*Notatki z AiSD. Nr ??.* 23 maja 2013

# Wyszukiwanie wzorców

IIUWr. II rok informatyki.

## 1 Notacja

- $\Sigma$ - ustalony alfabet
- $T[1..n]$ i $P[1..m]$ - ciągi symboli z $\Sigma$
- $T$ nazywamy *tekstem* a $P$ - *wzorcem*
- Mówimy, że $P$ *występuje z przesunięciem* $s$ w tekście $T$ jeśli $0 \leq s \leq n-m$ oraz $T[s+1..s+m] = P[1..m]$.
- $w \sqsubset x$ - $w$ jest *prefiksem* $x$-a (tzn. $\exists_{y \in \Sigma^*}\ wy = x$)
- $w \sqsupset x$ - $w$ jest *sufiksem* $x$-a (tzn. $\exists_{y \in \Sigma^*}\ yw = x$)
- $P_k$ - $k$-elementowy prefiks $P[1..k]$ wzorca $P[1..m]$
- $T_k$ - $k$-elementowy prefiks $T[1..k]$ tekstu $T[1..m]$

**Fakt 1** *Niech $x, y$ i $z$ będą takie, że $x \sqsupset z$ i $y \sqsupset z$. Wówczas*

- $|x| \leq |y| \Rightarrow x \sqsupset y$,
- $|x| \geq |y| \Rightarrow y \sqsupset x$,
- $|x| = |y| \Rightarrow x = y$,

## 2 Definicja problemu

*Dane:* wzorzec $P[1..m]$ oraz tekst $T[1..n]$
*Zadanie:* znaleźć wszystkie wystąpienia $P$ w $T$ (tj. znaleźć wszystkie $s$ z przedziału $\langle 0, n-m \rangle$ takie, że $P \sqsupset T_{s+m}$).

## 3 Algorytmy

### 3.1 Algorytm naiwny

```
procedure Naive-string-matcher(T, P)
    n ← length(T)
    m ← length(P)
    for s ← 0 to n − m do
        if P[1..m] = T[s + 1..s + m]  then write("wzorzec występuje z przesunięciem", s)
```

Koszt: $\Theta((n-m+1)m)$ w najgorszym przypadku.

## 3.2 Algorytm Karpa-Rabina

IDEA:
Słowa nad $d$-literowym alfabetem $\Sigma$ traktujemy jako liczby $d$-arne. Jeśli $p$ oznacza liczbę odpowiadajcacą wzorcowi $P$, a $t_s$ - liczbę odpowiadającą $T[s+1..s+m+1]$ $(s = 0, ..n-m)$, to wzorzec występuje z przesunięciem $s$ iff $p = t_s$. Gdy $m$ jest duże, to $p$ oraz $t_i$ są duże i ich porównywanie jest kosztowne. Dlatego wybieramy liczbę $q$ (zwykle jest to liczba pierwsza) taką, że $dq$ mieści się w słowie maszynowym i liczby $p$ oraz $t_i$ obliczamy modulo $q$. Wówczas

(1) $p \neq t_s \Rightarrow P$ nie występuje w $T$ z przesunięciem $s$,
(2) $p = t_s \Rightarrow P$ może występować w $T$ z przesunięciem $s$.

```
procedure Karp − Rabin − matcher(T, P, d, q)
    n ← length(T)
    m ← length(P)
    h ← d^{m−1} mod q
    p ← 0; t_0 ← 0
    for i ← 1 to m do
       p ← (dp + P[i]) mod q
       t_0 ← (dt_0 + T[i]) mod q
    for s ← 0 to n − m do
        if p = t_s then
            if P[1..m] = T[s + 1..s + m] then write("wzorzec występuje z przesunięciem", s)
        if s < n − m then t_{s+1} ← (d(t_s − T[s + 1]h) + T[s + m + 1]) mod q
```

KOSZT: $\Theta((n-m+1)m)$ w najgorszym przypadku. Gdy wzorzec występuje w tekście niewiele razy oraz gdy $t_i$ przyjmują wartości $\{0, .., q-1\}$ z równym prawdopodobieństwem, to wybierając $q$ większe od $m$ koszt powyższej procedury można oszacować przez $O(m+n)$.
UWAGA: Algorytm ten łatwo uogólnia się na problem szukania wzorców dwuwymiarowych.

