

WYSZUKIWANIE WZORCÓW

IIUWr. II rok informatyki.

1 Definicja problemu i notacja

Definicja 1 Niech w i x będą słowami nad alfabetem Σ . Mówimy, że

- w jest prefiksem x , jeśli istnieje $y \in \Sigma^*$ takie, że $wy = x$,
- w jest sufiksem x , jeśli istnieje $y \in \Sigma^*$ takie, że $yw = x$.

OZNACZENIA:

- $w \sqsubset x$ - w jest prefiksem x -a
- $w \sqsupset x$ - w jest sufiksem x -a

Definicja 2 Mówimy, że słowo P występuje w słowie T z przesunięciem s , jeśli istnieje słowo y o długości s takie, że $yP \sqsubset T$.

Problem wyszukiwania wzorca definiowany jest na wiele sposobów. My będziemy zainteresowani następującą jego wersją:

Definicja 3 (PROBLEM WYSZUKIWANIA WZORCA)

Dane: słowa P i T ; nazywamy je odpowiednio wzorcem i tekstem
 Zadanie: znaleźć wszystkie wystąpienia P w T
 (tj. znaleźć wszystkie przesunięcia, z którymi P występuje w T).

Tradycyjnie długość P oznaczana jest przez m , a długość T przez n . Ponadto będziemy stosować następujące oznaczenia:

- x_i - i -ta litera słowa X (w szczególności: p_i oznacza i -tą literę wzorca P a t_i - i -tą literę tekstu T),
- X_k - k literowy prefiks słowa X , tj. $X_k = x_1 \dots x_k$.

2 Algorytmy

2.1 Algorytm naiwny

Algorytm naiwny polega na sprawdzeniu występowania wzorca ze wszystkimi kolejnymi przesunięciami. Dla każdego przesunięcia sprawdzamy zgodność wzorca z tekstem literka po literce.

```

procedure AlgorytmNaiwny( $T, P$ )
  for  $s \leftarrow 0$  to  $n - m$  do
     $i \leftarrow 1$ 
    while ( $i \leq m$  and  $p_i = t_{s+i}$ ) do  $i \leftarrow i + 1$ 
    if ( $i = m$ ) write("wzorec występuje z przesunięciem",  $s$ )
  
```

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

Narzucające się usprawnienia algorytmu naiwnego możemy podzielić z grubsza na dwie grupy:

- efektywniejsze sprawdzanie występowania wzorca dla danego przesunięcia,
- wyeliminowanie złych przesunięć.

Doprowadziły one do powstania wielu, znacznie efektywniejszych, algorytmów.

2.2 Algorytm Karpa-Rabina

IDEA:

Słowa nad d -literowym alfabetem Σ traktujemy jako liczby d -arne. Jeśli p oznacza liczbę odpowiadającą wzorcowi P , a t_s - liczbę odpowiadającą $T[s + 1..s + m + 1]$ ($s = 0, \dots, n - m$), to wzorec 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( $T, P, d, q$ )
   $n \leftarrow \text{length}(T)$ 
   $m \leftarrow \text{length}(P)$ 
   $h \leftarrow d^{m-1} \bmod q$ 
   $p \leftarrow 0$ ;  $t_0 \leftarrow 0$ 
  for  $i \leftarrow 1$  to  $m$  do
     $p \leftarrow (dp + P[i]) \bmod q$ 
     $t_0 \leftarrow (dt_0 + T[i]) \bmod q$ 
  for  $s \leftarrow 0$  to  $n - m$  do
    if  $p = t_s$  then
      if  $P[1..m] = T[s + 1..s + m]$  then write("wzorec występuje z przesunięciem",  $s$ )
    if  $s < n - m$  then  $t_{s+1} \leftarrow (d(t_s - T[s + 1]h) + T[s + m + 1]) \bmod q$ 

```

KOSZT: $\Theta((n - m + 1)m)$ w najgorszym przypadku. Gdy wzorec występuje w tekście niewiele razy oraz gdy t_i przyjmują wartości $\{0, \dots, q - 1\}$ 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.

2.3 Wyszukiwanie wzorców automatami skończonymi.

2.3.1 Konstrukcja automatu

IDEA:

Dla danego wzorca P skonstruujemy automat skończony M_P o stanach ze zbioru $\{0, \dots, m\}$. Automat, czytając tekst T , będzie znajdował się w stanie d , jeśli ostatnich d liter tekstu może rozpoczynać wzorec 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 wzorec.

Definicja 4 Dla automatu skończonego $M = (Q, q_0, A, \Sigma, \delta)$, określamy funkcję $\phi : \Sigma^* \rightarrow Q$:

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

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

Definicja 5 Dla wzorca P definiujemy funkcję $\sigma : \Sigma^* \rightarrow \{0, \dots, m\}$:

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

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

Fakt 1 (Własności funkcji σ)

- (a) $\sigma(x) = |P|$ iff $P \sqsupset x$
- (b) $x \sqsupset y \Rightarrow \sigma(x) \leq \sigma(y)$

Definicja 6 (Automatu skończonego M_P dla wzorca P)

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

2.3.2 Program symulujący automat M_P .

```

procedure Finite – automaton – matcher( $T, \delta, m$ )
   $n \leftarrow \text{length}(T)$ 
   $q \leftarrow 0$ 
  for  $i \leftarrow 1$  to  $n$  do
     $q \leftarrow \delta(q, T[i])$ 
    if  $q = m$  then write( "wzorzec występuje z przesunięciem" ,  $i - m$ )

```

KOSZT PROCEDURY: $O(n)$ (koszt ten nie obejmuje kosztu obliczenia funkcji δ).

2.3.3 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** $\forall x \in \Sigma^* \forall a \in \Sigma \quad q = \sigma(x) \Rightarrow \sigma(xa) = \sigma(P_q a)$
- **Twierdzenie 1** $\forall i=0,1,\dots,n \quad \phi(T_i) = \sigma(T_i)$.

2.3.4 Obliczanie funkcji δ

- Sposób naiwny.

