jest zaakceptowane przez B musi być takim „zlepkiem", bo by „wrócić" do q trzeba było przejść sekwencją przejść po literach stanowiących słowo matchujące wyrażenie regularne.

1.4 Lemat o pompowaniu

1.4.1 Twierdzenie

Lemat 1.4.1 (O pompowaniu)**.** Jeśli L jest językiem regularnym to:

$$\exists_{n>0} : \forall_{w:|w|\geq n} \exists_{xyz=w} : |xy| \leq n \wedge |y| \geq 1 \wedge \forall_{i\in\mathbb{N}} : xy^iz \in L$$

Dowód. Skoro L jest językiem regularnym to istnieje DFA A, który rozpoznaje L.

[1] kiedyś się pojawią, mówię Wam

Niech $n = |Q|$ i weźmy dowolne słowo w dla którego $m = |w| \geq n$.

Skoro słowo jest akceptowane to istnieje ścieżka $s = q_0, \dots q_m \in F$

Mamy więc $m+1 > n$ stanów, czyli jakiś stan musi się powtarzać. Z takich stanów wybieramy q_i, którego pierwsze powtórzenie jest najwcześniej. Niech q_j będzie drugim wystąpieniem stanu q_i

Mamy zatem $x = a_0 \dots a_{i-1}$, $y = a_i \dots a_{j-1}$, $z = a_j \dots a_{m-1}$

Oczywiście $|xy| \leq n$ bo inaczej q_i nie powtarzałby się najwcześniej.

Czynimy teraz fajną obserwację, mianowicie skoro $q_i = q_j$ to słowo y może wystąpić dowolną (również zerową) liczbę razy w akceptowanym słowie.

Innymi słowy, dla dowolnego $i \in \mathbb{N}$ $xy^iz \in L$ □

Proszę pamiętać, że jest to warunek konieczny, ale **nie jest on wystarczający** by język był regularny. Istnieją języki które zdecydowanie nie są regularne, a spełniają warunki przedstawione w lemacie o pompowaniu.

1.4.2 Przykładowe zastosowania

Pokażemy najbardziej klasyczny przykład znany ludzkości.

Twierdzenie 1.4.1. Język

$$L = \{a^n b^n : n \in \mathbb{N}\}$$

nie jest językiem regularnym.

Dowód. Będziemy usiłowali pokazać, że L *nie* spełnia lematu o pompowaniu. To oznacza, że będziemy musieli pokazać, że zachodzi dla niego zaprzeczenie własności opisanej w tym lemacie. Innymi słowy, chcemy pokazać że:

$$\forall_{n>0}\exists_{w\in L}\forall_{xyz=w:|xy|\leq n\wedge|y|\geq 1}\exists_{i\in N} xy^iz \notin L$$

Prościej o tym myśleć jak o pewnej grze: nasz przeciwnik daje nam stałą n; my odpowiadmy mu słowem; ten nam je dzieli na 3 części które spełniają warunki lematu, a my możemy „wypompować" część y. Wygrywamy jeśli słowo nie należy do L. Jeśli mamy tutaj strategię wygrywającą, to znaczy że język nie jest regularny.

W przypadku naszego języka: nasz oponent daje nam jakieś n. Pora się odegrać, odegrać słowem postaci $a^{2n}b^{2n}$. Słowo to niewątpliwie należy do L, więc mogliśmy to zrobić. Przeciwnik musi je jakoś podzielić, tak by $w = xyz$, $|xy| \leq n$ i $y \geq 1$. Widzimy tutaj bardzo śmieszną rzecz: niezależnie od podziału, $xy = a^k$ dla jakiegoś k. Wobec tego również $y = a^l$ dla niezerowego l, które jest mniejsze niż n. W takim razie „depompujemy" y, ustawiając i na 0 i otrzymując słowo postaci $a^{2n-l}b^{2n}$, które z pewnością nie należy do języka. Hehe. □

1.5 Własności języków regularnych

1.5.1 Suma

Twierdzenie 1.5.1. Języki regularne są zamknięte na sumowanie.

Dowód. Mając wyrażenia regularne α_1 i α_2 możemy zrobić wyrażenie $\beta = \alpha_1 + \alpha_2$. □

1.5.2 Przecięcie

Twierdzenie 1.5.2. Języki regularne są zamknięte na przecięcie.

Dowód. Mając DFA $A_1 = (Q_1, \Sigma, \delta_1, s_1, F_1)$ i DFA $A_2 = (Q_2, \Sigma, \delta_2, s_2, F_2)$ możemy skonstruować DFA B, które będzie rozpoznawać język $L(A_1) \cap L(A_2)$, takie że:

$$B = (Q', \Sigma, \delta', q', F')$$

gdzie:

- $Q' = Q_1 \times Q_2$
- $\delta'(q_1, q_2) = (\delta_1(q_1), \delta_2(q_2))$
- $s' = (s_1, s_2)$
- $F' = \{(q_1, q_2) : q_1 \in F_1 \wedge q_2 \in F_2\}$

Intuicyjnie: DFA B symuluje chodzenie po A_1 i A_2 akceptując wtedy i tylko wtedy, gdy w obu DFA dane słowo znalazłoby się w stanie akceptującym. Formalniej:

$$w \in L(A_1) \cap L(A_2) \iff w \in L(A_1) \wedge w \in L(A_2) \iff$$
$$\tilde{\delta}_1(s_1, w) \in F_1 \wedge \tilde{\delta}_2(s_2, w) \in F_2 \iff \delta'((s_1, s_2), w) \in F' \iff w \in L(B)$$

□

1.5.3 Dopełnienie

Twierdzenie 1.5.3. Wyrażenia regularne są zamknięte na dopełnienie.

Dowód. Mając DFA A możemy „odwrócić" wszystkie stany akceptujące, tzn. stworzyć DFA B takie, że $F_B = \{q \in Q_A : q \notin F_A\}$. Wtedy:

$$w \notin L(A) \iff \tilde{\delta}(s_a, w) \notin F_A \iff \delta(s_a, w) \in F_B \iff w \in L(B)$$

□

1.5.4 Konkatenacja

Twierdzenie 1.5.4. Wyrażenia regularne są zamknięte na konkatenację.

Dowód. Mając wyrażenia regularne α_1 i α_2 możemy skonstruować wyrażenie regularne $\alpha_1\alpha_2$. □

1.5.5 Gwiazdka Kleenego

Twierdzenie 1.5.5. Wyrażenia regularne są zamknięte na gwiazdkę Kleenego.

Dowód. Mając wyrażenie regularne α możemy stworzyć wyrażenie α^*. □

1.6 A-równoważność

Definicja 1.6.1. Stany q_1, q_2 zadanego DFA są **A-równoważne** jeśli

$$\forall_{w\in\Sigma^*}\hat{\delta}(q_1,w)\in F \iff \hat{\delta}(q_2,w)\in F$$

i **A-rozróżnialne** jeśli

$$\exists_{w\in\Sigma^*}\hat{\delta}(q_1,w)\in F \iff \hat{\delta}(q_2,w)\notin F$$

Lemat 1.6.1. A-równoważnosć jest relacją równoważności.

Dowód. Zwrotność jest oczywista, jako że $\forall_{q\in Q}\forall_{w\in\Sigma^*}\tilde{\delta}(q,w)=\tilde{\delta}(q,w)$. Symetryczność również; przechodniość też. Pozostawimy je jako ćwiczenie dla czytelnika. □

1.7 Algorytm (z) D(upy)

- Wejście: DFA A
- Wyjście: (trójkątna) macierz Boolowska M, w której $M_{i,j}=1 \iff a_i, a_j$ są rozróżnialne

Na początku wypełniamy M zerami. Jeśli $a_i\in F, a_j\notin F$ to $M_{i,j}=1$. Dopóki macierz M się zmienia to dla każdych dwóch stanów q_1,q_2 oraz dla każdej litery litery a, jeśli $\hat{\delta}(q_1,a)=q_k$ i $\hat{\delta}(q_2,a)=q_l$ oraz $M_{k,l}=1$ to $M_{i,j}:=1$.

Jeśli po zakończeniu algorytmu jakieś 2 stany q_i,q_j są takie, że $M_{i,j}=0$ to wtedy wiemy, że q_i oraz q_j nie są rozróżnialne, a więc są A-równoważne. Zauważmy, że w ten sposób znajdujemy wszystkie stany które są sobie równoważne (działa to trochę jak taki Floyd-Warshall). Teraz co robimy, to kolapsujemy wszystkie równoważne stany w jeden wierzchołek. Okazuje się, że tak powstałe DFA jest minimalne pod względem liczby stanów (w porównaniu do innych DFA rozpoznających ten sam język).

Twierdzenie 1.7.1. Automat otrzymany w wyniku pracy algorytmu D jest najmniejszy spośród DFA rozpoznających dany język.

Dowód. Dowód nie wprost: załóżmy, że w wyniku minimalizacji jakiegoś automatu otrzymaliśmy jakiś automat A, a istnieje DFA B takie, że $L(A)=L(B)$ i B ma mniej stanów. Zdefiniujmy sobie DFA C[2] takie, że jego stanem startowym jest s_B, a $Q_C=Q_A\cup Q_B$, funkcje przejść również „sumujemy". Zauważmy, że dla automatu C:

$$\forall_{q_A\in Q_A}\exists_{q_B\in Q_B}\ q_a \textit{ i } q_b \textit{ są } C\textit{-równoważne}$$

Dowód tej obserwacji jest prosty: jako, że A i B akceptują te same języki, to na pewno s_A i s_B są C-równoważne. No ale w takim razie dla każdego $a\in\Sigma$ jest tak, że $\delta(s_A,a)$ i $\delta(s_B,a)$ są również C-równoważne (i tak dalej). Stosując tę konstrukcję dalej, otrzymamy że dla każego stanu z A istnieje równoważny stan w B.

Na podstawie tej informacji wykonujemy *potężną* obserwację: skoro, z założenia nie wprost, A miał więcej stanów w B; to w połączeniu z powyższą obserwacją oznacza, że istnieją stany $q_1,q_2\in A$ takie, że oba z nich są równoważne z jakimś stanem $q_3\in B$. Ponieważ jednak równoważnosć jest przechodnia, mamy że q_1 i q_2 są wzajemnie równoważne. To prowadzi do

[2]Tworzymy je tylko i wyłącznie z przyczyn formalnych, bo A-równoważnosć jest zdefiniowana dla określonego automatu.

sprzeczności, bo wiemy, że algorytm D będzie na tyle miły, że skolapsuje wszystkie stany które są wzajemnie równoważne. To kończy dowód.

□

1.8 L-równoważność (L-kongruencja)

Definicja 1.8.1. Mówimy, że relacja jest L-kongruentna dla pewnego języka $L \in \mathcal{P}(\Sigma^*)$ wtedy i tylko wtedy, gdy:

1. $\forall_{v \in \Sigma^*} x \equiv y \implies xv \equiv yv$
2. $x \equiv y \implies (x \in L \iff y \in L)$

1.9 Twierdzenie Myhille'a-Nerode'a

Twierdzenie 1.9.1 (Myhill–Nerode)**.** Język L jest regularny wtedy i tylko wtedy gdy istnieje relacja równoważności $\equiv \subseteq \Sigma^* \times \Sigma^*$ o skończenie wielu klasach równoważności, która jest L-kongruentna.

Dowód.

$\implies$

Skoro L jest regularny to istnieje DFA A, na podstawie którego definiujemy

$$x \equiv y \iff \hat{\delta}(s, x) = \hat{\delta}(s, y)$$

Należy teraz dowieść, że:

1. Relacja ta jest L-kongruentna:
   (a) $x \equiv y \iff \hat{\delta}(s, x) = \hat{\delta}(s, y) = q \implies \hat{\delta}(q, v) = \hat{\delta}(q, v) \iff xv \equiv yv$
   (b) $x \equiv y \iff \hat{\delta}(s, x) = \hat{\delta}(s, y) = q$, więc jeśli $q \in F$ to warunek spełniony, a jeśli $q \notin F$ to również.
2. Relacja ta ma skończenie wiele klas równoważności, bo funkcja $\hat{\delta}$ może „przemapować" dane słowo na maksymalnie $|Q|$ stanów, a wszystkie słowa które „przechodzą" na ten sam stan są ze sobą równoważne.

$\impliedby$

Tworzymy automat, którego stanami są klasy równoważności słów wyznaczone przez relację $\equiv$. Możemy tak zrobić, bo wiemy że klas równoważności jest skończenie wiele. Jako stany finalne ustalamy te klasy równoważności, w których wszystkie elementy należą do języka (z racji że jest to L-kongruencja wszystkie elementy klasy mogą albo należeć, albo nie należeć do języka). Stanem startowym jest klasa, do której należy ε.

Teraz żeby to się spięło i w ogóle, należy jeszcze ustalić jak wyglądają funkcje przejścia. Robimy to w ten sposób, że dla każdej klasy próbujemy „dokleić" jedną literkę z alfabetu i dodajemy przejście do klasy, w której wylądują powstałe słowa. Warto tutaj zauważyć, że z racji że jest to L-kongruencja, jeśli dodam literkę $a \in \Sigma$ do dowolnych słów z mojej klasy, to wszystkie słowa powstałe jako rezultat muszą znaleźć się w jednej klasie (z racji warunku pierwszego w definicji L-kongruencji).

Procedura „generowania przejścia" jest wykonywana dla każdej klasy i każdej litery; zauważmy, że wówczas funkcja przejścia jest już poprawnie zdefiniowaną funkcją.

Teraz należy udowodnić, że powstały DFA akceptuje język, na którym zdefiniowana jest L-kongruencja.

Zauważamy, że $\hat{\delta}(s, w)$ jest klasą równoważności, do której należy słowo w. Dowód przebiega z pomocą indukcji po długości w: gdy $|w| = 0$, to $w = \varepsilon$ i z definicji q_s jako klasy zawierającej epsilon wszystko działa poprawnie.

W kroku indukcyjnym, jeśli mamy słowo $w' = wa$, to z założenia indukcyjnego wiemy, że $\hat{\delta}(w)$ jest klasą równoważności, do której należy w. Wówczas przejście po literze a z tamtej klasy prowadzi nas do klasy zawierającej słowo wa, z tego jak zdefiniowaliśmy funkcję δ.

Cóż możemy zrobić z tą fascynującą obserwacją? Zauważyć, że teraz pokazanie że $L(A) = L$ mamy teraz za darmo, bo jeśli $w \in L$ to klasa równoważności która zawiera w jest stanem akceptującym; w przeciwnym razie nie jest. To dowodzi poprawności konstrukcji.

□

1.10 Podstawowe problemy decyzyjne dotyczące języków regularnych

W poniższych sekcjach będziemy zakładać, że mamy do czynienia z ε-NFA lub wyrażeniami regularnymi i będziemy poszukiwać algorytmów działających w czasie wielomianowym (tak jak na ćwiczeniach).

1.10.1 Należenie określonego słowa do języka

Mając dane ε-NFA oraz słowo $w = a_1a_2a_3 \dots a_n$ będziemy (trochę podobnie jak przy konwersji z ε-NFA na DFA) trzymać wszystkie stany, w których możemy się znaleźć po „przejściu" prefiksu danego słowa.

Z domknięcia przechodniego stanu startowego idziemy sobie literką a_0 do wszystkich stanów, które są „osiągalne" i ten zbiór również epsilon-domykamy (zauważmy, że każdy krok tego typu można robić liniowo od liczby stanów).

Formalniej, możemy chcieć sobie zdefiniować ciąg zbiorów stanów $B_0, B_1, \dots, B_n$:

$$B_0 = \{s\}^{A,\varepsilon}$$
$$B_1 = \Delta(B_0, a_0)$$
$$B_2 = \Delta(B_1, a_1)$$
$$\dots$$

Pamiętacie te delty, co nie? Były w definicji. Hehe.

W każdym razie, po wyliczeniu tego (widać że wielomianowo) chcecie sprawdzić, czy $\exists_{q \in B_n} q \in F$.

1.10.2 Niepustość języka

Mając dane ε-NFA możemy sprawdzić, czy akceptuje jakieś słowo puszając DFS od stanu startowego, szukając stanów akceptujących. Istnienie jakiejkolwiek ścieżki tego typu jest równoważne istnieniu słowa, które zostanie zaakceptowane; problem osiągalności w grafie jest, jak wiemy, dosyć mocno wielomianowy.

1.10.3 Skończoność języka regularnego

Dostajemy ε-NFA i chcielibyśmy sprawdzić, czy język który jest przez nie akceptowany jest skończony. Pierwsze co robimy, to wywalamy stany które nie są osiągalne (wielomianowo).

Jeśli wyjdzie nam że język jest pusty (czyli wywaliliśmy wszystkie stany akceptujące) to znaczy, że język jest skończony (bo jest pusty).

Ponadto wywalamy wszystkie stany, z których nie da się osiągnąć żadnego stanu akceptującego.