## 3.3 Wyszukiwanie wzorców automatami skończonymi.

### 3.3.1 Konstrukcja automatu

IDEA:
Dla danego wzorca $P$ skonstruujemy automat skończony $M_P$ o stanach ze zbioru $\{0, ..m\}$. Automat, czytając tekst $T$, będzie znajdować się w stanie $d$, jeśli ostatnich $d$ liter tekstu może rozpoczynać wzorzec i dla żadnego $e > d$, $e$ ostatnio wczytanych liter nie może rozpoczynać wzorca. W szczególności dojście do stanu $m$ będzie oznaczać, że $m$ ostatnio wczytanych liter tekstu tworzy wzorzec.

**Definicja 1** *Dla automatu skończonego $M = (Q, q_0, A, \Sigma, \delta)$, określamy funkcję $\phi : \Sigma^* \to Q$:*

$$\begin{aligned} \phi(\varepsilon) &= q_0 \\ \phi(wa) &= \delta(\phi(w), a), \end{aligned}$$

Innymi słowy $\phi(w)$ ="stan, w którym znajdzie się $M$ po przeczytaniu $w$".

**Definicja 2** *Dla wzorca $P$ definiujemy funkcję $\sigma : \Sigma^* \to \{0, \ldots, m\}$:*

$$\sigma(x) = \max\{k \mid P_k \sqsupset x\}$$

Czyli $\sigma(x)$ ="długość najdłuższego prefiksu $P$, który jest sufiksem $x$-a".

**Fakt 2** *(Własności funkcji $\sigma$)*

*(a)* $\sigma(x) = |P| \quad$ *iff* $\quad P \sqsupset x$

*(b)* $x \sqsupset y \quad \Rightarrow \quad \sigma(x) \leq \sigma(y)$

**Definicja 3** *(Automatu skończonego $M_P$ dla wzorca $P$)*

- *zbiór stanów:* $Q = \{0, 1, \ldots, m\}$,
- *stan początkowy:* $q_0 = 0$,
- *zbiór stanów końcowych:* $A = \{m\}$,
- *funkcja przejścia:* $\forall_{q \in Q, a \in \Sigma} \ \delta(q, a) = \sigma(P_q a)$.

### 3.3.2 Program symulujący automat $M_P$.

```
procedure Finite − automaton − matcher(T, δ, m)
    n ← length(T)
    q ← 0
    for i ← 1 to n do
        q ← δ(q, T[i])
        if q = m then write( "wzorzec występuje z przesunięciem" , i − m)
```

Koszt procedury: $O(n)$ (koszt ten nie obejmuje kosztu obliczenia funkcji $\delta$).

### 3.3.3 Analiza poprawności

Poniższe lematy i twierdzenie pokazują, że jeśli po wczytaniu $i$-tej litery tekstu $M_P$ jest w stanie $q$ ($=\phi(T_i)$), to $q$ jest długością najdłuższego sufiksu $T_i$, który jest prefiksem $P$ ($= \sigma(T_i)$). Ponieważ $\sigma(T_i) = m$ iff $P \sqsupset T_i$, więc stan akceptujący będzie osiągany wtedy i tylko wtedy, gdy $m$ ostatnio przeczytanych znaków tworzy wzorzec.

- **Lemat 1** $\forall_{x \in \Sigma^*} \forall_{a \in \Sigma} \quad \sigma(xa) \leq \sigma(x) + 1$,
- **Lemat 2** $\forall_{x \in \Sigma^*} \forall_{a \in \Sigma} \quad q = \sigma(x) \quad \Rightarrow \quad \sigma(xa) = \sigma(P_q a)$
- **Twierdzenie 1** $\forall_{i=0,1,\ldots,n} \quad \phi(T_i) = \sigma(T_i)$.

### 3.3.4 Obliczanie funkcji $\delta$

- Sposób naiwny.

```
procedure Compute − Transition − Function(P, Σ)
    m ← length(P)
    for q ← 0 to m do
        for each a ∈ Σ do
            k ← min(m + 1, q + 2)
            repeat k ← k − 1 until P_k ⊐ P_q a
            δ(q, a) ← k
    return δ
```

  Koszt: $O(m^3 \, |\Sigma|)$

- Sposób zdecydowanie mniej naiwny (będzie przedmiotem ćwiczeń).

