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 ← 0 to n − m do
        i ← 1
        while (i ≤ m and p_i = t_{s+i}) do i ← i + 1
        if (i = m) write(”wzorzec 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:

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

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. □

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 $T[s+1..s+m]$ $(s = 0,..n-m)$, 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; t0 ← 0
   for i ← 1 to m do
      p ← (dp + P[i]) mod q
      t0 ← (dt0 + T[i]) mod q
   for s ← 0 to n − m do
      if p = ts then
         if P[1..m] = T[s + 1..s + m] then write("wzorzec występuje z przesunięciem", s)
      if s < n − m then ts+1 ← (d(ts − 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.

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 $\{0,..m\}$. 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 4 *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 5 *Dla wzorca P definiujemy funkcję $\sigma : \Sigma^* \to \{0, \ldots, m\}$:*

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

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

Fakt 1 *(Własności funkcji σ)*

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

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

Definicja 6 *(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:* $\forall_{q \in Q, a \in \Sigma} \quad \delta(q, a) = \sigma(P_q a)$.

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

2.3.4 Obliczanie funkcji δ

- 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 $O(m|\Sigma|)$.

2.4 Algorytm Knutha-Morrisa-Pratta.

2.4.1 Idea

Podobna jak poprzednio: po przeczytaniu T_i chcemy wiedzieć jaki najdłuższy prefiks P jest sufiksem T_i. Załóżmy, że długość tego prefiksu wynosi k. Jeśli $T[i+1] = P[k+1]$, to wiemy, że po przeczytaniu T_{i+1} 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 $\pi : \{1, .., m\} \to \{0, .., m-1\}$*

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

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 π.

2.4.2 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)
```

Fakt 2 *Algorytm $KMP{-}Matcher$ w czasie $O(n+$"czas działania procedury Compute-Prefix-Function")*.

UZASADNIENIE. W każdej iteracji pętli **for** wartość zmiennej q zwiększa się o nie więcej niż 1 lub maleje (w pętli **while**), ale nigdy nie jest ujemna.

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
```

Fakt 3 *Procedura $Compute-Prefix-Function$ działa w czasie $O(m)$.*

UZASADNIENIE. Podobne jak dla algorytmy $KMP-Matcher$. Tym razem obserwujemy zmiany wartości zmiennej k.

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 $Q \sqsupset R$ 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 $m - \max\{k \mid 0 \leq k < m \;\&\; P[j+1..m] \sim P_k\}$.