Co dalej? Zauważmy, że jeśli język jest nieskończony to teraz w ε-NFA musi istnieć cykl **nieepsilonowy**. W takim razie zaczynamy ważyć wszystkie krawędzie; epsilonowym dajemy 0, wszystkim innym -1. Pytanie nam się degeneruje do pytania, czy istnieje cykl ujemny – na to pytanie z kolei jesteśmy w stanie wielomianowo odpowiedź algorytmem Bellmana-Forda.

Puszczamy algorytm Bellmana-Forda; jeśli znajdzie cykl ujemny to oznacza że istnieje cykl nieepsilonowy, a biorąc pod uwagę że stan końcowy jest osiągalny z tego cyklu, to wiemy że można po nim chodzić tyle razy ile chcemy (a więc język jest nieskończony czy coś).

1.10.4 Minimalność liczby stanów DFA

Puszczamy algorytm D i sprawdzamy czy coś zmienił (takie zadanie serio było na liście).

1.11 Słowo resetujące

Definicja 1.11.1. Mówimy, że słowo w jest **resetujące**, jeśli:

$$\exists_{q\in Q}\forall_{q'\in Q}\hat{\delta}(q', w) = q$$

Innymi słowy: jest to takie słowo, że niezależnie z którego stanu byśmy je puścili, wszystko koniec końców „znajdzie się" w tym samym stanie.

Jeśli dla DFA istnieje słowo resetujące, to ma ono długość co najwyżej $\frac{n^3-n}{6}$. Nie wiemy natomiast, czy istnieje lepsze ograniczenie. Istnieje *Hipoteza Černego*[3], obniżająca to ograniczenie górne do $(n-1)^2$, ale nikt jej jeszcze nie dowiódł.

Lemat 1.11.1. Dla DFA o n stanach oraz stanów q_i, q_j jeśli istnieje w takie, że $\widehat{\delta}(q_i, w) = \widehat{\delta}(q_j, w)$ to istnieje też v, $|v| \leq n^2$ dla którego $\widehat{\delta}(q_i, v) = \widehat{\delta}(q_j, v)$

Dowód. Tworzymy multigraf skierowany par stanów $G = (Q^2, E)$. Krawędź skierowaną z etykietą a między wierzchołkami $(q, p), (q', p')$ istnieje wtedy i tylko wtedy gdy $\delta(q, a) = q' \wedge \delta(p, a) = p'$

Słowo synchronizujące dla dwóch stanów odpowiada ścieżce w naszym grafie z wierzchołka (q_i, q_j) do pewnego (q, q).

[3] Ján Černý był Czechem

Z założenia mamy, że istnieje pewna taka ścieżka, być może dłuższa niż n^2. Zauważmy jednak że mamy n^2 wierzchołków, a wierzchołków na ścieżce nie ma sensu powtarzać, zatem istnieje ścieżka o co najwyżej n^2 krawędziach.

Słowo v z tezy odzyskujemy czytając etykiety na krawędziach tejże ścieżki. □

Twierdzenie 1.11.1. Jeśli DFA ma słowo resetujące to ma ono długość co najwyżej $|Q|^3$

Dowód. Niech $Q = \{q_1, \ldots, q_n\}$. Skoro istnieje słowo resetujące dla całego automatu to w szczególności jest ono resetujące dla każdej pary (q_i, q_j). Z poprzedniego lematu wiemy, że każdą parę umiemy zsynchronizować słowem długości co najwyżej n^2.

Prowadzi nas to do następującego algorytmu:

1. Niech $q_{1,1} = q_1, \ldots, q_{n,1} = q_n$
2. Niech $w_1 = \varepsilon$
3. Dla $i = 1, \ldots, n-1$:
   (a) Znajdź słowo resetujące v dla stanów $q_{i,i}, q_{i+1,i}$
   (b) $w_{i+1} = w_i v$
   (c) Ustaw $q_{j,i+1} = \widehat{\delta}(q_{j,i}, v)$
4. w_n jest słowem resetującym całego automatu.

Skonkatenowaliśmy $n-1$ słów długości co najwyżej n^2 – zatem $|w_n| \leq n^3$.

Indukcyjnie pokazujemy, że w_i synchronizuje pierwsze i stanów, a zatem w_n jest resetujące dla całego automatu.

□

Rozdział 2

Języki bezkontekstowe

2.1 Gramatyki bezkontekstowe

Definicja 2.1.1. Gramatyka bezkontekstowa (Context-Free Grammar) to czwórka $G = (N, \Sigma, P, S)$ gdzie

- N to skończony zbiór zmiennych (nieterminale)
- Σ - alfabet (terminale)
- P - produkcje $P \subseteq N \times (N \cup \Sigma)^*$
- $S \in N$ - symbol startowy

Definicja 2.1.2. Forma zdaniowa to dowolne słowo nad $(N \cup \Sigma)^*$

Intuicyjnie – gramatyka bezkontekstowa definiuje zasady na podstawie których możemy tworzyć nowe słowa nad alfabetem Σ. Nieterminale N są takimi stanami pośrednimi, które możemy podmieniać używając produkcji z P. Każda produkcja jest postaci

$$\text{nieterminal} \to \text{forma zdaniowa}$$

Bardziej konkretne przykłady zobaczymy w sekcji Konstrukcja CFG i PDA dla prostych języków.

Definicja 2.1.3. Dla gramatyki G definiujemy relację $\to_G$ na formach zdaniowych. Mówimy, że

$$\alpha \to_G \beta$$

wtedy i tylko wtedy gdy:

$$\exists_{\alpha_1,\alpha_2,\gamma \in (N \cup \Sigma)^*} : \exists_{A \in N} : (A, \gamma) \in P \wedge \alpha = \alpha_1 A \alpha_2 \wedge \beta = \alpha_1 \gamma \alpha_2$$

Bardziej po ludzku – formę zdaniową β możemy uzyskać (w jednym kroku) z formy zdaniowej α o ile w α jest jakieś wystąpienie nieterminala A, które możemy zgodnie z produkcjami podmienić na formę zdaniową γ aby uzyskać β.

Definicja 2.1.4. $\to_G^*$ to zwrotne i przechodnie domknięcie $\to_G$

Z α możemy uzyskać β w dowolnej liczbie kroków, jeśli istnieje ciąg przekształceń $\alpha \to_G \alpha_1 \to_G \cdots \to_G \gamma$

Definicja 2.1.5. Język generowany przez gramatykę G to

$$L(G) = \{w \in \Sigma^* \mid S \to_G^* w\}$$

podobnie dla $A \in N$

$$L(G, A) = \{w \in \Sigma^* \mid A \to_G^* w\}$$

Definicja 2.1.6. Język jest bezkontekstowy (Context-Free Language) jeśli jest generowany przez jakąś gramatykę bezkontekstową.

2.1.1 Postać normalna Greibach

Definicja 2.1.7. Mówimy że gramatyka G jest w **postaci normalnej Greibach** jeśli każda produkcja jest w postaci:

$$A \to aB_1 \dots B_n$$

gdzie $a \in \Sigma$, $B_i \in N$, $n \in \mathbb{N}$

Dopuszczamy produkcję $S \to \varepsilon$ jeśli $\varepsilon \in L(G)$ ale wtedy S nie występuje po prawej stronie żadnej produkcji.

Zaznaczmy tylko, że Sheila Greibach jest kobietą, nie jest to więc postać Greibacha.

2.2 Automaty ze stosem

2.2.1 Nieformalna definicja

Tak jak w przypadku DFA i NFA mieliśmy automaty z jakimiś stanami, które przechodziły (albo nie) po kolejnych literach, tak automaty ze stosem mają jeszcze dodatkowo stos, na podstawie którego możemy podejmować decyzję co zrobić.

2.2.2 Formalnie

Definicja 2.2.1. Automat ze stosem (pushdown automaton PDA) definiujemy jako

$$P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

gdzie

- Q – zbiór stanów
- Σ – skończony alfabet słów
- Γ – skończony alfabet stosu
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to \mathcal{P}(Q \times \Gamma^*)$ – funkcja przejścia
- q_0 – stan startowy
- Z_0 – stosowy symbol startowy
- $F \subseteq Q$ – zbiór stanów akceptujących

Intuicyjnie δ dla każdego stanu q, litery a, symbolu na szczycie stosu z oddaje zbiór nowych możliwych stanów wraz z symbolami które mają być dodane na stos. Symbol z jest usuwany ze szczytu stosu w momencie przejścia, wiele nowych symboli może zostać dodanych. Jeśli

zdarzy się, że opróżnimy stos to mamy tak zwany przypał (automat się zacina), ale się tym nie przejmujemy.

Warto zauważyć, że $\delta(q, a, z)$ jest **zbiorem**, a więc PDA jest niedeterministyczny (automat może wybrać dowolne z przejść). Istnieje deterministyczna wersja PDA o której powiemy więcej w sekcji Podklasy języków bezkontekstowych.

Definicja 2.2.2. Konfiguracja PDA to trójka

$$(q, w, \gamma)$$

gdzie

- q – aktualny stan
- w – część słowa pozostała do przeczytania
- γ – (wszystkie) symbole na stosie

Definicja 2.2.3. Dla konfiguracji PDA P definiujemy relację $\vdash_P$

$$(q, aw, X\beta) \vdash_P (p, w, \alpha\beta) \iff (p, \alpha) \in \delta(q, a, X)$$

Definicja 2.2.4. Definiujemy $\vdash_P^*$ zwrotne i przechodnie domknięcie $\vdash_P$

Intuicja stojąca za $\vdash_P$ i $\vdash_P^*$ jest taka sama jak za $\to_G$ i $\to_G^*$ tj. chcemy opisać które konfiguracje są osiągalne z których.

Lemat 2.2.1. Dla PDA P jeśli

$$(q, x, \alpha) \vdash_P^* (p, y, \beta)$$

to

$$(q, xw, \alpha\gamma) \vdash_P^* (p, yw, \beta\gamma)$$

Dowód. Skoro

$$(q, x, \alpha) \vdash_P^* (p, y, \beta)$$

to znaczy że $x = a_0 a_1 \dots a_{n-1} y$ oraz istnieje ciąg n przejść

$$(q, a_0 a_1 \dots a_n y, \alpha) \vdash_P (q_1, a_1 \dots a_n y, \alpha_1) \vdash_P \dots \vdash_P (p, y, \beta)$$

przy czym niektóre z a_i mogą być ε (bo PDA robi ε-przejścia).

Zauważamy, że nigdy nie możemy wyjść poza x bo nie możemy dodawać liter do słowa, a także nie możemy opróżnić stosu bo to zatrzymuje automat.

W takim razie możemy zastosować te same przejścia z δ uzyskując ciąg

$$(q, a_0 a_1 \dots a_n yw, \alpha\gamma) \vdash_P (q_1, a_1 \dots a_n yw, \alpha_1\gamma) \vdash_P \dots \vdash_P (p, yw, \beta\gamma)$$

□

2.2.3 Akceptacja

2.2.3.1 Akceptacja stanem akceptującym

$$L(P) = \{w \mid (q_0, w, Z_0) \vdash_P^* (q_F, \varepsilon, \gamma) \wedge q_F \in F\}$$

2.2.3.2 Akceptacja pustym stosem

$$N(P) = \{w \mid (q_0, w, Z_0) \vdash_P^* (q, \varepsilon, \varepsilon)\}$$

2.2.3.3 Równoważność akceptacji

Mamy dwie różne definicje tego co znaczy, że jakieś słowo jest akceptowane – pytanie czy są one równoważne tj. czy jeśli mamy słowo $w \in L(P)$ to umiemy skonstruować automat Q taki że $w \in N(Q)$ i na odwrót.

Okazuje się że tak, te dwie definicje są równoważne.

Twierdzenie 2.2.1. Dla języka L następujące stwierdzenia są równoważne:

1. Istnieje P taki że $L(P) = L$
2. Istnieje Q taki że $N(Q) = L$

Dowód.

(1) $\implies$ (2)

Dodajemy do Γ specjalny symbol $\perp$, który będzie oznaczał dno stosu.

Na początku ε-prześciami modyfikujemy stos tak, aby zawierał on $Z_0 \perp$.

Następnie tworzymy stan opróżniający stos q_{pop} oraz przejścia

$$\delta(q_{pop}, \varepsilon, z) = \{(q_{pop}, \varepsilon)\}$$

oraz dla każdego stanu akceptującego $q_F \in F$

$$\delta(q_F, \varepsilon, z) = \{(q_{pop}, \varepsilon)\}$$

gdzie $z \in \Gamma \cup \{\perp\}$

Pokażemy teraz, że $L(P) = N(Q)$

$L(P) \subseteq N(Q)$

Niech $w \in L(P)$. Z definicji $L(P)$ mamy dla pewnego $q_F \in F$

$$(q_0, w, Z_0) \vdash_P^* (q_F, \varepsilon, \gamma)$$

Dzięki nowym przejściom mamy

$$(q_F, \varepsilon, z\gamma) \vdash_Q (q_{pop}, \gamma)$$

oraz

$$(q_{pop}, \varepsilon, \gamma) \vdash_Q^* (q_{pop}, \varepsilon, \varepsilon)$$

Łącząc powyższe trzy otrzymujemy

$$(q_0, w, Z_0) \vdash_Q^* (q_{pop}, \varepsilon, \varepsilon)$$

zatem $w \in N(Q)$

$N(Q) \subseteq L(P)$

Niech $w \in N(Q)$ czyli z definicji, dla pewnego q mamy

$$(q_0, w, Z_0) \vdash_Q^* (q, \varepsilon, \varepsilon)$$

Zauważmy jednak, że zaczynamy z symbolem $\perp$ na dnie stosu, którego musieliśmy się w którymś momencie pozbyć.

Jedyne przejścia które na to pozwalają to dodane przez nas

$$\delta(q_F, \varepsilon, \perp) = \{(q_{pop}, \varepsilon)\}$$

oraz

$$\delta(q_{pop}, \varepsilon, \perp) = \{(q_{pop}, \varepsilon)\}$$

Ponado jeśli dla $q \neq q_{pop}$

$$\delta(q, a, z) = (q_{pop}, \gamma)$$

to musi być tak, że $q \in F$.

W takim razie, aby z konfiguracji startowej zaakceptować w przez pusty stos to musieliśmy przejść z q_F do q_{pop}.

Ponieważ od wyjścia z q_F wolno nam wykonywać jedynie ε-przejścia to skoro na końcu mamy konfigurację $(q_{pop}, \varepsilon, \varepsilon)$ to w q_F musieliśmy mieć konfigurację $(q_F, \varepsilon, \gamma)$ co jest równoważne faktowi, że $w \in L(P)$.

(2) $\implies$ (1)

Zaczynamy tak samo jak poprzednio, dodając symbol dna stosu do Γ i włożeniem go pod Z_0.

Tworzymy jeden stan akceptujący q_{acc} oraz dla każdego stanu q dodajemy przejście

$$\delta(q, \varepsilon, \perp) = \{(q_{acc}, \varepsilon)\}$$

Pokażemy teraz, że $N(Q) = L(P)$

$N(Q) \subseteq L(P)$

Niech $w \in N(Q)$. Mamy zatem

$$(q_0, w, Z_0) \vdash_Q^* (q, \varepsilon, \varepsilon)$$

Z lematu 2.2.1 mamy

$$(q_0, w, Z_0 \perp) \vdash_P^* (q, \varepsilon, \perp)$$

Dzięki nowym przejściom mamy

$$(q, \varepsilon, \perp) \vdash_P (q_{acc}, \varepsilon, \varepsilon)$$

czyli

$$(q_0, w, Z_0 \perp) \vdash_P^* (q_{acc}, \varepsilon, \varepsilon)$$

i w efekcie $w \in L(P)$

$L(P) \subseteq N(Q)$

Niech $w \in L(P)$ czyli

$$(q_0, w, Z_0) \vdash_Q^* (q_{acc}, \varepsilon, \gamma)$$

Ponieważ (poza inicjalizacją) nigdy nie dodajemy $\perp$ na stos to jedyne przejścia które prowadzą z q do q_{acc} to

$$(q, v, \perp) \vdash_P (q_{acc}, v, \varepsilon)$$

Zauważmy, że skoro w zostało zaakceptowane to $v = \varepsilon$.

Ponadto, poza ostatnim, wszystkie przejścia które wykonuje P są legalne w Q z tym, że nie mają $\perp$ na dnie stosu.

W takim razie dla pewnego q mamy

$$(q_0, w, Z_0) \vdash_Q^* (q, \varepsilon, \varepsilon)$$

czyli $w \in N(Q)$.

□

2.2.4 Równoważność PDA i CFG

Twierdzenie 2.2.2. Niech L będzie językiem. Następujące stwierdzenia są równoważne:

1. Istnieje gramatyka bezkontekstowa G taka że $L(G) = L$
2. Istnieje PDA P taki że $L(P) = L$
3. Istnieje PDA Q taki że $N(Q) = L$

Dowód. Oczywiście mamy (2) $\iff$ (3) z twierdzenia 2.2.1. Pokażemy, że (3) $\iff$ (1)

(3) $\implies$ (1)

Mając PDA P akceptujący pustym stosem chcemy stworzyć gramatykę G, która w pewnym sensie symuluje ten automat. Będziemy chcieli żeby produkcje w gramatyce odpowiadały jakimś przejściom tego automatu.