  Wykorzystuje funkcję prefiksową, którą zdefiniujemy opisując algorytm Knutha-Morrisa-Pratta. Czas jego działania wynosi $O(m|\Sigma|)$.

## 3.4 Algorytm Knutha-Morrisa-Pratta.

### 3.4.1 Idea

Zasada podobna jak poprzednio: po przeczytaniu $T_i$ chcemy wiedzieć jak długi prefiks $P$ jest sufiksem $T_i$. Załóżmy, że długość tego prefiksu wynosi $k$. Jeśli $T[i+1] = P[k+1]$, to wiemy, że teraz ta długość wynosi $k+1$. Gorzej jeśli $T[i+1] \neq P[k+1]$. Funkcja $\delta$ pozwalała nam tę dlugość określić w jednym kroku. Pociągało to jednak za sobą konieczność wstępnego obliczenia wartości $\delta$ dla wszystkich par $(k, a)$. To jest kosztowne! Teraz unikamy tego, pozwalając, by algorytm poświęcił więcej czasu na określenie długości prefiksu w trakcie czytania tekstu. Algorytm korzysta przy tym z pomocniczej funkcji $\pi$, którą oblicza wstępnie na podstawie wzorca w czasie $O(m)$.

**Definicja 4** *Dla wzorca $P$ definiujemy funkcję prefiksową $\pi : \{1, .., m\} \to \{0, .., m-1\}$*

$$\pi(q) \quad = \quad \max\{k \mid k < q \ i \ P_k \sqsupset P_q\}$$

KOMENTARZ: W sytuacji gdy $k$ ostatnich znaków tekstu tworzy prefiks $P$, a kolejny znak tekstu jest niezgodny z $k+1$-szym znakiem $P$, algorytm może sprawdzać czy znak ten jest zgodny z krótszymi prefiksami $P$, będącymi jednocześnie sufiksami wczytanego tekstu. Jako kandydatów na te prefiksy algorytm próbuje te prefiksy wzorca, które są sufiksami $P_k$. O tym, które są to prefiksy mówi funkcja $\pi$.

### 3.4.2 Algorytm

```
procedure KMP − Matcher(T, P)
    n ← length(T);  m ← length(P)
    π ← Compute − Prefix − Function(P)
    q ← 0
    for i ← 1 to n do
        while q > 0 and P[q + 1] ≠ T[i] do q ← π(q)
        if P[q + 1] = T[i] then q ← q + 1
        if q = m then write( "wzorzec występuje z przesunięciem", i − m)
                      q ← π(q)
```

### 3.4.3 Obliczanie funkcji prefiksowej

```
procedure Compute − Prefix − Function(P)
    m ← length(P)
    π(1) ← 0;  k ← 0
    for q ← 2 to m do
        while k > 0 and P[k + 1] ≠ P[q] do k ← π(k)
        if P[k + 1] = P[q] then k ← k + 1
            π(q) ← k
    return π
```

KOSZT: Procedura $Compute - Prefix - Function$ działa w czasie $O(m)$, a procedura $KMP - Matcher$ w czasie $O(n+m)$.

## 3.5 Algorytm Boyera-Moore'a

IDEA:

Metoda podobna do metody naiwnej: sprawdzamy kolejne przesunięcia $s$, ale dla danego $s$ tekst sprawdzamy począwszy od końca wzorca. Gdy napotkamy niezgodność korzystamy z dwóch heurystyk do zwiększenia $s$ (stosujemy tę, która proponuje większe przesunięcie):

- heurystyka "zły znak",
- heurystyka "dobry sufiks".

### 3.5.1 Heurystyka "zły znak"

Jeśli niezgodność wystąpiła dla $P[j] \neq T[s+j]$ $(1 \leq j \leq m)$, to niech

$$k = \begin{cases} \max\{z \mid P[z] = T[s+j]\} & \text{jeśli takie } z \text{ istnieje,} \\ 0 & \text{w p.p.} \end{cases}$$

Jeśli $k = 0$ lub $k < j$, to ta heurystyka proponuje przesunąć $s$ o $j - k$ znaków. Gdy $k > j$, to heurystyka nic nie proponuje.