```

procedure Compute – Transition – Function( $P, \Sigma$ )
   $m \leftarrow \text{length}(P)$ 
  for  $q \leftarrow 0$  to  $m$  do
    for each  $a \in \Sigma$  do
       $k \leftarrow \min(m + 1, q + 2)$ 
      repeat  $k \leftarrow k - 1$  until  $P_k \sqsupset P_q a$ 
       $\delta(q, a) \leftarrow k$ 
  return  $\delta$ 

```

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 $O(m|\Sigma|)$.

2.4 Algorytm Knutha-Morrisa-Pratta.

2.4.1 Idea

Modyfikujemy algorytm naiwny tak, by przesunięcia wzorca dokonywać w oparciu o informację o przeczytanym fragmencie tekstu.

PRZYKŁAD.

Szukając algorytmem naiwnym wzorca $P = aaabaabab$ w tekście $T = aaabaaa....$ napotykamy niezgodność w trakcie sprawdzania siódmego znaku wzorca.

W takiej sytuacji nie ma sensu sprawdzać, czy wzorzec występuje z przesunięciem 1. Gdyby bowiem wzorzec miał występować z takim przesunięciem, to sześcioliterowy prefiks wzorca (tj. $aaabaa$) musiałby być sufiksem przeczytanego fragmentu tekstu, czyli słowa $aaabaaa$. Z tego samego powodu przesunięcia 2 i 3 nie są sensowne.

Określając sensowne przesunięcia możemy brać pod uwagę:

- najdłuższy, pasujący do momentu wystąpienia niezgodności, prefiks wzorca,
- literkę tekstu powodującą niezgodność,
- literkę wzorca powodującą niezgodność.

Im więcej z tych informacji uwzględnimy, tym większą mamy szansę na wyeliminowanie niedobrych przesunięć. Z drugiej jednak strony może to się odbyć kosztem większego czasu przetwarzania. \square

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 δ pozwalała nam tę długość określić w jednym kroku. Pociągało to jednak za sobą konieczność wstępnego obliczenia wartości δ 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 π , którą oblicza wstępnie na podstawie wzorca w czasie $O(m)$.

Definicja 7 Dla wzorca P definiujemy funkcję prefiksową $\pi : \{1, \dots, m\} \rightarrow \{0, \dots, m-1\}$

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

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 π .

2.4.2 Algorytm

```
procedure KMP - Matcher( $T, P$ )
   $n \leftarrow \text{length}(T)$ ;  $m \leftarrow \text{length}(P)$ 
   $\pi \leftarrow \text{Compute - Prefix - Function}(P)$ 
   $q \leftarrow 0$ 
  for  $i \leftarrow 1$  to  $n$  do
    while  $q > 0$  and  $P[q+1] \neq T[i]$  do  $q \leftarrow \pi(q)$ 
    if  $P[q+1] = T[i]$  then  $q \leftarrow q+1$ 
    if  $q = m$  then write( "wzorzec występuje z przesunięciem",  $i-m$ )
       $q \leftarrow \pi(q)$ 
```

2.4.3 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 $O(n + m)$.

2.5 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”.

2.5.1 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.

2.5.2 Heurystyka ”dobry sufiks”

Definicja 8 Mówimy, że Q jest podobne do R (i piszemy $Q \sim R$) iff $Q \sqsubset R$ lub $R \sqsubset Q$.

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

2.6 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 $t_1 \dots t_j$, tj. $C_j[k] \equiv (P_k \sqsubset T_j)$.

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 ($m < \text{długość słowa maszynowego}$), 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

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

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 przeczytaniu znaku.

3 Algorytm Karpa–Millera–Rosenberga (KMR)

IDEA ALGORYTMU:

- Niech $w = PT$, a więc w jest konkatenacją wzorca P i tekstu T .
- Numerujemy wszystkie podśłowa słowa w o długości m w jednoznaczny sposób, tj. taki, że takie same podśłowa otrzymują ten sam numer, a różne podśłowa - różne numery.
- Wypisujemy wszystkie pozycje większe od m , na których zaczynają się podśłowa o takim samym numerze co podśłowo 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 podśłów danej długości jest nie więcej niż pozycji, na których mogą się one zaczynać).
- Startujemy od ponumerowania podśłów 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 podśłów długości k' dla dowolnego $k' \in \{k+1, \dots, 2k\}$:
 - Dla każdego $i = 1, \dots, |PT| - k'$ tworzymy parę $\langle nr_k(i), nr_k(i + k' - k + 1) \rangle$, gdzie $nr_s(j)$ jest numerem s -literowego podśłowa zaczynającego się od pozycji j (w obliczonej przez nas numeracji podśłów s -literowych).
 - Sortujemy leksykograficznie utworzone pary. Przeglądając ciąg par z lewa na prawo nadajemy im numery = "liczba różnych par na lewo".

| | | | | | | | | | | | | | |
|----------|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Pozycja | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| Podślowo | 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 |

| | | | | | | | | | | | |
|----------|------|------|------|------|------|------|------|------|------|------|------|
| Pozycja | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| Podślowo | 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 |

PRZYKŁAD

Założmy, że ponumerowaliśmy podśłowa 2 literowe w słowie $w = bbaabbaaaabbaa$ w następujący sposób:

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

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 podśłowom z kolejnym pozycji słowa w :

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

□

Fakt 2 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 podśłów, w szczególności do problemu znajdowania najdłuższego powtarzającego się podśłowa.