Definiujemy nieterminale jako

$$N = \{S\} \cup \{(q, z, p) : q, p \in Q, z \in \Gamma\}$$

Intuicyjnie chcemy, żeby taki nieterminal (q, z, p) reprezentował nam takie słowa w, że

$$(q, w, z\gamma) \vdash_P^* (p, \varepsilon, \gamma)$$

Dla każdego $p \in Q$ dodajemy produkcję:

$$S \to (q_0, Z_0, p)$$

Ponadto jeśli dla $a \in \Sigma$ lub $a = \varepsilon$ mamy

$$(r, z_1 \dots z_n) \in \delta(q, a, z)$$

to dodajemy dla wszystkich $(r_1, \dots, r_{n-1}) \in Q^n$ produkcję:

$$(q, z, p) \to a(r, z_1, r_1)(r_1, z_2, r_2) \dots (r_{n-1}, z_n, p)$$

Innymi słowy – jeśli zdecydujemy, że chcemy przejść ze stanu q do stanu p ściągając przy tym z ze stosu i użyjemy przy tym przejścia automatu $(q, z, p, r_1, z_1 \dots z_n) \in \delta$ to musimy teraz przejść ze stanu r do stanu p i ściągnąć wszystkie dodane symbole $z_1, \dots, z_n$ – nie wiemy jak to zrobimy, dlatego przechodzimy po wszystkich możliwych kombinacjach stanów pośrednich.

Po drodze możemy przechodzić przez dowolne stany automatu – stąd dodajemy taką produkcję dla każdych możliwych $r_1, \dots, r_{n-1}$.

Pozostaje pokazać, że $N(P) = L(G)$ – to się robi jakąś indukcją.

(1) $\implies$ (3)

Idea jest analogiczna jak poprzednio tylko tym razem z gramatyki G robimy automat P. Na szczęście jest on niedeterministyczny więc będziemy mogli „zgadnąć" produkcję a następnie za jej pomocą sparsować słowo z wejścia.

Zaczynamy od zmodyfikowania naszej gramatyki tak aby każda produkcja była postaci:

$$A \to a$$

gdzie $w \in \Sigma \cup \{\varepsilon\}$

lub

$$A \to B_1 \dots B_n$$

gdzie $B_1, \dots, B_n \in N$

Robimy to w dwóch krokach:

1. Dla każdej litery $a \in \Sigma$ dodajemy nieterminal A oraz produkcję $A \to a$
2. zastępujemy każde (poprzednio istniejące) wystąpienie litery a w produkcjach na nieterminal A

Zmiana jest niewielka, łatwo pokazać że dostajemy gramatykę produkującą ten sam język.

Teraz tworzymy PDA akceptujący przez pusty stos:

- $Q = \{q\}$ – mamy jeden stan bo będziemy korzystać wyłącznie ze stosu
- $Z_0 = S$ – stosowy symbol startowy jest początkowym nieterminalem gramatyki
- alfabetem automatu jest Σ
- $\Gamma = N$ – na stos będziemy dodawać jedynie nieterminale

Ponadto prześcia definiujemy:

- Jeśli $A \to a$ to
$$(q, \varepsilon) \in \delta(q, a, A)$$
- Jeśli $A \to B_1 \dots B_n$ to
$$(q, B_1 \dots B_n) \in \delta(q, \varepsilon, A)$$

Innymi słowy – na stosie będziemy trzymać nieterminale, które pozostały nam do przeparsowania. Jeśli widzimy nieterminal stworzony dla litery a to zjadamy a z czytanego słowa i parsujemy dalej.

Teraz pokazujemy, że faktycznie $L(G) = N(P)$

Chcemy pokazać indukcją, że dla każdego $n \in \mathbb{N} \setminus \{0\}$ oraz każdych w, A, β zachodzi

$$A \to_G^n w\beta \iff (q, w, A) \vdash_P^n (q, \varepsilon, \beta)$$

przy czym w gramatyce robimy wywód lewostronny.

- Baza indukcji $n = 1$

  Ponieważ wykonujemy jedną produkcję/jedno przejście to są dwie sytuacje – albo $k = 0, w \in \Sigma \cup \{\varepsilon\}$ albo $w = \varepsilon, k > 0$.

  W pierwszym przypadku mamy

  $$A \to_G w \iff (q, \varepsilon) \in \delta(q, w, A)$$

  natomiast w drugim

  $$A \to_G B_1 \dots B_k \iff (q, B_1 \dots B_k) \in \delta(q, \varepsilon, A)$$

  obie równoważności wynikają bezpośrednio z konstrukcji.

- Krok indukcyjny

□

2.3 Konstrukcja CFG i PDA dla prostych języków

2.4 Drzewa wywodów

Definicja 2.4.1. Derywacja albo **wywód** to ciąg form zdaniowych $\alpha_0, \dots, \alpha_n$ takich, że $\alpha_i \to_G \alpha_{i+1}, \alpha_0 = S, \alpha_n = w \in \Sigma^*$

Definicja 2.4.2. Wywód lewostronny to taki wywód $\alpha_0, \dots, \alpha_1$ w którym jeśli $\alpha_i = xA\beta_i$ to $\alpha_{i+1} = x\gamma\beta_i$

Innymi słowy - rozwijamy zawsze skrajnie lewy nieterminal.

Definicja 2.4.3. Drzewo wywodu (parse tree) to ukorzenione drzewo z porządkiem na dzieciach w którym:

- każdy wierzchołek ma etykietę z $\Sigma \cup N \cup \{\varepsilon\}$
- etykieta korzenia to S
- Jeśli wierzchołek ma etykietę $A \in N$ a jego dzieci $X_1, \dots, X_n$ to $(A, X_1 \dots X_2) \in P$

Wywód lewostronny otrzymujemy przechodząc DFSem odwiedzając dzieci od lewej do prawej.

Definicja 2.4.4. Gramatyka G jest **jednoznaczna** (unambiguous) jeśli dla każdego słowa $w \in L(G)$ istnieje dokładnie jedno drzewo wywodu (dokładnie jeden wywód lewostronny).

2.5 Postać normalna Chomsky'ego

Definicja 2.5.1. Mówimy, że symbol $A \in N \cup \Sigma$ jest **generujący**, jeśli:

$$\exists_{w \in \Sigma^*} A \to_G^* w$$

Definicja 2.5.2. Mówimy, że symbol $A \in N \cup \Sigma$ jest **osiągalny**, jeśli:

$$\exists_{\alpha,\beta \in (N \cup \Sigma)^*} S \to \alpha A \beta$$

Definicja 2.5.3. Mówimy, że symbol $A \in N \cup \Sigma$ jest **użyteczny**, jeśli jest osiągalny i generujący.

Definicja 2.5.4. Mówimy, że gramatyka G jest w **postaci normalnej Chomsky'ego** jeśli każda produkcja jest w jednej z następujących postaci:

$$A \to BC$$

gdzie $B, C \in N$, oraz

$$A \to a$$

gdzie $A \in N$ oraz $a \in \Sigma$.

Ponadto dopuszczamy, by dla stanu startowego S istniała produkcja $S \to \varepsilon$ (jeśli gramatyka akceptuje słowo puste), ale wówczas S nie może znaleźć się po prawej stronie żadnej produkcji.

Twierdzenie 2.5.1. Dla każdej gramatyki bezkontekstowej G istnieje gramatyka bezkontekstowa G' w postaci normalnej Chomsky'ego taka, że $L(G) = L(G')$.

Dowód. Dowód ten jest prosty, ale uciążliwy w realizacji (bo ma wiele kroków).

Po pierwsze musimy pozbyć się nieterminali, które nie są użyteczne:

1. Symbole które nie są osiągalne znajdujemy najzwyklejszym DFSem puszczonym po produkcjach od symbolu startowego.
2. Symbole które nie są generujące „diagnozujemy" ustawiając wszystkie terminale i epsilon jako „generujące"; następnie przechodzimy po wszystkich produkcjach i ustawiamy jako generujące wszystkie nieterminale N takie, że $N \to \alpha$, gdzie $\alpha \in N \cup \Sigma$ i każdy symbol w α jest generujący. Robimy tak dopóki coś się zmienia; widać, że działa to poprawnie.

Poprawne działanie obu powyższych algorytmów formalnie należałoby dowieść indukcją, ale no, widać o co chodzi.

Teraz zostały nam produkcje które są jakkolwiek sensownej postaci; musimy zatem wziąć na celownik te, które nadal nam mieszają szybki.

Po pierwsze, chcemy wywalić wszystkie produkcje *wymazujące*, to znaczy produkcje postaci $A \to \varepsilon$. W tym celu robimy bardzo zabawny fikołek i jeśli mamy jakąś produkcję z nieterminala B postaci $B \to \alpha A \beta$ (gdzie $\alpha, \beta \in N$) to dodajemy jeszcze produkcję $B \to \alpha\beta$, by móc zamodelować sytuację gdy A przechodzi na epsilon bez konieczności trzymania produkcji $A \to \varepsilon$.

Oczywiście, jeśli A występuje po prawej stronie jakiejś produkcji k razy, to dodajemy jakoś $2^k - 1$ produkcji tego typu (bo musimy dodać produkcję „symulującą zniknięcie" A w dowolnym miejscu).

Gdy już pozbyliśmy się produkcji wymazujących, to musimy rozprawić się z produkcjami *jednostkowymi*, czyli produkcjami typu $A \to B$ (gdzie $A, B \in N$). Intuicyjnie chcemy móc zrobić

„skok" z A do produkcji na które przechodzi B. Sformalizowanie tego jest trochę zabawniejsze (tak by działało).

Definiujemy sobie zbiór wszystkich par nieterminali takich, że są w produkcjach jednostkowych:

$$J = \left\{(A, B) \in N^2 : A \to B\right\}$$

Teraz robimy kurczę fikołka roku: **domykamy to przechodnio**. Teraz dla każdego elementu J postaci A, X, jeśli $X \to \gamma$ gdzie γ jest taką formą zdaniową, że nie składa się wyłącznie z jednego nieterminala, dodajemy produkcję $A \to \gamma$. Dzięki porobieniu takich „skrótów" do wszystkich możliwych miejsc możemy wyrąbać produkcje jednostkowe i wszystko będzie super.

Teraz musimy się uporać z dziwnymi produkcjami, gdzie po prawej stronie występują jednocześnie nieterminale i terminale. Mowa o produkcjach typu $A \to \alpha a \beta$, gdzie $a \in \Sigma$, $\alpha, \beta \in (N \cup \Sigma)^*$. Co robimy w takiej sytuacji? W sumie to nic ciekawego, dodajemy jakiś nieterminal A_a i produkcję $A_a \to a$, dzięki czemu produkcję $A \to \alpha a \beta$ możemy transformować w $A \to \alpha A_a \beta$.

Po tych wszystkich transformacjach wszystko jest już postaci $A \to \alpha$ gdzie $\alpha \in N^*$ lub $A \to a$, gdzie $a \in \Sigma$. Zmierzamy w bardzo dobrym kierunku, a jedyne co nam zostało to zamiana produkcji, które po prawej stronie mają ostro więcej niż 2 nieterminale, czyli:

$$A \to A_1 A_2 A_3 \dots A_k$$

dla $A, A_1, A_2, A_3, \dots, A_k \in N$.

Aby sobie z tym poradzić, wprowadzamy sobie nieterminale $X_1, X_2, \dots, X_{k-2}$:

$$X_1 \to A_2 X_2$$

$$X_2 \to A_3 X_3$$

$$\dots$$

$$X_{k-3} \to A_{k-2} X_{k_2}$$

$$X_{k-2} \to A_{k-1} A_k$$

Po wprowadzeniu takich nieterminali i takich produkcji, elegancko przerabiamy naszą problematyczną produkcję do postaci $A \to A_1 X_1$. Jeśli procedurę tę wykonamy dla wszystkich produkcji, to teraz wszystko, elegancko, po prawej stronie będzie mieć 2 nieterminale (jeśli będziemy je mieć).

Jeśli mamy literkę po prawej, to mamy jedną, więc jest super.

Teraz jeszcze technikalium: jeśli nasz język akceptuje puste słowo, to nieterminal startowy S musi przechodzić w epsilon; wtedy też nie chcemy by stan startowy pojawiał się po prawej stronie jakiejkolwiek produkcji. Wobec tego robimy nowy stan startowy S' i dajemy mu produkcje $S' \to S$ oraz $S' \to \varepsilon$. To kończy naszą konwersję.

□

2.6 Lemat o pompowaniu dla języków bezkontekstowych

Twierdzenie 2.6.1 (O pompowaniu dla języków bezkontekstowych)**.** Jeżeli język L nad Σ^* jest bezkontekstowy, to:

$\exists_{n_0 \in \mathbb{N}}$
$\forall_{w \in L}$
$\exists_{a,b,c,d,e \in \Sigma^*}\ w = abcde \wedge |w| \geq n_0 \wedge |bcd| \leq n_0 \wedge |bd| \geq 1$
$\forall_{i \in \mathbb{N}}\ a \cdot b^i \cdot c \cdot d^i \cdot e \in L$

Dowód. Generalnie chcemy, żeby w derywacji naszego słowa w, które pompujemy dwa razy na jednej ścieżce pojawił się ten sam nieterminal A Mamy wtedy $A \rightarrow_G^* c$ oraz $A \rightarrow_G^* bcd$ Wtedy możemy wstawić „górną" produkcję (tę co doprowadza do bcd) w miejsce „dolnej" (tej co doprowadza do c) i w ten sposób pompujemy *bcd* do *bbcdd*.

Niech $n = |N|$ – liczba dostępnych nieterminali oraz niech m – długość najdłuższej produkcji. Wtedy jeśli nasze słowo jest długości co najmniej $n_0 = m^{n+1} + 1$ to drzewo wywodu musi mieć wysokość co najmniej $n + 1$ a zatem jakiś nieterminal musi się powtórzyć na jakiejś ścieżce.

Aby zapewnić warunek $|bcd| \leq n_0$ z tezy bierzemy ten nieterminal, którego „niższe" wystąpienie jest najwyżej ze wszystkich – tj. znajduje się na głębokości co najwyżej $n + 1$.

Jeśli mamy pecha i dla takiego wyboru $|bd| = 0$ to znaczy że przeszliśmy ścieżką, która nic nie produkuje. Szukamy wtedy innego spełniającego warunek nieterminala – musi taki istnieć bo w końcu słowo które produkujemy jest długie.

□

2.7 Lemat Ogdena

2.8 Operacje na językach bezkontekstowych

Twierdzenie 2.8.1. Języki bezkontekstowe są zamknięte na sumę.

Dowód. Niech $L_1, L_2 \in CFL$. Mamy zatem gramatyki bezkontekstowe G_1, G_2

Dla języka $L = L_1 \cup L_2$ konstruujemy gramatykę $G = G_1 \cup G_2$ (zakładamy, że zbiory nieterminali tych gramatyk są rozłączne). Dodajemy do gramatyki produkcję:

$$S \rightarrow S_1 \mid S_2$$

□

Twierdzenie 2.8.2. Języki bezkontekstowe są zamknięte na konkatenację.

Dowód. Czynimy tak jak w przypadku sumy ale naszą produkcją którą dodajemy jest

$$S \rightarrow S_1 S_2$$

□

Twierdzenie 2.8.3. Języki bezkontekstowe są zamknięte na gwiazdkę Kleenego.

Dowód. Jak wyżej, ale naszą produkcją jest

$$S \to S_1 S \mid \varepsilon$$

□

Twierdzenie 2.8.4. Języki bezkontekstowe **nie** są zamknięte na przecięcie.

Dowód. Kontrprzykładem jest $L_1 = \left\{a^n b^n c^k : n, k \in \mathbb{N}\right\}$ $L_2 = \left\{a^k b^n c^n : n, k \in \mathbb{N}\right\}$. Oczywiście $L_1 \cap L_2 = \left\{a^n b^n c^n : n \in \mathbb{N}\right\}$ co wiemy że nie jest bezkontekstowe. □

Twierdzenie 2.8.5. Języki bezkontekstowe **nie** są zamknięte na dopełnienie.

Dowód. Mamy z praw de Morgana

$$(L_1 \cap L_2) = \overline{(\overline{L_1} \cup \overline{L_2})}$$

Jeśli zatem mielibyśmy zamknięcie na dopełnienie to byśmy mieli zamknięcie na przecięcie co już pokazaliśmy że nie jest prawdziwe. □

2.9 Związane problemy decyzyjne

2.9.1 Należenie do języka bezkontekstowego

Stosujemy algorytm CYK:

2.10 Podklasy języków bezkontekstowych

2.10.1 Definicje

Definicja 2.10.1. Mówimy, że gramatyka jest **liniowa** wtedy i tylko wtedy, gdy po prawej stronie każdej produkcji znajduje się maksymalnie jeden nieterminal.