### 3.5.2 Heurystyka "dobry sufiks"

**Definicja 5** *Mówimy, że $Q$ jest* podobne do $R$ *(i piszemy $Q \sim R$) iff $Q \sqsupset R$ lub $R \sqsupset Q$.*

Heurystyka "dobry sufiks" mówi, że gdy napotkamy niezgodność $P[j] \neq T[s+j]$ $(1 \leq j \leq m)$, to $s$ możemy zwiększyć o $m -$ $\max\{k \mid 0 \leq k < m \;\&\; P[j+1..m] \sim P_k\}$.

## 3.6 Algorytm Shift–AND

Idea algorytmu:

- W trakcie czytania tekstu pamiętamy informację o wszystkich prefiksach wzorca, które są sufiksami przeczytanego fragmentu tekstu.
- Algorytm ten przeznaczony jest do wyszukiwania krótkich wzorców, więc powyższa informacja może być przechowywana w jednym słowie maszynowym i w prosty sposób, kilkoma rozkazami, uaktualniana po wczytania kolejnego znaku.

Niech $C_j[0..m]$ będzie wektorem charakterystycznym zbioru prefiksów wzorca, które są sufiksami $t_1...t_j$, tj. $C_j[k] = true$ iff $P_k \sqsupset T_j$.

Obserwacje:

O1. Wektor $C_j$ można w prosty sposób wyznaczyć na podstawie wektora $C_{j-1}$, wzorca oraz $j$-tego znaku tekstu.

Mamy bowiem:

$$C_j[k] = \begin{cases} true & \text{dla } k = 0 \\ C_{j-1}[k-1] \wedge (p_k = t_j) & \text{dla } k > 0 \end{cases}$$

O2. Wzorzec występuje z przesunięciem $j - m$ wtedy i tylko wtedy, gdy $C_j[m] = true$.

Uwagi implementacyjne:

- Jeśli wzorzec jest krótki ($m <$ długość słowa maszynowego), do uaktualnienia wektora charakterystycznego możemy wykorzystać długie operacje logiczne. W tym celu dla każdej litery $d$ alfabetu tworzymy wektor $R_d$ taki, że $R_d[i] \equiv (p_i = d)$. Wówczas

$$C_j = Shift(C_{j-1})\ AND\ R_{p_j},$$

gdzie operacja $Shift$ oznacza przesunięcie w prawo o jeden bit z ustawieniem skrajnie lewego bitu na 1.

- Wystarczy pamiętać jeden (bieżący) wektor charakterystyczny i uaktualniać go po każdym przeczytanym znaku.

# 4 Algorytm Karpa–Millera–Rosenberga (KMR)

Idea algorytmu:

- Niech $w = PT$, a więc $w$ jest konkatenacją wzorca $P$ i tekstu $T$.
- Numerujemy wszystkie podsłowa słowa $w$ o długości $m$ w jednoznaczny sposób, tj. taki, że takie same podsłowa otrzymują ten sam numer, a różne podsłowa - różne numery.
- Wypisujemy wszystkie pozycje większe od $m$, na których zaczynają się podsłowa o takim samym numerze co podsłowo zaczynające się na pozycji 1 (a więc wzorzec).

Numerowanie podsłów

- Do numerowania wykorzystujemy kolejne liczby naturalne. W ten sposób zawsze będziemy mieli do czynienia z numerami nie większymi od $n$ (bo różnych podsłów danej długości jest nie więcej niż pozycji, na których mogą się one zaczynać).
- Startujemy od ponumerowania podsłów długości 1. W tym celu sortujemy w czasie liniowym litery występujące w słowie.
- Jeśli mamy ustaloną numerację słów długości $k$, możemy w prosty sposób znaleźć numerację podsłów długości $k'$ dla dowolnego $k' \in \{k+1, \ldots, 2k\}$:
  - Dla każdego $i = 1, \ldots, |PT| - k'$ tworzymy parę $\langle nr_k(i), nr_k(i + k' - k + 1)\rangle$, gdzie $nr_s(j)$ jest numerem $s$-literowego podsłowa zaczynającego się od pozycji $j$ (w obliczonej przez nas numeracji podsłów $s$-literowych).
  - Sortujemy leksykograficznie utworzone pary. Przeglądając ciąg par z lewa na prawo nadajemy im numery = " liczba różnych par na lewo".