Definicja 2.10.2. Definiujemy **deterministyczny automat ze stosem (DPDA)** tak jak zwykłe PDA, przy czym:

- Dla każdych $a \in \Sigma, q \in Q, b \in \Gamma$, jeśli $\delta(q, \varepsilon, b) \neq \varnothing$ to $\delta(q, a, b) = \varnothing$ (to znaczy, jeśli z jakiegoś stanu możemy wykonać ε-przejście przy danym symbolu na szczycie stosu, to nie możemy w takiej sytuacji zrobić niczego innego).
- Dla każdych $q \in Q, a \in (\Sigma \cup \varepsilon), b \in \Gamma$ jest tak, że $|\delta(q, a, b)| = 1$, to znaczy jeśli mamy zdefiniowane przejście w jakiejś sytuacji to jest ono jednoznacznie określone.

Definicja 2.10.3. Mówimy, że język $L \subset \Sigma^*$ należy do klasy języków **DCFL** jeśli istnieje takie DPDA A, że $L(A) = L$.

2.10.2 Zawierania

Aby być w stanie sensownie dowodzić, że nie ma pewnych zawierań między różnymi podklasami języków bezkontekstowych musimy wprowadzić lemat o pompowaniu dla języków generowanych przez gramatyki liniowe (czyli języków należących do klasy LinCFL).

Twierdzenie 2.10.1 (O pompowaniu dla języków liniowych)**.**

$$\forall_{L\in \text{LinCFL}}$$
$$\exists_{n\in\mathbb{N}}$$
$$\forall_{w\in L:|w|\geq n}$$
$$\exists_{uvpxy=w}|uvxy| \leq n \wedge |vx| \geq 1$$
$$\forall_{i\in\mathbb{N}} uv^ipx^iy \in L$$

Dowód. Bierzemy $n = 4 \cdot |N| \cdot k$, gdzie jako k oznaczamy długość najdłuższej formy zdaniowej po prawej stronie wszystkich wywodów w gramatyce. W wywodzie generującym każde słowo takiej długości musi pojawić się jakiś nieterminal A, taki że:

$$S \to_G^{k_0} \alpha A\beta \to_G^{k_1} \alpha\gamma A\delta\beta$$

gdzie $k_1, k_2 \in \mathbb{N}$, $\alpha, \beta, \gamma, \delta \in \Sigma^*$[1], a $A \in N$.

Zauważam, że jako że w wywodzie dopóki są nieterminale to jest dokładnie jeden (bo to jest gramatyka liniowa) to od razu wiem, że $A \to_G^{k_1} \gamma A\delta$ dla $\gamma, \delta \in \Sigma^*$. W takim razie mogę „podstawiać" coś takiego za A tyle razy ile chcę (w szczególności mogę w ogóle pominąć tę „pętlę") skąd mam, że:

$\forall_{i\in\mathbb{N}} S \to_G^* \alpha\gamma^i A\delta^i\beta$. Jednocześnie, A ma w domknięciu przechodnim jakieś słowo (bo inaczej nie pojawiłoby się w wywodzie dla słowa w). Reasumując, $\exists_{x\in\Sigma^*} A \to_G^* x$, a stąd już mamy, że:

$$\forall_{i\in\mathbb{N}} S \to_G^* \alpha\gamma^i x\delta^i\beta$$

dla pewnych $\alpha, \beta, \gamma, \delta, x \in \Sigma^*$.

Wiemy, że $|\gamma\delta| \geq 1$, bo inaczej to by oznaczało jakieś bzdurne przejście typu $A \to A$, które można wywalić i otrzymać równoważną gramatykę.

Jednocześnie $|\alpha\gamma\delta\beta \leq n$, bo zanim doszło do powtórzenia to nie mogliśmy „wypisać" więcej liter niż długość maksymalnej strony wywodu przemnożona przez liczbę nieterminali. Czy jakoś tak.

W każdym razie chyba ten dowód działa (samo twierdzenie na pewno jest prawdziwe, dude trust me). □

Twierdzenie 2.10.2. $\text{LinCFL} \not\subset \text{REG}_\Sigma$ oraz $\text{DCFL} \not\subset \text{REG}_\Sigma$

Dowód. Język $\{a^nb^n : n \in \mathbb{N}\}$ istnieje. Możemy skonstruować prostą gramatykę i proste DPDA które go rozpoznaje, ale wiemy (z lematu o pompowaniu dla wyrażeń regularnych) że język ten nie jest regularny. □

Twierdzenie 2.10.3. $\text{REG}_\Sigma \subset \text{LinCFL}$ oraz $\text{REG}_\Sigma \subset \text{DCFL}$

[1]Każda forma zdaniowa na tym wywodzie będzie mieć **dokładnie jeden** nieterminal (poza słowem, które pojawi się na końcu)

Dowód. W pierwszym przypadku dosyć prosto skonstruować gramatykę liniową, która oddaje swoimi przejściami operacje które można robić na regexach, w drugim przypadku konwersja DFA na DPDA jest trywialna. □

Teraz pora na ciekawsze zawierania (a raczej braki tychże).

Twierdzenie 2.10.4. DCFL $\not\subset$ LinCFL

Dowód. Rozpatrzmy język L_{BAL} (ten ze zbalansowanymi nawiasowaniami). Na pewno jest on w DCFL, ale z lematu o pompowaniu dla bezkontekstowych języków liniowych się on wywali (wystarczy spróbować zdepompować słowo postaci $(^n)^n(^n)^n$, gdzie n to stała z lematu o pompowaniu). □

Twierdzenie 2.10.5. LinCFL $\not\subset$ DCFL

Dowód. Zbiór $\{a^nb^n : n \in \mathbb{N}\} \cup \{a^nb^{2n} : n \in \mathbb{N}\}$ jest LinCFL, ale deterministyczny automat ze stosem ma tylko jedną ścieżkę obliczeń, więc po dojściu do pierwszego stanu akceptującego (po wczytaniu a^nb^n) możemy wszystkie następne c traktować jako b i zaakceptować $a^nb^nc^n$. $a^nb^nc^n$ nie jest CFL. Sprzeczność. □

Twierdzenie 2.10.6. LinCFL $\subset$ CFL oraz DCFL $\subset$ CFL

Dowód. Oczywiste. □

Twierdzenie 2.10.7. CFL $\not\subset$ LinCFL oraz CFL $\not\subset$ DCFL

Dowód. Skoro LinCFL i DCFL nie są wzajemnie swoimi podzbiorami, to każdy z nich musi mieć element taki, że znajduje się on w CFL, ale nie w jednym z nich; to kończy dowód. □

2.11 Gramatyki liniowe

Definicja 2.11.1. Mówimy, że gramatyka jest **prawoliniowa** wtedy i tylko wtedy, gdy każda produkcja ma postać:

$$A \to aB$$

lub

$$A \to a$$

gdzie $a \in \Sigma^*$ oraz $A, B \in N$.

Definicja 2.11.2. Mówimy, że gramatyka jest **silnie prawoliniowa** wtedy i tylko wtedy, gdy każda produkcja ma postać:

$$A \to aB$$

lub

$$A \to \varepsilon$$

gdzie $a \in \Sigma$ i $A, B \in N$.

Twierdzenie 2.11.1. Następujące stwierdzenia są sobie równoważne dla dowolnego języka $L \subset \Sigma^*$:

1. L jest generowany przez pewno wyrażenie regularne.
2. L jest generowany przez pewną gramatykę prawoliniową.
3. L jest generowany przez pewną gramatykę silnie prawoliniową.

Dowód. Równoważność (2) i (3) jest dosyć widoczna: każda gramatyka silnie prawoliniowa jest w szczególności prawoliniowa, natomiast w drugą stronę możemy wykonać konwersję bardzo podobną do tej, którą stosowaliśmy przy otrzymywaniu CNF. Tym samym pozostawiamy ją jako ćwiczenie dla czytelnika.

Ciekawszą rzeczą jest konwersja DFA na gramatyki prawoliniowe i z powrotem.

Jeśli mamy DFA A, to konwertujemy je do postaci gramatyki G następująco:

1. Ustalamy zbiór nieterminali gramatyki N jako zbiór stanów Q naszego DFA.
2. Dla każdej litery $a \in \Sigma$ i każdego stanu $q \in Q$ dodajemy produkcję taką, że:

$$q \to a \cdot \delta(q, a)$$

3. Nieterminal startowy naszej gramatyki to stan q_s.
4. Dla każdego stanu $q \in F$ dodajemy produkcję:

$$q \to \varepsilon$$

Otrzymana gramatyka jest w oczywisty sposób prawoliniowa (i to silnie!). Wypadałoby jakoś dowieść równoważność tej gramatyki i DFA, ale to nawet widać: jeśli DFA akceptuje słowo w, to oznacza że mam w nim ścieżkę od stanu startowego do końcowego, która „zjada" po kolei literki z w; jestem w stanie ją odtworzyć chodząc tymi samymi przejściami z gramatyki. Formalnie pewnie poleciałaby jakoaś indukcja.

Dowód w drugą stronę przebiega bardzo podobnie, bo jeśli mam jakiś wywód, to jestem w stanie go trywialne przekształcić na ścieżkę w DFA, korzystając z konstrukcji.

Teraz konstruujemy automat z gramatyki silnie prawoliniowej; robimy tym razem ε-NFA, bo możemy i będzie prościej. Konstrukcja leci bardzo podobnie, tylko że na odwrót: każdy nieterminal A staje się stanem i ma przejście po literze a do stanu B jeśli w gramatyce znajduje się produkcja postaci:

$$A \to aB$$

Zauważmy, że to musi być teraz NFA, bo może być wiele produkcji przechodzących z jednego nieterminala dodając tę samą literkę.

Stanem startowym staje się nieterminal startowy, a akceptującymi stają się te nieterminale, które mogą przejść na ε.

Jeśli mamy wywód akceptujący dla słowa w w gramatyce, to nasze NFA z pewnością je zaakceptuje (idąc krawędziami „pochodzącymi" od produkcji). Z kolei jeśli nasze NFA akceptuje jakieś słowo, to jestem w stanie „odtworzyć" to przejście w gramatyce (to serio widać, formalnie pewnie poleci jakaś indukcja po długości słowa).

□

Rozdział 3

Języki rekurencyjne i rekurencyjnie przeliczalne

3.1 Teza Churcha

Teza Churcha jest nieformalną tezą głoszącą, że jakaś funkcja matematyczna z liczb naturalnych może być policzona za pomocą *efektywnej metody* (to znaczy takiej, którą można w jakiś sposób w pełni zautomatyzować) wtedy i tylko wtedy gdy może być obliczona przez Maszynę Turinga.

W pierwszej połowie XX wieku próbowano sformalizować pojęcie **funkcji obliczalnych**, w związku z czym powstały 3 sformalizowane modele obliczeń:

1. Rachunek λ
2. Funkcje pierwotnie rekurencyjne
3. Maszyny Turinga

Okazuje się, że funkcja jest λ-obliczalna wtedy i tylko wtedy gdy jest pierwotnie rekurencyjna, oraz że jest pierwotnie rekurencyjna wtedy i tylko wtedy gdy jest obliczalna przez Maszynę Turinga.

Z racji równoważności tych trzech modeli obliczeń ludzie doszli do wniosku, że być może nie ma lepszego sposobu na automatyczne obliczanie funkcji.

3.2 Maszyna Turinga

Definicja 3.2.1. Maszyna Turinga to tupla

$$MT = (Q, \Sigma, \Gamma, \delta, q_0, \sqcup, F)$$

gdzie

- Q – skończony zbiór stanów
- $\Sigma \subseteq \Gamma$ – skończony alfabet wejściowy
- Γ – skończony alfabet taśmowy
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{-1, +1\}$ – stan, litera $\to$ nowy stan, zmiana litery, ruch głowicą (lewo, prawo)

- q_0 – stan startowy
- $\sqcup \in \Gamma \setminus \Sigma$ – pusty symbol / zero (domyślny symbol taśmy)
- F – zbiór stanów akceptujących

Ponadto zakładamy, że $Q \cap \Gamma = \varnothing$.

Definicja 3.2.2. Opis chwilowy (konfiguracja) Maszyny Turinga to słowo nad językiem $\Gamma^* q \Gamma \Gamma^*$.

Definicja 3.2.3. Konfiguracja startowa Maszyny Turinga to słowo postaci $q_0 w \sqcup$.

Definicja 3.2.4. Definiujemy relację $\vdash_M$ na konfiguracjach MT.

Przejście w lewo „wewnątrz taśmy”:

$$\alpha A q B \beta \vdash_M \alpha q' A B' \beta$$

gdzie $\alpha, \beta \in \Gamma^*$, $A, B \in \Gamma$, $q, q' \in Q$, oraz $\delta(q, B) = (q', B', -1)$.

Analogicznie definiujemy przejście w prawo „wewnątrz taśmy”:

$$\alpha q B \beta \vdash_M \alpha B' q' \beta$$

gdzie $\alpha, \beta \in \Gamma^*$, $B \in \Gamma$, $q, q' \in Q$, oraz $\delta(q, B) = (q', B', 1)$.

Definiujemy również „skrajne” przejścia – przejście w prawo „na skraju taśmy”:

$$q B \vdash_M B' q' \sqcup$$

gdzie $B, B' \in \Gamma$, $q, q' \in Q$, oraz $\delta(q, B) = (q', B', 1)$

Przejście w lewo „na skraju taśmy”:

$$q B \vdash_M q' \sqcup B'$$

gdzie $B, B' \in \Gamma$, $q, q' \in Q$, oraz $\delta(q, B) = (q', B', -1)$

Definicja 3.2.5. Język akceptowany przez Maszynę Turinga definiujemy jako

$$L(M) = \left\{ w \in \Sigma^* \mid \exists_{q_f \in F} \exists_{\alpha, \beta \in \Gamma^*} q_0 w \sqcup \vdash_M^* \alpha q_f \beta \right\}$$

Twierdzenie 3.2.1. Maszyny Turinga nie wolno nazywać „automatem”.

Dowód. Maszyna Turinga jest narzędziem dużo potężniejszym niż godne pogardy poznane dotychczas automaty. Dlatego należy się jej odpowiedni szacunek. □

Wniosek 3.2.1. Nazwanie Maszyny Turinga „automatem” skutkuje skreśleniem z listy studentów.

3.3 Niedeterministyczna Maszyna Turinga

3.3.1 Definicja

Definicja 3.3.1. Niedeterministyczną Maszynę Turinga definiujemy tak samo jak deterministyczną Maszynę Turinga, przy czym funkcja δ zostaje relacją; ciągnie to za sobą dosyć oczywiste konsekwencje w definiowaniu relacji $\vdash$.

Co ciekawe, definicja języka akceptowanego się nie zmienia; nadal zwyczajnie chcemy mieć stan akceptujący w domknięciu przechodnim.

3.3.2 Równoważność z deterministyczną Maszyną Turinga

Twierdzenie 3.3.1. Następujące stwierdzenia są równoważne dla dowolnego języka $L \subset \Sigma^*$:

1. Istnieje deterministyczna Maszyna Turinga DM taka, że $L(DM) = L$.
2. Istnieje niedeterministyczna Maszyna Turinga NM taka, że $L(NM) = L$.

Dowód. Dowód faktu, że mając deterministyczną Maszynę Turinga możemy utworzyć niedeterministyczną Maszynę Turinga jest trywialny (każda funkcja jest relacją).

Skupmy się zatem na dowodzie w drugą stronę; mamy daną niedeterministyczną Maszynę Turinga N i chcemy stworzyć równoważną deterministyczną Maszynę Turinga M. Aby to uczynić, będziemy (intuicyjnie) chodzić BFS-em po nieskończonym grafie konfiguracji maszyny N.

Formalizacja tego jest bardzo ciężka, mimo faktu że idea jest prosta (zatem podamy ideę). W alfabecie chcemy mieć jakiś separator, który jedyne do czego będzie używany, to do rozpatrywania kolejnych konfiguracji „w kolejce".

Na początku „w kolejce" konfiguracji do analizy mamy jedynie konfigurację startową. Mając głowicę w jakimś położeniu, znając jej stan i to co jest „na prawo" od niej, dopisujemy do kolejki symbol separacji konfiguracji, a następnie całą naszą konfigurację po wykonanym którymś przejściu z maszyny N. Procedurę tę wykonujemy dla każdego możliwego przejścia z N.

Gdy „przerobimy" wszystkie możliwe przejścia z N (dla każdej sytuacji jest ich skończenie wiele, więc zakodowanie tego jak M ma się zachowywać by wygenerować wszystkie te konfiguracji jest osiągalne), przechodzimy do następnej konfiguracji z kolejki. No BFS, co tu jeszcze można powiedzieć?

Za każdym razem, gdy analizujemy nową konfigurację, sprawdzamy czy zapisany w niej stan jest akceptujący; jeśli tak, to wtedy akceptujemy słowo. Jeśli nie ma zdefiniowanych przejść dla określonej sytuacji, to nie robimy nic i przechodzimy do kolejnej konfiguracji w kolejce.

Again, formalizowanie tego wymagałoby wprowadzenia dodatkowych stanów które trzymałyby informacje o tym, co „widzi" maszyna N; ponadto wszystkie stany maszyny N muszą wejść do alfabetu taśmowego maszyny M by móc je również trzymać taśmie. To wszystko jest wykonywalne, ale przykre w implementacji. □

3.4 k-taśmowa maszyna Turinga

Jest równoważna zwykłej maszynie. Czemu ? Bo możemy trzymać wszystkie taśmy szeregowo na jednej taśmie. Aby zrobić przejście przechodzimy od lewej do prawej po całej taśmie i zbie-

ramy informacje o położeniu i stanach głowic – następnie wykonujemy odpowiednie przejścia rozszerzając przy tym reprezetancje taśm jeśli trzeba.

3.5 PDA z k stosami

Okazuje się że jeśli dodamy PDA drugi stos to jest on równoważny Maszynie Turinga – pierwszego stosu będziemy używali jako taśmy, drugiego będziemy używali jako bufor aby móc dowolnie przeglądać i modyfikować pierwszy stos.

3.6 Uniwersalna Maszyna Turinga

Maszyny Turinga są super, ale czy taka maszyna potrafi zasymulować jakąś inną, być może dowolną maszynę? Otóż okazuje się że tak – umiemy skonstruować taką Maszynę Turinga, która jak dostanie na wejściu zakodowaną Maszynę Turinga M oraz jej wejście w to będzie w stanie zasymulować zachowanie M na w i zwrócić (zakodowany w tym samym formacie co wejście) wynik symulacji.

Niech $M = (Q, \Sigma, \Gamma, \delta, q_0, \sqcup, F)$ będzie (deterministyczną) maszyną którą chcemy symulować. Pokażemy jak zakodować M nad alfabetem $\Sigma_U = \{0, 1\}$.

- Niech $Q = \{q_1, \ldots, q_n\}$

  Kodujemy q_i jako ciąg i jedynek.

- $\sqcup$ będziemy kodować jako pojedynczą jedynkę.

- Podobnie niech $\Sigma = \{a_1, \ldots, a_n\}$

  Kodujemy a_i jako ciąg $i + 1$ jedynek.

- Dla $\Gamma = \{z_1, \ldots z_n\}$

  Kodujemy z_i jako ciąg $i + |\Sigma + 1|$ jedynek – dla odróżnienia od Σ.

- δ jest funkcją, czyli zbiorem par $((q_1, z_1), (q_2, z_2, \pm 1)$

  Parę $((q_i, z_j), (q_k, z_l), 1)$ zakodujemy jako

  $$\underbrace{1\ldots1}_{i}0\underbrace{1\ldots1}_{j}0\underbrace{1\ldots1}_{k}0\underbrace{1\ldots1}_{l}01$$

  Natomiast parę $((q_i, z_j), (q_k, z_l), -1)$ zakodujemy jako

  $$\underbrace{1\ldots1}_{i}0\underbrace{1\ldots1}_{j}0\underbrace{1\ldots1}_{k}0\underbrace{1\ldots1}_{l}011$$

  Całą deltę kodujemy kodując wszystkie jej pary (w dowolnej kolejności) i oddzielając je dwoma zerami.

- $F = \{q_{i_1}, \ldots q_{i_n}\}$ kodujemy jako

  $$\underbrace{1\ldots1}_{i_1}0\ldots0\underbrace{1\ldots1}_{i_n}$$

Całą maszynę M zapisujemy wpisując δ, q_0, F oddzielając je ciągiem 000.

Słowo $w = a_{i_1} \dots a_{i_n}$ kodujemy jako

$$\underbrace{1 \dots 1}_{i_1} 0 \dots 0 \underbrace{1 \dots 1}_{i_n}$$

Teraz musimy jeszcze powiedzieć jak kodujemy konfigurację maszyny M a robimy to dość prosto – Konfigurację $X_1 \dots X_k q_i X_{k+1} \dots X_n$ kodujemy jako

$$0\underbrace{1 \dots 1}_{\text{kod } X_1}0\underbrace{1 \dots 1}_{\text{kod } X_k}00\underbrace{1 \dots 1}_{\text{kod } q_i}00\ \underbrace{1 \dots 1}_{\text{kod } X_{k+1}}\ 0\underbrace{1 \dots 1}_{\text{kod } X_n}0$$

Możemy założyć, że kodowanie M jest zapisane na taśmie wejściowej po której nigdy nie piszemy, zaś konfigurację M będziemy trzymać na osobnej taśmie.

Krok symulacji przebiega teraz następująco:

1. Zlokalizuj głowicę M (szukamy 00)
2. Znajdź w kodowaniu δ wpis odpowiadający przejściu na $q_i X_k$
3. Wykonaj zadane przejście – możemy wpisać nową konfigurację na osobną, tymczasową taśmę i przepisać ją z powrotem na taśmę na której operujemy.

3.7 Przykłady konstrukcji Maszyn Turinga dla prostych problemów z RE i R

3.7.1 Maszyna Turinga rozpoznająca język $a^n b^n c^n$

3.7.2 Maszyna Turinga rozpoznająca język L_{HP}

3.8 Języki rekurencyjnie przeliczalne

Definicja 3.8.1. Mówimy że język L jest **rekurencyjnie przeliczalny** jeśli istnieje Maszyna Turinga M taka że $L(M) = L$.

Zbiór języków rekurencyjnie przeliczalnych oznaczamy przez RE.

3.9 Języki rekurencyjne

Definicja 3.9.1. Mówimy, że MT M ma **własność stopu** jeśli po skończonej liczbie kroków trafia do jednego z wyróżnionych stanów q_{acc}, q_{rej} z których jedyne przejścia prowadzą z powrotem do nich samych.

Powyższa definicja działa zarówno w przypadku deterministycznym (w którym mamy jedną ścieżkę obliczeń) jak i niedeterministycznym (wtedy chcemy żeby każda ścieżka obliczeń miała taką własność). Niestety formalnie Maszyna Turinga się nie zatrzymuje, ale przyjmujemy że jak już wpadniemy w stan q_{acc} to akceptujemy słowo i się zatrzymujemy a gdy wpadniemy w stan q_{rej} to odrzucamy słowo i się zatrzymujemy.

Definicja 3.9.2. Mówimy że język L jest **rekurencyjny** albo **rozstrzygalny** jeśli istnieje Maszyna Turinga z własnością stopu M taka że $L(M) = L$.

Zbiór języków rekurencyjnych oznaczamy przez R.

3.9.1 Gramatyki typu 0

Znamy już trochę rodzajów gramatyk, ale każda ma jakieś ograniczenia, które tylko przeszkadzają nam w generowaniu naprawdę fajnych słów. To właśnie gramatyki typu 0 dadzą nam prawdziwe artystyczne spełnienie. Standardowo mamy

$$G = (N, \Sigma, P, S)$$

przy czym produkcje są w postaci $\alpha \to \beta$, gdzie $\alpha, \beta \in (N \cup \Sigma)^*$. Prawdopodobnie chcemy też mieć $\alpha \neq \varepsilon$ — nie popadajmy w paranoję, generowanie słów z pustych ciągów nie ma sensu. Ale ogólnie wygląda na to, że obie strony produkcji mogą być zasadniczo dowolnymi formami zdaniowymi, w szczególności elementami Σ^*. Dla $\Sigma = \{a, b\}$ może na przykład istnieć produkcja

$$aba \to bb$$

Ja dalej w to nie wierzę, ale tak wynikałoby z definicji, prawda?

Język generowany przez taką gramatykę definiujemy tak, jak już to wielokrotnie widzieliśmy — możemy podmieniać ciągi zgodnie z produkcjami i w ten sposób tworzyć derywacje.

Co zatem możemy takim cudem wygenerować? Okazuje się, że są to dokładnie języki RE, nie mniej, nie więcej.

Twierdzenie 3.9.1. $L \in \mathrm{RE} \iff$ istnieje gramatyka G typu 0 taka, że $L(G) = L$

Dowód. Prowadzimy go oczywiście w obie strony.

$\Longleftarrow$: Dostajemy gramatykę typu 0 i chcemy skonstruować maszynę Turinga, która akceptuje język generowany przez tę gramatykę. Mamy zatem jakieś słowo w i chcemy sprawdzić, czy da się je wygenerować. Przypomnijmy sobie, że generowanie to już teraz tylko branie elementów zbioru $(N \cup \Sigma)^*$ i podmienianie ich na inne w większym słowie. Niech więc nasza maszyna Turinga będzie niedeterministyczna i, zaczynając od symbolu S, zwyczajnie zgaduje *co* podmienić i *na co* podmienić, byle zgodnie z regułami produkcji. Po każdej takiej podmianie niech sprawdza, czy nie wyszło nam przypadkiem słowo w. Jeśli akurat wyszło, to akceptujemy je — nasze obliczenie jest dowodem, że da się je wygenerować z zadanej gramatyki. Wiemy też, że jeśli się da, to któraś ścieżka obliczeń NMT zgadnie ten właściwy sposób. Jeśli zaś $w \notin L(G)$, to maszyna Turinga oczywiście nigdy nie zgadnie odpowiedniej derywacji i nie zaakceptujemy.

$\Longrightarrow$: Teraz na podstawie maszyny Turinga chcemy skonstruować gramatykę typu 0. Na początku ustalmy pewne założenia na temat otrzymanej maszyny M — zaraz przekonamy się, że są całkiem sensowne:

- M jest jednotaśmowa i deterministyczna — wiemy, że każda wielotaśmowa i/lub niedeterministyczna da się do takiej sprowadzić
- M w jakiś sposób oznacza najdalsze użyte komórki taśmy, z lewej i z prawej — powiedzmy, że będą to symbole $\triangleright$ i $\triangleleft$ — gdy taśma się rozszerza, te markery oczywiście mogą być przesuwane, my chcemy tylko móc się zorientować, jaki fragment taśmy został jakkolwiek ruszony

– gdy M znajdzie się w stanie $q_f \in F$, czyli zaakceptuje słowo, sprząta jeszcze po sobie — wymazuje całą taśmę i przysuwa prawy ogranicznik $\triangleleft$ do lewego ogranicznika $\triangleright$

– na koniec przechodzi w stan q_{done}, z którego nie ma już przejść.

Zatem zakładamy że konfiguracja początkowa naszej maszyny to $q_0 \triangleright w \triangleleft$, natomiast jeśli w jest akceptowane, to M zatrzymuje się w konfiguracji $\triangleright q_{\text{done}} \triangleleft$. Niby sporo założeń, ale chyba możemy wymagać od maszyny Turinga, że zostawi po sobie jako taki porządek, jeśli nie wpływa to w żaden sposób na rozpoznawany przez nią język, prawda?

Teraz przechodzimy do fajnego pomysłu: chcemy, aby derywacje w naszej gramatyce symulowały pracę M *od końca*. Intuicyjnie, chcemy dojść do słowa $w \in L(M)$, zaczynając od konfiguracji $\triangleright q_{\text{done}} \triangleleft$ i zastanawiając się, jakież to słowo mogło pojawić się na wejściu, że M zakończyła pracę w takiej właśnie konfiguracji. Mamy zatem

$$N = (\Gamma \setminus \Sigma) \cup Q \cup \{S\}$$

gdzie S jest po prostu świeżym symbolem startowym.

Jeśli dla pewnych $q, q' \in Q, a, a' \in \Gamma$ mamy

$$\delta(q, a) = (q', a', +1)$$

to dodajemy produkcję

$$a'q' \to qa$$

Widać, że taka produkcja odpowiada za „wycofanie" przejścia w prawo.

Podobnie, jeśli dla pewnych $q, q' \in Q, a, a' \in \Gamma$ mamy

$$\delta(q, a) = (q', a', -1)$$

to *dla każdego* $c \in \Gamma$ dodajemy

$$q'ca' \to cqa$$

W ten sposób umożliwiamy „wycofanie" przejścia w lewo. Musimy uwzględnić c, ponieważ nie wiemy, co wcześniej było po lewej.

Na koniec dodajemy produkcje

$$\begin{aligned} S &\to \triangleright q_{\text{done}} \triangleleft \\ q_0 \triangleright &\to \varepsilon \\ \triangleleft &\to \varepsilon \end{aligned}$$

Jako że poziom formalizmu tego rozwiązania nie budzi zastrzeżeń, widzimy, że M akceptuje w wtedy i tylko wtedy, gdy istnieje derywacja

$$S \to_G \triangleright q_{\text{done}} \triangleleft \to_G^* q_0 \triangleright w \triangleleft \to_G w \triangleleft \to_G w$$

Dokładnie tego sobie życzyliśmy. Ten dowód wymaga jeszcze dopowiedzenia paru technikaliów, ale z pewnością gorliwy czytelnik (których, jak wiemy, nie brakuje) poradzi sobie z nimi bez większych trudności.

Dowiedliśmy tym samym, że gramatyki typu 0 są wystarczające do generowania języków rozpoznawanych przez Maszyny Turinga. Można się zastanawiać czy to musi być koniecznie gramatyka typu 0 – dla niektórych języków nie (bo są proste języki), ale dla niektórych języków okazuje się że tak, co dowodzimy biorąc wybrany rodzaj gramatyki i wskazując język akceptowany przez pewną MT, ale niedajacy się wyrazić w tej gramatyce.

□

3.10 Relacje między wariantami maszyn Turinga i gramatykami typu 0

3.11 Związek między funkcjami a językami akceptowanymi przez Maszyny Turinga

3.12 Własności języków rekurencyjnych i rekurencyjnie przeliczalnych

3.12.1 Zamkniętość na przecięcie

Twierdzenie 3.12.1. Języki rekurencyjne są zamknięte na przecięcie.

Dowód. Mając maszynę Turinga z własnością stopu A i maszynę Turinga z własnością stopu B możemy stworzyć maszynę Turinga z własnością stopu C, która najpierw symuluje maszynę A na swoim wejściu, a potem maszynę B. Akceptuje wtedy i tylko wtedy, gdy obie maszyny akceptują. Zauważamy, żę ma ona własność stopu, bo jedyne ciekawe co robi to symuluje MT z własnością stopu. Tym samym przecięcie dwóch problemów z R musi również być w R. □

Twierdzenie 3.12.2. Języki rekurencyjnie przeliczalne są zamknięte na przecięcie.

Dowód. Konstrukcja przebiega identycznie jak w powyższym dowodzie, przy czym któraś z maszyn może się nam zaciąć na wejściu – niespecjalnie nas to obchodzi, bo w takim razie dane słowo i tak nie należałoby do przecięcia dwóch języków. Tym samym mamy maszynę Turinga (niekoniecznie z własnością stopu) która akceptuje wtedy i tylko wtedy, gdy słowo należy do przecięcia dwóch języków rekurencyjnie przeliczalnych; tym samym ta klasa również jest zamknięta na przecięcie. □

3.12.2 Zamkniętość na konkatenację

Twierdzenie 3.12.3. Języki rekurencyjne są zamknięte na konkatenację.

Dowód. Mając maszyny Turinga M_1 i M_2 chcemy □

Twierdzenie 3.12.4. Języki rekurencyjnie przeliczalne są zamknięte na konkatenację.

3.12.3 Zamkniętość na dopełnienie w językach rekurencyjnych

Twierdzenie 3.12.5. Języki rekurencyjne są zamknięte na dopełnienie.

Dowód. Jeśli język jest rekurencyjny, to znaczy, że istnieje Maszyna Turinga M taka, że zwraca TAK lub NIE w zależności od tego, czy określone słowo do niego należy czy nie.

W takim razie, otrzymując jakieś słowo w możemy otrzymać dopełnienie języka $L(M)$ negując odpowiedź M. Powstała maszyna ma nadal własność stopu, bo M ma własność stopu. □

3.12.4 Rozpacz przy dopełnianiu języków rekurencyjnie przeliczalnych

Twierdzenie 3.12.6. Jeśli $L \in RE \setminus R$, to $\overline{L} \notin RE$

Dowód. Załóżmy, że $L \in RE \setminus R$ i $\overline{L} \in RE$. Mamy więc maszynę M rozpoznającą L oraz N rozpoznającą $\overline{L}$

Konstruujemy DMT M' która:

1. Wczytuje wejście w
2. Aż do akceptacji:
   (a) wykonaj jeden krok symulacji M na w
   (b) wykonaj jeden krok symulacji N na w
3. Jeśli M zaakceptowało to wypisz „TAK”
4. Jeśli N zaakceptowało to wypisz „NIE”

Oczywiście $w \in L \vee w \in \overline{L}$ więc albo M zaakceptuje w albo zrobi to N – któraś w końcu musi. Stanie się to po skończonej liczbie kroków niezależnie od w zatem $L(M') = M$ oraz M' ma własność stopu.

Trochę przypał bo w takim razie $L \in (RE \setminus R) \cap R$ czyli nie istnieje.

□

3.12.5 Zamkniętość na sumę

Twierdzenie 3.12.7. Języki rekurencyjne są zamknięte na sumę.