Przykład
Załóżmy, że ponumerowaliśmy podsłowa 2 literowe w słowie $w = bbaabbaaaabbaa$ w następujący sposób:

| Pozycja | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Podsłowo | *bb* | *ba* | *aa* | *ab* | *bb* | *ba* | *aa* | *aa* | *aa* | *ab* | *bb* | *ba* | *aa* |
| Numer | 4 | 3 | 1 | 2 | 4 | 3 | 1 | 1 | 1 | 2 | 4 | 3 | 1 |

Tworząc numerację podsłów 4 literowych przypisujemy kolejnym pozycjom słowa $w$ następujące pary:

| Pozycja | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Podsłowo | *bbaa* | *baab* | *aabb* | *abba* | *bbaa* | *baaa* | *aaaa* | *aaab* | *aabb* | *abba* | *bbaa* |
| Para | 4,1 | 3,2 | 1,4 | 2,3 | 4,1 | 3,1 | 1,1 | 1,2 | 1,4 | 2,3 | 4,1 |

Po posortowaniu par otrzymujemy ciąg:

$$(1,1),\ (1,2),\ (1,4),\ (1,4),\ (2,3),\ (2,3),\ (3,1),\ (3,2),\ (4,1),\ (4,1),\ (4,1),$$

co umożliwia nam łatwe nadanie numerów parom:

| Para | (1,1) | (1,2) | (1,4) | (2,3) | (3,1) | (3,2) | (4,1) |
|---|---|---|---|---|---|---|---|
| Numer | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

i przypisanie ich podsłowom z kolejnym pozycji słowa $w$:

| Pozycja | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Podsłowo | *bbaa* | *baab* | *aabb* | *abba* | *bbaa* | *baaa* | *aaaa* | *aaab* | *aabb* | *abba* | *bbaa* |
| Numer | 7 | 6 | 3 | 4 | 7 | 5 | 1 | 2 | 3 | 4 | 3 |

□

**Fakt 3** *Algorytm KMR działa w czasie* $O(n \log n)$.

Dowód: Chcąc znaleźć numerację słów $m$ literowych wystarczy obliczyć numerację dla $\lceil \log m \rceil$ różnych długości. Obliczenie numeracji dla każdej z długości może być wykonane w czasie liniowym. □

Uwaga: Algorytm KMR może być zastosowany do wielu problemów związanych z wyszukiwaniem takich samych podsłów, w szczególności do problemu znajdowania najdłuższego powtarzającego się podsłowa.

# 5 Algorytm Fishera–Patersona

Algorytm ten służy do wyszukiwania wzorców w tekście, w których (zarówno we wzorcu jak i w tekście) mogą znajdować się nieznaczące znaki. Znaki takie są zgodne ze wszystkimi znakami alfabetu.

Przykładowo, jeśli przez $\diamond$ oznaczymy znak nieznaczący, to wzorzec $P = a \diamond a$ występuje dwukrotnie w tekście $T = babab \diamond$ (z przesunięciami 1 i 3).

**Fakt 4** *Każdy algorytm wyszukujący wzorców w tekście, który używa znaków wzorca i tekstu jedynie w porównaniach* $\equiv$ *ma złożoność* $\Omega(nm)$.

Uzasadnienie.

Niech $T = \diamond^n$ i $P = \diamond^m$. Poprawnie działający algorytm (nazwijmy go $A$) wypisze da tych danych $0, 1, \ldots, n - m$. Załóżmy, że taki algorytm nie wykonuje porównania znaków $T[i]$ i $P[j]$. Niech $a$ i $b$ będą różnymi znakami alfabetu i niech $T'$ będzie tekstem powstałym przez wstawienie $a$ na $i$-tą pozycję tekstu $T$ a $P'$ będzie wzorcem powstałym przez wstawienie $b$ na $j$-tą pozycję wzorca.

Algorytm $A$ działając na $T'$ i $P'$ wykona te same porównania $\equiv$ co na danych $T$ i $P$. W szczególności wypisze taki sam wynik. □

Idea rozwiązania.

Definiujemy *iloczyn tekstowy* dwóch słów $u = u_0 \ldots u_n$ i $v = v_0 \ldots v_m$ jako słowo $w = w_0 \ldots w_{n+m-1}$ takie, że $\forall_{k=0,\ldots,n+m-1}$:

$$w_k = \bigwedge_{i,j:i+j=k} u_i \equiv v_j$$