Twierdzenie 3.12.8. Języki rekurencyjnie przeliczalne są zamknięte na sumę.

3.12.6 Zamkniętość na gwiazdkę Kleenego

Twierdzenie 3.12.9. Języki rekurencyjne są zamknięte na gwiazdkę Kleenego.

Twierdzenie 3.12.10. Języki rekurencyjnie przeliczalne są zamknięte na gwiazdkę Kleenego.

3.13 Enumeratory

Definicja 3.13.1. Enumerator dla zbioru $S \subseteq \mathbb{N}$ to taka maszyna Turinga, która ma dedykowaną taśmę „wyjściową” i która wypisuje na nią elementy S (i tylko elementy S, w dowolnej kolejności), zapisane unarnie i separowane znacznikiem.

Jeśli $x \in S$ to x zostanie wypisane po jakimś skończonym czasie; jeśli $x \notin S$, to nigdy nie zostanie wypisane.

Enumerator nie może cofać głowicy na taśmie wyjściowej, w szczególności modyfikować tego co już napisał.

W celach ujednolicenia zapisu umawiamy się, że liczby zapisywane są unarnie za pomocą symbolu 0, a separowane symbolem 1.

Przykładowo, jeśli na fragmencie taśmy wyjściowej enumeratora znajduje się napis

$$\ldots 1000100100100001 \ldots$$

to znaczy że wypisał on na tym fragmencie liczby 3, 2, 2 oraz 4.

3.14 Związki enumeratorów z RE i R

Każde słowo z dowolnego języka $L \subseteq \Sigma^*$ możemy traktować jako liczbę w systemie o podstawie $|\Sigma|$ – będziemy zatem utożsamiać L z jakimś $S_L \subseteq \mathbb{N}$.

Okazuje się że jest prawdziwe twierdzenie:

Twierdzenie 3.14.1. Język L jest rekurencyjnie przeliczalny wtedy i tylko wtedy gdy istnieje enumerator dla S_L

Dowód. $\Longrightarrow$

Bierzemy MT M rozpoznającą L. Będziemy symulować kolejne słowa równolegle tj. konstruujemy enumerator M' który:

1. Tworzy pustą kolejkę symulacji słów
2. Dla kolejnych $w \in \Sigma^*$:
   (a) Dodaje symulację w do kolejki
   (b) Wykonuje po jednym kroku każdej aktywnej symulacji
   (c) Jeśli któreś słowo zostało zaakceptowane to wypisz je unarnie na taśmie wyjściowej.

$\Longleftarrow$

Aby sprawdzić czy $w \in L$ zapisujemy w unarnie i do znudzenia gapimy się na wyjście enumeratora aż zobaczymy na nim w – jeśli $w \in L$ to enumerator kiedyś wypisze w i je zaakceptujemy, a jak nie no to nie.

□

Mamy też podobne twierdzenie dla języków rekurencyjnych:

Twierdzenie 3.14.2. Język L jest rekurencyjny wtedy i tylko wtedy gdy istnieje enumerator dla S_L wypisujący elementy S_L w ustalonym porządku liniowym (np. leksykograficznym).

Dowód. $\Longrightarrow$

Bierzemy MT M z własnością stopu rozpoznającą L. Będziemy symulować kolejne słowa w porządku leksykograficznym tj. konstruujemy enumerator M' który:

1. Dla kolejnych $w \in \Sigma^*$ symuluje M na w. W końcu się zatrzymamy i jeśli zaakceptowaliśmy to wypisujemy w na wyjście.

$\Longleftarrow$

Aby sprawdzić czy $w \in L$ zapisujemy w unarnie i czytamy wyjście enumeratora aż napotkamy w lub coś większego w zadanym porządku – wiemy, że w już później nie wystąpi więc możemy się spokojnie zatrzymać odrzucając w.

□

Rozdział 4

Języki kontekstowe

4.1 Gramatyki kontesktowe

Definicja 4.1.1. Gramatyka kontekstowa (Context-Free Grammar) to czwórka $G = (N, \Sigma, P, S)$ gdzie

- N to skończony zbiór zmiennych (nieterminale)
- Σ - alfabet (terminale)
- P - produkcje $P \subseteq (N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^*$
- $S \in N$ - symbol startowy

gdzie każda produkcja jest postaci:

$$\alpha_1 A \alpha_2 \to_G \alpha_1 \gamma \alpha_2$$

gdzie $\alpha_1, \alpha_2, \gamma \in (N \cup \Sigma)^*$ oraz $A \in N$ oraz $|\gamma| \geq 1$
Praktycznie wszystko co poznaliśmy z bezkontekstowych znajduje się też tutaj.

Definicja 4.1.2. Język generowany przez gramatykę G to oczywiście

$$L(G) = \{w \in \Sigma^* \mid S \to_G^* w\}$$

Definicja 4.1.3. Język jest kontekstowy (Context-Sensitive Language) jeśli jest generowany przez jakąś gramatykę kontekstową.

Definicja 4.1.4. Gramatyka monotoniczna to gramatyka $G = (N, \Sigma, P, S)$, w której każda produkcja jest postaci

$$\alpha \to \beta$$

gdzie $\alpha, \beta \in (N \cup \Sigma)^*$ oraz $|\alpha| \leq |\beta|$

Klasycznie dopuszczamy produkcję

$$S \to \varepsilon$$

gdy S nie znajduje się po prawej stronie w żadnej produkcji

Każda gramatyka kontekstowa jest monotoniczna z definicji.
Ciekawszy jest:

Lemat 4.1.1. Dla gramatyki monotonicznej istnieje równoważna gramatyka kontekstowa.

Dowód. □

4.2 Automat ograniczony liniowo (LBA)

Definicja 4.2.1. Linear-Bounded Automaton to ograniczona MT będąca największą tuplą na tym przedmiocie

$$LBA = (Q, \Sigma, \Gamma, \delta, q_0, \sqcup, \vdash, \dashv, F)$$

gdzie

- Q – skończony zbiór stanów
- $\Sigma \subseteq \Gamma$ – skończony alfabet wejściowy
- Γ – skończony alfabet taśmowy
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{-1, +1\}$ – stan, litera $\to$ nowy stan, zmiana litery, ruch głowicą (lewo, prawo)

Przy czym jeśli $((q, \vdash), (q', X, d)) \in \delta$ to $X = \vdash, d = 1$ analogicznie po drugiej stronie taśmy

- q_0 – stan startowy
- $\sqcup \in \Gamma \setminus \Sigma$ – pusty symbol / zero (domyślny symbol taśmy)
- F – zbiór stanów akceptujących

Ponadto zakładamy, że $Q \cap \Gamma = \varnothing$.

Twierdzenie 4.2.1. Dla każdej CSG G istnieje LBA A takie że $L(G) = L(A)$

Dowód. □

4.3 Przykład konstrukcji CSG i LBA dla prostego języka kontekstowego

Na ćwiczeniach robiliśmy $a^n b^n c^n$ który w jednej grupie ćwiczeniowej zajął około 45 minut.*kmwtw*

4.4 Własności języków kontekstowych

4.4.1 Przecięcie

4.4.2 Konkatenacja

$G_1 = (N_1, \Sigma, P_1, S_1)$
$G_2 = (N_2, \Sigma, P_2, S_2)$

$$G = (S \cup N_1 \cup N_2, \Sigma, \{S \to S_1 S_2\} \cup P_1 \cup P_2, S)$$

$$S \notin N_1 \cup N_2$$

4.4.3 Dopełnienie

Idzie to z twierdzenia 6.6 (Immerman-Szelepcsény)

4.4.4 Suma

$G_1 = (N_1, \Sigma_1, P_1, S_1)$
$G_2 = (N_2, \Sigma_2, P_2, S_2)$

$$G = (S \cup N_1 \cup N_2, \Sigma_1 \cup \Sigma_2, \{S \to S_1, S \to S_2\} \cup P_1 \cup P_2, S)$$

$$S \notin N_1 \cup N_2$$

4.4.5 Gwiazdka Kleenego

Rozdział 5

Nierozstrzygalność

5.1 Język L_d oraz L_u

5.1.1 Język L_d

Definicja 5.1.1. Definiujemy język L_d następująco:

$$L_d = \{w_i : w_i \notin L(M_i)\}$$

Twierdzenie 5.1.1. $L_d \notin \mathrm{RE}$

Dowód. Załóżmy, że istnieje Maszyna Turinga M taka, że $L(M) = L_d$. Wówczas ma ona jakiś indeks k (Maszyn Turinga jest przeliczalnie wiele). Rozważmy co się dzieje ze słowem w_k:

- Jeśli $w_k \in L_d$, to znaczy, że $w_k \in L(M_k)$, ale to znaczy, że $w_k \notin L_d$, co prowadzi nas do sprzeczności.
- Jeśli $w_k \notin L_d$, to znaczy że $w_k \notin L(M_k)$, ale to znaczy, że $w_k \in L_d$, co również prowadzi do sprzeczności.

W obu przypadkach otrzymujemy sprzeczność, co kończy dowód. □

5.1.2 Język L_u

Definicja 5.1.2. Definiujemy język L_u następująco:

$$L_u = \{(M, w) : w \in L(M)\}$$

Język L_u nazywamy *językiem uniwersalnym*, bo zawiera wszystkie pary (maszyna, słowo) takie, że dana maszyna akceptuje określone słowo.

Twierdzenie 5.1.2. $L_u \in \mathrm{RE} \setminus \mathrm{R}$

Dowód. Oczywiście $L_u \in \mathrm{RE}$ bo możemy po prostu zasymulować maszynę M na słowie w na Uniwersalnej Maszynie Turinga.

Aby pokazać, że $L_u \notin \mathrm{R}$ załóżmy nie wprost, że mamy maszynę M z własnością stopu taką, że $L(M) = L_u$.

Konstruujemy maszynę M', która:

1. wczytuje wejście x
2. symuluje M na (x, x)
3. neguje wyjście M – możemy to zrobić bo M ma własność stopu

Dajemy teraz maszynie M' na wejściu w.

Mamy teraz dwie sytuacje:

- M' zaakceptowała w

$$w \in L(M') \Rightarrow (w, w) \notin L_u \Rightarrow w \notin L(w) \Rightarrow w \in L_d$$

- M' odrzuciła w

$$w \notin L(M') \Rightarrow (w, w) \in L_u \Rightarrow w \in L(w) \Rightarrow w \notin L_d$$

No i mamy przypał bo $L(M') = L_d \notin \mathrm{RE}$ – w takim razie nie może istnieć maszyna M z własnością stopu czyli $L_u \notin \mathrm{R}$.

Warto zauważyć, że jeśli M nie ma własności stopu to krok (3) maszyny M' nie ma sensu – chcemy, żeby M' zatrzymywała się na wszystkich TAK-instancjach (z definicji akceptacji), ale NIE-instancje maszyny M mogą działać w nieskończoność. Wiemy zatem raczej co do $L(M')$ nie należy niż należy.

□

5.2 Problem stopu

Innym przydatnym przykładem nierozstrzygalnego problemu jest dość znany problem stopu czyli pytanie czy Maszyna Turinga M zatrzymuje się na słowie w.

Definicja 5.2.1. Definiujemy język L_{HALT} następująco:

$$L_{HALT} = \{(M, w) : M \text{ zatrzymuje się na } w\}$$

Twierdzenie 5.2.1. $L_{HALT} \in \mathrm{RE} \setminus \mathrm{R}$

Dowód. Oczywiście $L_{HALT} \in \mathrm{RE}$ bo możemy po prostu zasymulować M na w i jeśli się zatrzymamy to odpowiedź jest „TAK" i wszystko super. Jeśli zaś odpowiedź to „NIE" to nie musimy się w ogóle zatrzymywać.

Aby pokazać że $L_{HALT} \notin \mathrm{R}$ zakładamy nie wprost, że istnieje M z własnością stopu, taka że $L(M) = L_{HALT}$.

Konstruujemy teraz M' która:

1. wczytuje wejście x
2. symuluje M na (x, x)
3. jeśli M odpowiedziała „TAK" to wpadamy w nieskończoną pętlę
4. w przeciwnym razie zatrzymujemy się w dowolnym stanie

Wrzucamy teraz do M wejście (M', M')

Mamy dwie sytuacje:

- M zaakceptowała (M', M')

  Skoro $(M', M') \in L(M) = L_{HALT}$ to oznacza, że wykonaliśmy krok (4) co ma miejsce jedynie gdy $(x, x) \notin L(M)$. Tak się składa że u nas $x = M'$ czyli $(M', M') \notin L(M)$.

- M odrzuciła (M', M')

  Tutaj sytuacja jest podobna – skoro $(M', M') \notin L(M) = L_{HALT}$ to wykonaliśmy krok (3) co ma miejsce jedynie gdy $(x, x) \in L(M)$, a ponieważ $x = M'$ to $(M', M') \in L(M)$.

Łącząc oba powyższe dostajemy $(M', M') \in L(M) \iff (M', M') \notin L(M)$ co jest oczywiście sprzeczne.

W takim razie nie istnieje M z własnością stopu rozpoznająca L_{HALT}, zatem $L_{HALT} \notin \mathrm{R}$. □

5.3 Redukcja Turinga

5.3.1 Redukcja

Definicja 5.3.1. Redukcja z języka L_1 do języka L_2 to funkcja $\sigma : \Sigma^* \to \Sigma^*$ obliczalna przez maszynę Turinga, taka że $x \in L_1 \iff L(f(x)) \in L_2$

Jeśli istnieje redukcja z L_1 do L_2 to zapisujemy to faktem $L_1 \leq L_2$

5.3.2 Redukcja w sensie Turinga (przy rozstrzygalności)

Definicja 5.3.2. Mówimy, że Maszyna Turinga M jest **maszyną z wyrocznią** L wtedy i tylko wtedy, gdy „ma ona dostęp" do zbioru L (innymi słowy, istnieje specjalna taśma wejściowa na którą M może wpisać słowo co do którego ma zapytanie i zasymulować stosowną Maszynę Turinga rozpoznającą język L.

Definicja 5.3.3. Mówimy, że istnieje **redukcja Turinga** z języka L_1 do języka L_2, jeśli istnieje Maszyna Turinga M z wyrocznią L_2 taka, że $L(M) = L_1$.

Taki dowód udowadniania nierozstrzygalności określany jest również jako dowodzenie nierozstrzygalności przed podprogram.

5.4 Twierdzenie Rice'a

Welcome to the rice fields

Okazuje się, że Maszyny Turinga nie potrafią aż tak dużo powiedzieć o językach rozpoznawanych przez maszyny Turinga.

Definicja 5.4.1. Własność języków RE to dowolne $P \subseteq \mathrm{RE}$. Mówimy, że P jest **nietrywialna** gdy $P \neq \varnothing$ i $P \neq \mathrm{RE}$.

Twierdzenie Rice'a w wersji „chwytliwej" brzmi następująco:

Każda nietrywialna własność RE *jest nierozstrzygalna.*

Jak jednak należy to rozumieć? Przecież nierozstrzygalny może być *język*, a własność jest *zbiorem języków* $P \subseteq \text{RE}$. Przykładowo, własnością może być zbiór wszystkich języków regularnych, zbiór wszystkich języków których elementem jest liczba 42 czy zbiór wszystkich języków których elementem jest zbiór pusty. Definiujemy teraz język złożony z **kodowań Maszyn Turinga takich, że rozpoznają jakiś język z własności**:

$$L_P = \{M : L(M) \in P\}$$

I to właśnie o takim języku twierdzenie Rice'a mówi, że jest nierozstrzygalny. Możemy teraz zapisać bardziej formalnie:

Twierdzenie 5.4.1 (Rice)**.**

$$\forall_{P \subseteq \text{RE}}((P \neq \varnothing \wedge P \neq \text{RE}) \implies \{M : L(M) \in P\} \notin \text{R})$$

Dowód. Niech P będzie dowolną nietrywialną własnością oraz niech $L_P = \{M : L(M) \in P\}$.

Ponieważ mówimy o przynależności do R to z lematu 3.12.5 wiemy że $L_P \in \text{R} \iff \overline{L_P} \in \text{R}$. Możemy zatem udowodnić, że $L_P \notin \text{R}$ lub $\overline{L_P} = L_{\overline{P}} \notin \text{R}$. Przyjmujemy zatem, że $\varnothing \notin P$ co będzie przydatne w konstrukcji dowodu.

Przeprowadzimy redukcję z problemu stopu. Niech $(M, w) \in L_{HALT}$ będzie dowolną jego instancją. Ponadto, ponieważ $L_P \neq \varnothing$ to możemy wybrać dowolną maszynę $N \in L_P$.

Konstruujemy teraz instancję problemu L_P, czyli Maszynę Turinga M'.

M' działa następująco:

1. wczytuje wejście x
2. symuluje M na w
3. symuluje N na x

Zostało nam do pokazania, że $(M, w) \in L_{HALT} \iff M' \in L_P$

$\implies$

Skoro $(M, w) \in L_{HALT}$ to krok (2) zawsze kończy swoje wykonanie a co za tym idzie M' wykona krok (3), czyli innymi słowy zachowuje się jak $N \in L_P$. W takim razie w tym przypadku $L(M') = L(N)$, czyli $M' \in L_P$.

$\impliedby$

Zamiast pokazywać, że $M' \in L_P \implies (M, w) \in L_{HALT}$ pokażemy kontrapozycję, czyli $(M, w) \notin L_{HALT} \implies M' \notin L_P$

Skoro $(M, w) \notin L_{HALT}$ to krok (2) nigdy się nie kończy a zatem M' nigdy nic nie zaakceptuje – innymi słowy $L(M') = \varnothing$.

Tutaj wkracza nasze założenie, że $\varnothing \notin P$ – korzystamy z niego aby dostać $M' \notin L_P$.

Mamy zatem (dla dowolnego nietrywialnego P) $L_{HALT} \leq L_P$, ale wiemy już że $L_{HALT} \notin \text{R}$ z czego mamy $L_P \notin \text{R}$, co kończy dowód.

□

Przykład 5.4.1. Pokażemy sobie, że język

$$L = \{M : 111 \in L(M)\}$$

jest nierozstrzygalny.

Dowód. Definiujemy

$$P = \{K \in \mathrm{RE} : 111 \in K\}$$

Jest to własność języków RE. Wtedy dla interesującego nas języka mamy

$$L = \{M : 111 \in L(M)\} = \{M : L(M) \in P\}$$

Oczywiście

- $P \neq \varnothing$, bo na przykład $\{111\} \in P$
- $P \neq \mathrm{RE}$, bo $\varnothing \notin P$, podczas gdy $\varnothing \in \mathrm{RE}$

Zatem P jest własnością nietrywialną, czyli z twierdzenia Rice'a wnioskujemy, że $L \notin \mathrm{R}$. □

5.5 Problemy nierozstrzygamlne

5.5.1 POST

5.5.2 Kafelkowanie

Problem kafelkowania TILING jest zadany następująco: mamy skończony zbiór kwadratowych kafelków o takich samych wymiarach 1×1 $T = \{T_1, \ldots, T_n\}$. Każdy kafelek ma cztery boki – na każdym jest zapisany jakiś symbol/kolor.

Naszym celem jest wyłożyć ćwiartkę płaszczyzny kafelkami (tj. znaleźć funkcję $f : \mathbb{N}^2 \to T$) w taki sposób aby kafelki dotykały się jedynie identycznymi symbolami. Ponadto mamy już zadany kafelek który się musi znaleźć w lewym dolnym rogu (0, 0)

Twierdzenie 5.5.1. TILING $\notin$ RE

Dowód. Pokażemy redukcję z problemu $\overline{L_{HALT}} \notin$ RE tj. dla zadanej DMT M i słowa w skonstruujemy taki zestaw kafelków (oraz kafelek narożny) że istnieje kafelkowanie wtedy i tylko wtedy gdy maszyna się **nie** zatrzymuje.

Zaczynamy od obserwacji, że jak w jest ustalone to możemy je zaszyć w stanach maszyny tj. stworzyć maszynę M' która zaczyna od wypisania w na początkowo pustą taśmę, a następnie zachowuje się identycznie jak M.

Z tego powodu przyjmujemy, że M zaczyna z pustą taśmą. Ponadto przyjmujemy, że taśma M jest nieskończona tylko w prawo (bo możemy „złożyć taśmę na pół" tj. indeksować $0, -1, 1, -2, \ldots$)

Mając takie założenia idea konstrukcji jest następująca:

- wiersze kafelkowania będą kolejnymi konfiguracjami maszyny M
- kafelek $(0, 0)$ wymusza aby pierwszy wiersz reprezentował konfigurację $q_0 \sqcup \sqcup \ldots$
- dajemy kafelki pozwalające kopiować „bezpieczne" fragmenty konfiguracji
- przejścia głowicy załatwiamy specjalnymi kafelkami
- do stanów końcowych (akceptujących/odrzucających) nie pasuje żaden kafelek tj. nie możemy nad nimi skonstruować kolejnego rzędu.

Ostatni podpunkt zapewnia nas, że jeśli M się zatrzymuje to kafelkowanie nie istnieje. Jeśli natomiast M się *nie* zatrzymuje to kafelkowanie generujemy przepisując kolejne konfiguracje działającej w nieskończoność maszyny.

□

5.5.3 CFL-INTERSECT

Rozdział 6

Złożoność obliczeniowa

6.1 Klasy złożoności

6.1.1 Definicje

Definicja 6.1.1. Mówimy, że maszyna M (deterministyczna lub nie) **działa w czasie** $T : \mathbb{N} \to \mathbb{N}$ jeśli dla każdej konfiguracji startowej q_0w każde obliczenie jest akceptujące lub odrzucające i ma długość co najwyżej $T(|w|)$

Definicja 6.1.2. Język L jest w klasie P (polynomial) jeśli istnieje wielomian p oraz Deterministyczna Maszyna Turinga działająca w czasie p taka, że $L(M) = L$

Definicja 6.1.3. Język L jest w klasie NP (nondeterministic polynomial) jeśli istnieje wielomian p oraz Niedeterministyczna Maszyna Turinga działająca w czasie p.

Definicja 6.1.4. $L \in \text{coNP} \iff \overline{L} \in \text{NP}$.

Definicja 6.1.5. $\text{PTIME} = \bigcup_{k=0}^{\infty} \text{TIME}(n^k)$

Definicja 6.1.6. $\text{NPTIME} = \bigcup_{k=0}^{\infty} \text{NTIME}(n^k)$

Definicja 6.1.7. Mówimy, że funkcja f jest **redukcją wielomianową** (Karpa) jeśli istnieje wielomian p oraz Maszyna Turinga obliczająca f w czasie p.

Jeśli istnieje redukcja wielomianowa z języka L_1 do języka L_2 to zapisujemy to jako $L_1 \leq_p L_2$

Definicja 6.1.8. Mówimy, że Maszyna Turinga M ma wyrocznię dla języka L, jeśli „ma ona dostęp" do zbioru L (w czasie stałym), tj. wpisuje jakieś słowo na specjalną taśmę, przechodzi do odpowiedniego stanu $q_?$ po czym na specjalnej taśmie wyjściowej w następnym kroku będzie miała odpowiedź TAK lub NIE, zależnie od tego czy dane słowo należy do L czy nie[1].

Definicja 6.1.9. Język L jest w klasie P^{NP}, jeśli istnieje taki wielomian p i taka deterministyczna Maszyna Turinga M z wyrocznią dla jakiegoś problemu z klasy NP taka, że $L(M) = L$ i M działa w czasie p.

Definicja 6.1.10. Język L jest w klasie NP^{NP}, jeśli istnieje taki wielomian p i taka niedeterministyczna Maszyna Turinga M z wyrocznią dla jakiegoś problemu z klasy NP taka, że $L(M) = L$ i M działa w czasie p.

Definicja 6.1.11. $L \in \text{coNP}^{\text{NP}} \iff \overline{L} \in \text{NP}^{\text{NP}}$.

[1] Jest to zasadniczo bardzo podobne do redukcji Turinga z nierozstrzygalności i również nazywamy to redukcją Turinga.

Definicja 6.1.12. Dla klasy problemów $C \subseteq R$ problem L jest **C-trudny** jeśli $\forall_{L' \in C} L' \leq_p L$

Definicja 6.1.13. Dla klasy problemów $C \subseteq R$ problem L jest **C-zupełny** jeśli jest C-trudny i $L \in C$

6.1.2 Zawierania

6.2 Redukcje

Lemat 6.2.1. Jeśli L_1 jest C-trudny i $L_1 \leq_p L_2$ to L_2 też jest C-trudny.

Dowód. Skoro L_1 jest C-trudny to znaczy, że dla każdego $L' \in C$ mamy redukcję f z L' do L_1. Mamy też redukcję g z L_1 do L_2.

Składając $g \circ f$ dostajemy redukcję z L' do L_2 co dowodzi, że L_2 jest trudniejszy niż dowolny $L'C$ □

Lemat 6.2.2. L jest C-trudny wtedy i tylko wtedy, gdy $\overline{L}$ jest coC-trudny.

Dowód. Dowodzimy równoważność w obie strony:

- $\Longrightarrow$

  Weźmy sobie $L' \in C$. Jako, że L' jest trudne, to $L' <_p L$. Wiemy wobec tego, że istnieje taka funkcja[2] f, obliczalna w czasie wielomianowym, że $x \in L' \iff f(x) \in L$.

  Równoważnie, $x \notin L' \iff f(x) \notin L$. No ale to z kolei oznacza, że $x \in \overline{L'} \iff f(x) \in \overline{L}$. Wobec tego, dla każdego problemu należącego do coC jest tak, że redukuje się on do $\overline{L}$, czyli $\overline{L}$ jest coC-trudny.

- $\Longleftarrow$ Centralnie piszemy to samo, ale w drugą stronę.

  Bierzemy sobie $L' \in coC$. Z racji faktu, że L' jest trudne mamy, że $L' <_p \overline{L}$. To oznacza, że istnieje funkcja f, taka że $x \in L' \iff f(x) \in \overline{L}$.

  Równoważnie, $x \notin L' \iff f(x) \notin \overline{L}$. Z tego wiemy, że $x \in \overline{L'} \iff f(x) \in L$. Wobec tego, każdy problem z C redukuje się do L, czyli L jest C-trudne.

□

6.3 Problemy trudne i zupełne

6.4 Twierdzenie Cooka-Levina

Twierdzenie 6.4.1 (Cook-Levin)**.** Problem SAT jest NP-zupełny

Dowód. Oczywiście SAT $\in$ NP bo możemy "zgadnąć" wartościowanie i sprawdzić że wartościuje formułę na 1.

Pokażemy, że SAT jest NP-trudny.

W tym celu musimy pokazać, że jeśli $L \in$ NP to istnieje redukcja z L do SAT. Skoro $L \in$ NP to istnieje niedeterministyczna Maszyna Turinga M, która go akceptuje.

[2] obliczalna w czasie wielomianowym bla bla bla

Intuicyjnie – będziemy śledzić obliczenie prowadzone przez M na zadanym słowie wejściowym w i zapisywać je jako formułę logiczną.

M działa w czasie $p(|x|)$ zatem możemy wyobrazić sobie tablicę o wymiarach $p(|x|) \times p(|x|)$, w której wiersze reprezentują kolejne konfiguracje maszyny.

Możemy skonstruować formułę logiczną która będzie opisywała jakie są poprawne przejścia w konfiguracjach maszyny.

W tym celu będziemy potrzebowali jakoś kodować za pomocą formuł boolowskich, gdzie znajduje się jaka wartość na taśmie. Definiujemy sobie zatem zmienną x_{ija}. Będziemy usiłowali osiągnąć jej taką semantykę, żeby mówiła ona, że w i-tym wierszu (czyli i-tej w kolejności konfiguracji), j-tej kolumnie (czyli j-tym symbolu w konfiguracji) znajduje się symbol a.

Przy takiej interpretacji chcemy jeszcze dodać warunki na konfigurację startową i końcową w taki sposób, aby istnienie wartościowania formuły na 1 było równoważne istnieniu obliczenia akceptującego.

W tym celu będą potrzebne nam cztery rzeczy; każdą z nich wymusimy oddzielną formułą logiczną, a ich koniunkcja posłuży nam do przeprowadzenia udanej symulacji w całości.

Zanim jednak do tego przejdziemy, umawiamy się, że jak nasza NTM wejdzie w stan akceptujący, to ma zdefiniowane przejście które nic nie robi i idzie po prostu w prawo (i dalej jest akceptujące). Dlaczego tak robimy? Bo będzie mniej dziwnych przypadków później.

Nietrudno dla dowolnej NTM stworzyć równoważną która spełnia ten warunek, więc to założenie nie powinno nikogo boleć.

6.4.1 Wartości zmiennych są spójne

Zauważamy, że semantyka zmiennych x bardzo szybko spadłaby z rowerka, jeśli istniałyby 2 takie zmienne, że $x_{ija} = 1 \wedge x_{ijb} = 1$ (bo to by znaczyło że w jednym miejscu na taśmie są 2 symbole) lub gdyby $\forall_a x_{ija} = 0$ (bo to by znaczyło, że w jakimś miejscu na taśmie nie ma **nic** (blank również jest symbolem).

Aby zapobiec takim shenanigansom, definiujemy sobie formułę φ_1:

$$\varphi_1 = \bigwedge_{\substack{1 \le i \le p(n) \\ 1 \le j \le p(n)}} A_{ij} \wedge B_{ij}$$

Gdzie A_{ij} oraz B_{ij} odpowiadają, odpowiednio, formule wymuszającej by chociaż jeden symbol znalazł się na taśmie oraz formule wymuszającej, by nie było sytuacji gdzie w jednym miejscu jest ponad jeden symbol. Zdefiniujmy je zatem:

$$A_{ij} = \bigvee_{a \in \Gamma} x_{ija}$$

A_{ij} wymaga zatem, by *któryś* symbol był zwartościowany jako „prawda" (jeśli formuła ma mieć szanse się zwartościować na „prawdę").

$$B_{ij} = \bigwedge_{\substack{a \in \Gamma \\ b \in \Gamma \\ a \neq b}} \neg(x_{ija} \wedge x_{ijb})$$

B_{ij} z kolei zabrania, by jakiekolwiek 2 x_{ij*} zwartościowały się na „tak".

Zauważamy, że obie te formuły są wielomianowe, a jako że rozmiar naszej tablicy jest również wielomianowy względem wejścia wszystko jest absolutnie spoko.

6.4.2 Symulacja rozpoczyna się poprawnie

Aby w ogóle rozpocząć symulację NTM, musimy „zainicjalizować" jej konfigurację startową. Jest to akurat dosyć proste – wiemy, że na pierwszym miejscu znajduje się stan startowy q_s (umawiamy się, że wierzymy że NTM której taśma rozszerza się tylko w jedną stronę jest tak samo mocna jak taka, której taśma rozszerza się w obie strony; dowód dla czytelnika czy coś), dalej znajduje się słowo w, a potem blanki (tak by długość wiersza wynosiła $p(|w|)$.

Innymi słowy, robimy formułę wymuszającą początkowe wartościowanie:

$$\varphi_2 = a_{1,1,q_s} \wedge a_{1,2,w_1} \wedge a_{1,2,w_2} \wedge \cdots \wedge a_{1,k,\sqcup} \wedge a_{1,k+1,\sqcup} \wedge \cdots \wedge a_{1,p(|w|),\sqcup}$$

Długość formuły φ_2 jest oczywiście wielomianowa względem w.

6.4.3 Spełniamy jeśli NTM przeszła do stanu akceptującego

Biorąc pod uwagę naszą wcześniejszą umowę (że jeśli NTM zaakceptuje, to jedyne przejście które ma, to takie które zostawia je w stanie akceptującym i przesuwa w prawo), aby sprawdzić czy NTM akceptuje możemy po prostu sprawdzić czy którykolwiek ze znaków w ostatnim wierszu jest stanem akceptującym. Definiujemy zatem formułę:

$$\varphi_3 = \bigvee_{q_f \in F} \bigvee_{1 \leq j \leq p(|w|)} x_{p(|w|),j,q_f}$$

Jeśli gdziekolwiek znajdziemy jakiś stan akceptujący, jesteśmy w domu; jak nie, to znaczy że dane obliczenie nie akceptuje słowa (więc to wartościowanie jest do bani).

6.4.4 Przejścia są wykonywane poprawnie

To jest najciekawsza część dowodu – nareszcie nadszedł ten moment, że musimy zaimplementować jakoś przejścia.

Zauważamy, że przejście głowicy powoduje jedynie lokalną zmianę konfiguracji – z konfiguracji na konfigurację zmienią się maksymalnie 3 symbole (stan może ulec zmianie, znak na prawo od wcześniej lokacji głowicy może się zmienic, oraz głowica może przesunąć się w lewo, zmieniając swoją kolejnosć z literką po lewej).

Musimy zatem oddać każde możliwe przejście z σ – to jest to miejsce, gdzie wiele dowodów się poddaje i zaczyna machać, ale nie my.

Będzie nam za niedługo potrzebna formuła o semantyce „wymuś, by symbole na koordynatach (i_1, j_1) oraz (i_2, j_2) były sobie równe":

$$E(i_1, j_1, i_2, j_2) = \bigwedge_{a \in \Gamma} ((x_{i_1,j_1,a} \wedge x_{i_2,j_2,a}) \vee (\neg x_{i_1,j_1,a} \wedge \neg x_{i_2,j_2,a}))$$

Zdefiniujmy sobie formułę „wymuszającą przejście w prawo dla stanu q i znaku a, na koordynatach i, j", którą nazwiemy $T_r(q, a, i, j)$[3]:

$$T_r(q,a,i,j) = \bigvee_{(q',a',1)\in\delta(q,a)} ((x_{i,j,q} \wedge x_{i,j+1,a}) \implies (E(i,j-1,i+1,j-1) \wedge x_{i+1,j,a'} \wedge x_{i+1,j+1,q'}))$$

Zauważmy, że ta semantyka nawet ma sens: jeśli pod współrzędnymi i, j jest stan q, to jeśli mamy przesunąć się prawo, to chcemy aby zmienna na lewo była niezmieniona, na jego miejsce wskoczyła zmodyfikowana zmienna (która wcześniej była po jego prawej) a on sam zmienił swój stan na nowy.

Bardzo analogicznie zrobimy dla przesunięcia w lewo:

$$T_l(q,a,i,j) = \bigvee_{(q',a',-1)\in\delta(q,a)} ((x_{i,j,q} \wedge x_{i,j+1,a}) \implies (x_{i+1,j-1,q'} \wedge E(i,j-1,i+1,j) \wedge x_{i+1,j+1,a'}))$$

Definiujemy teraz formułę odpowiedzialną za zebranie tego wszystkiego do kupy, by zapewnić by przejścia były poprawne:

$$C = \bigwedge_{\substack{q\in Q \\ a\in\Gamma \\ 1\le i\le p(|w|) \\ 1\le j\le p(|w|)}} T_r(q,a,i,j) \vee T_l(q,a,i,j)$$

C jest wielomianowa względem rozmiaru wejścia, bo rozmiar Q i Γ to stałe, więc dla każdego pola w tablicy (których jest wielomianowo wiele) dodajemy stale wiele formuł mających stałe długości.

Zauważmy jednak, że to nie koniec naszych przygód, a dopiero początek; musimy bowiem jeszcze wymyślić sprytny sposób, by w sytuacji gdzie wartości są poza tym lokalnym *miejscem niebezpiecznym* się bardzo elegancko przepisały.

Okazuje się, że możemy to nawet zakodować; będziemy potrzebować dodatkowego klocka, którym jest formuła logiczna sprawdzająca, czy coś **nie** jest stanem:

$$NQ(i,j) = \bigwedge_{q\in Q} \neg x_{ijq}$$

Jest to bardzo użyteczne, bo zauważam, że mogę „przepisać" dany znak z konfiguracji na konfigurację poniżej wtedy i tylko wtedy, gdy nie jest on stanem i nie ma stanu obok siebie (po lewej lub prawej). Jeśli jest stanem (lub ma go w pobliżu) to tym co ma znajdować się na konfiguracji poniżej zajęła się już formuła C.

Definiujemy więc formułę wymuszającą przepisanie symbolu z konfiguracji na konfigurację poniżej:

[3]Osoby zaniepokojone użyciem implikacji uspokajam, że $p \implies q \equiv \neg p \vee q$.

$$K(i,j) = (NQ(i,j-1) \wedge NQ(i,j) \wedge NQ(i,j+1)) \implies E(i,j,i+1,j)$$

Rozciągamy to na wszystkie możliwe współrzędne:

$$D = \bigwedge_{\substack{1 \le i \le p(|w|) \\ 1 \le j \le p(|w|)}} K(i,j)$$

D jest wielomianowa względem wejścia, bo $K(i,j)$ ma stałą długość, a z kolei tych formuł jest wielomianowo wiele (bo na każdą komórkę naszej gigatablicy przypada jedna).

I, koniec końców, otrzymujemy wyrażenie wymuszające poprawne przejścia:

$$\varphi_4 = C \wedge D$$

φ_4 również ma rozmiar wielomianowy, jako że jest koniunkcją dwóch formuł długości wielomianowej.

Proste, nie?

6.4.5 Pozbieranie do kupy

Teraz zbieramy to wszystko do kupy i otrzymujemy ostateczne gigawyrażenie:

$$\varphi = \varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_4$$

Jest ono spełnialne wtedy i tylko wtedy, gdy NTM akceptuje słowo w, a jego konstrukcja jest wielomianowa. Tym samym dowiedliśmy tw. Cooka-Levina. □

6.5 Twierdzenie Savitcha

Definicja 6.5.1. Definiujemy problem REACHABILITY jako:

- wejście: Nieskierowany graf $G = (V, E)$, wierzchołki v_1 i v_2.
- pytanie: Czy istnieje ścieżka z v_1 do v_2?

Twierdzenie 6.5.1. Problem REACHABILITY można rozwiązać w $\mathrm{SPACE}(\log^2(n))$.

Dowód. Konstruujemy algorytm, który sprawdza po kolei wszystkie możliwe wierzchołki „w połowie trasy" między v_1 a v_2; jest on rekurencyjny.

Złożoność pamięciowa wychodzi ładnie, bo głębokość wywołań rekurencyjnych jest jakaś logarytmiczna, i w każdym wywołaniu trzymamy liczniki logarytmicznej długości. Stąd też mamy $\log^2(n)$.

Wniosek 6.5.1. PSPACE = NPSPACE

Dowód. To, że PSPACE $\subset$ NPSPACE jest oczywiste. Skupmy się na drugim zawieraniu.

Rozważmy drzewo możliwych konfiguracji osiąganych przez niedeterministyczną Maszynę Turinga M. Jeśli zużywa ona wielomianowo wiele pamięci, to znaczy że istnieje taki wielomian p, że M zużywa maksymalnie $p(n)$ pamięci roboczej, gdzie n to rozmiar wejścia.

Zauważamy, że liczbę możliwych konfiguracji maszyny M możemy przeszacować przez $c^{p(n)}$, gdzie c jest jakąś stałą (np. $c = |\Gamma| + |Q|$). Zauważamy, że to co nas interesuje to to, czy z konfiguracji startowej możemy uzyskać konfigurację końcową. Centralnie jest to problem REACHABILITY, tyle że sąsiadów naszego grafu (czyli kolejne konfiguracje opisane funkcją przejścia) generujemy w locie.

Co zatem możemy zrobić, to ordynarnie generować po kolei wszystkie możliwe konfiguracje takie, że na taśmie znajduje się stan akceptujący i pytać, czy taka konfiguracja jest osiągalna z tej startowej. Zauważmy, że możemy reuse'ować pamięć, więc taka generacja kolejno wszystkich konfiguracji końcowych nas nie boli.

Innymi słowy, definiujemy sobie zbiór A:

$$A = \{w : \exists_{\alpha,\beta\in\Gamma}\exists_{q\in F} w = \alpha q \beta\}$$

Nasza deterministyczna MT będzie generować go w „locie" (nietrudno zobaczyć jak to można zrobić) i dla każdej otrzymanej teoretycznie możliwej konfiguracji końcowej pytać, czy da się ją osiągnąć z konfiguracji startowej.

Jako, że REACHABILITY zużywa deterministycznie $\log^2(n)$ pamięci, mamy że nasz algorytm będzie zużywać celem sprawdzenia osiągalności $\log^2(c^{p(n)}) = p(n)^2 \cdot \log^2(c)$, co jest dosyć mocno wielomianowe. Miejsce zużyte na trzymanie na jakichś taśmach konfiguracji startowej i końcowej też jest wielomianowe, więc cały algorytm jest wielomianowy. □

□

6.6 Twierdzenie Immermana-Szelepcsenyi'ego

W tej sekcji zajmiemy się twierdzeniem w pewnym sensie dualnym do tw. Savitcha pokazanego poprzednio.

Definicja 6.6.1. Definiujemy problem UNREACHABILITY jako:

- wejście: Nieskierowany graf $G = (V, E)$, wierzchołki v_1 i v_2.
- pytanie: Czy **nie** istnieje ścieżka z v_1 do v_2?

Twierdzenie 6.6.1 (Immerman-Szelepcsenyi)**.** UNREACHABILITY $\in$ NLOGSPACE

Dowód. Mamy do pokonania trzy przeszkody:

- Mamy logarytmicznie wiele pamięci – możemy przechowywać jedynie stałą liczbę liczników.
- Działamy na niedeterministycznej maszynie, musimy więc zapewnić że jeśli jakaś ścieżka obliczeń odpowie „TAK" to istotnie v_1 i v_2 nie są osiągalne.
- Mamy odpowiedzieć „TAK" jeśli ścieżki między v_1, v_2 *nie ma* czyli wszystkie możliwe spacery długości n nie osiągają v_2.

Chcielibyśmy utworzyć sobie niedeterministyczny pomocniczy algorytm

$$\text{reachable}(s, t, d, c)$$

który sprawdza, czy istnieje ścieżka długości d z s do t. Wartość c jest pomocniczą informacją, którą funkcja otrzymuje od wyroczni — wkrótce dowiemy się, jak ją tam zapakować. Na razie załóżmy, że magicznie znamy c i jest to *liczba* wierzchołków osiągalnych w d krokach z s.

De facto będzie to niedeterministyczna maszyna Turinga obliczająca funkcję boolowską. Ale co to tak naprawdę znaczy? Każda gałąź obliczeń musi zwrócić (czyli np. wypisać na taśmę wyjściową) jedną z trzech wartości:

- „TAK"
- „NIE"
- „NIE WIEM"

Któraś gałąź musi zwrócić coś innego niż „NIE WIEM", a te, które zwracają coś innego, muszą to zwrócić spójnie (czyli wszystkie „TAK" albo wszystkie „NIE"). Za odpowiedź maszyny przyjmujemy odpowiedź tych, które wiedzą.

Intuicyjnie, głównym problemem jest to, że nie możemy (tak jak dla REACHABILITY) zgadnąć spaceru i powiedzieć „TAK, ten spacer nie działa" bo być może źle zgadliśmy. Chcielibyśmy się upewnić, że obliczenia, które odpowiadają „NIE", faktycznie mają ku temu podstawy.

Przedstawimy teraz algorytm i przekonamy się, że spełnia on powyższe kryteria: Mamy tu dwa momenty, w których niedeterminizm rozdziela nasz program na niezależne, lecz spójne obliczenia. Skąd mamy spójność?

Jeśli poprawna odpowiedź to „TAK", to istnieje obliczenie, która *nie zignoruje* $v = t$ w kroku (A) i trafi na niego w kroku (B). Być może istnieje nawet wiele takich obliczeń i wszystkie odpowiedzą „TAK".

Skąd jednak wiemy, że żadne obliczenie nie odpowie wtedy „NIE"? A kiedy jest jedyna sytuacja, w której obliczenie odpowiada „NIE"? Wtedy, gdy $k = c$. A kiedy tak jest? Tylko wtedy, gdy było c wierzchołków, których obliczenie nie zignorowało w kroku (A) i do których dotarło w kroku (B), ale *żaden z nich* nie był t. Ale zaraz — przecież c to liczba wierzchołków osiągalnych z s w d krokach — wiemy to od wyroczni. Skoro zatem odwiedziliśmy *wszystkie* takie i *żaden z nich* nie był t, to po prostu t nie jest osiągalne z s w d krokach, czyli odpowiedź „NIE" jest słuszna i nikt inny nie mógł odpowiedzieć „TAK".

Ta obserwacja jest kluczowa dla działania algorytmu.

Nie wiem jak dalej :|.

□

6.7 Problem TAUTOLOGY

Definicja 6.7.1. Problem TAUTOLOGY definiujemy jako problem w którym:

- wejście: formuła rachunku zdań φ
- wyjście: Czy φ jest tautologią tj. czy każde wartościowanie wartościuje φ na 1 ?

Twierdzenie 6.7.1. TAUTOLOGY jest coNP-zupełny

Dowód. Pokażemy że $\overline{\text{TAUTOLOGY}}$ jest NP-zupełny.

Mamy zatem formułę φ i chcemy sprawdzić czy tautologią nie jest, czyli czy istnieje v takie, że $v(\varphi) = 0$.

Okazuje się, że $v(\neg\varphi) = 0 \iff v(\varphi) = 1$. Oznacza to, że problem $\overline{\text{TAUTOLOGY}}$ i SAT są sobie równoważne (wystarczy zanegować formułę na wejściu) – czyli $\overline{\text{TAUTOLOGY}}$ jest NP-zupełny czyli TAUTOLOGY jest coNP-zupełny.

□

6.8 Problem TQBF

6.9 Proste redukcje dla znanych problemów

6.9.1 Definicje

6.9.1.1 NAE-Sat (Not-All-Equal SAT)

1. IN: Formuła logiczna w 3-CNF
2. PYTANIE: Czy istnieje wartościowanie takie, że dla wszystkich klauzul prawdą jest, że w obrębie klauzuli znajduje się literał wartościowany na TAK jak i literał wartościowany na NIE?

6.9.1.2 EX2SAT

1. IN: Formuła logiczna φ w postaci

$$(x_1 \oplus y_1) \wedge (x_2 \oplus y_2) \wedge ... \wedge (x_n \oplus y_n)$$

   oraz K liczba naturalna zapisana binarnie.

2. PYTANIE: : Czy istnieje wartościowanie zmiennych dla φ, które spełnia dokładnie K klauzul w φ?

6.9.1.3 HDE (Homogenous Diophantine Equation)

Definicja 6.9.1. Jednorodne równanie Diofantyczne to równanie diofantyczne zdefiniowane przez wielomian, którego wszystkie niezerowe wyrazy mają ten sam stopień. Stopień wyrazu w równaniu Diofantycznym to suma wykładników zmiennych, które w nim występują – na przykład $x^5y^6z^1$ ma stopień 12.

Definicja 6.9.2. Definiujemy problem zero-one HDE następująco:

- IN: Jednorodne równanie diofantyczne stopnia 2, postaci $P(x_1, x_2, x_3, \ldots, x_n) = B$, gdzie $B \in \mathbb{N}$.

PYTANIE: Czy istnieje takie rozwiązanie tego równania, że wartości $x_1, x_2, \ldots, x_n \in \{0, 1\}$?

6.9.1.4 Bounded-Tiling

Definicja 6.9.3.

6.9.1.5 H-Coloring

Niech $H = (V_H, E_H)$ będzie grafem bez pętli. Problem H-COLORING jest następujący:

- WEJŚCIE: Graf symetryczny $G = (V_H, E_H)$ bez pętli
- PYTANIE: czy istnieje homomorfizm h z G w H, czyli

$$h \colon V_G \mapsto V_H$$
$$\forall_u \forall_v (u, v \in E_G \implies (h(u), h(v)) \in E_H)$$

H-COLORING jest NP-zupełny np. dla H będącego trójkątem — jest to wtedy 3-kolorowanie. Natomiast np. dla H będącego dowolną ścieżką ten problem jest w P.

Dowód. Trywialny. □

6.9.1.6 Factoring

- IN: Liczby naturalne N i M (Zapisane binarnie)
- OUT: Istnienie takiego K, że K dzieli N oraz $2 \leq \mathrm{K} \leq \mathrm{M}$
- COMPLEXITY: NP∩coNP

6.9.1.7 Inference

- IN: Dwie formuły Boolowskie φ_1, φ_2 nad tym samym zbiorem zmiennych
- PYTANIE: czy każde wartościowanie spełniające φ_1 jest również wartościowaniem spełniającym dla φ_2

6.9.1.8 Min-Inference

6.9.1.9 First-Sat

- IN: formuła φ nad Zmiennymi $X = \{x_1, ..., x_n\}$
- PYTANIE: czy φ ma minimalne względem porządku leksykograficznego spełniające wartościowanie $\alpha : X \to \{0, 1\}$ takie że $\alpha(x_i) = 1$?

6.9.1.10 Abduction

Definicja 6.9.4. Niech Y to będzie zbiór zmiennych boolowskich. Mówimy, że zbiór literałów U wyczerpuje zmienne Y, jeśli dla każdej zmiennej z Y zawiera tę zmienną *albo* jej zaprzeczenie.

Problem ABDUCTION jest następujący:

- WEJŚCIE: Spełnialna formuła boolowska φ nad zmiennymi $V, V' \subsetneq V$ oraz $x \in V \setminus V'$.
- PYTANIE: Czy istnieje zbiór literałów U wyczerpujący zmienne V' taki, że

$$\left(\varphi \wedge \bigwedge_{u \in U} u\right) \models x$$

Pokaż, że problem ABDUCTION jest w $\mathrm{NP}^{\mathrm{NP}}$.

6.9.1.11 Geography

Gra GEOGRAPHY rozgrywana jest przez dwóch graczy na grafie skierowanym $G = (V; E)$. W pierwszym ruchu gracz I koloruje pewien wierzchołek v w G, gracz II koloruje pewien wierzchołek v' taki, że $(v, v') \in E$. W kolejnych ruchach gracze kolorują kolejne niepokolorowane wierzchołki wyznaczając w G ścieżkę. Przegrywa gracz, który nie może wybrać żadnego niepokolorowanego jeszcze wierzchołka.

Problem GEOGRAPHY, to pytanie czy gracz I ma strategię wygrywającą tzn. wygrywa zawsze, niezależnie od wyborów II gracza

6.9.1.12 Regexp-Equiv

- IN: Dwa wyrażenia regularne, α i β
- OUT: Czy L(α) = L(β)?
- COMPLEXITY: PSPACE-complete

6.9.1.13 Quantified k-SAT

6.9.1.14

6.9.1.15