Wykład 12 i 13

Algorytmy tekstowe

Teksty mogą być reprezentowane za pomocą:

List

lub

Tablic

Problem wyszukiwania wzorca: WW

Dany wzorzec **x** i tekst **y:**

|x| = m, |y| = n i n >= m (n dużo większe niż y)

Znaleźć wszystkie wystąpienia wzorca **x** w **y**.

Oznaczenia:

z[i..j] = z[i]z[i+1]…z[j]

gdy **y**[i..i+m-1] = **x**, to **x** występuje w **y** na pozycji i.

Algorytm N (naiwny)

```
begin
    i := 1;
    while i <= n – m + 1 do
    begin
        if x[1..m] = y[i..i + m -1] then write(i);
        i := i +1;
    end;
end;
```

Algorytm N (naiwny – pełna wersja)

```
begin
   i := 1;
   while i <= n – m + 1 do
   begin
      j := 0; while x[j+1] = y[i+j] do  j := j + 1;
      if j = m then write(i);
      i := i +1; {przesunięcie równe 1}
   end;
end;
```

Złożoność pesymistyczna – $O(n^2)$
Złożoność oczekiwania – $O(n)$

Algorytm KMP (Knutha-Morrisa-Pratta)

Przesunięcie większe niż 1.

Korzystamy z informacji o wartości j, obliczonej w poprzednim kroku algorytmu.

P[j] = max { 0 <= k < j | x[1..k] jest sufiksem x[1..j]}

P[0] = P[1] = 0

przesunięcie(j) = max(1,j – P[j])

Algorytm KMP

```
begin
    i := 1; j := 0;
    while i <= n – m + 1 do
    {x[1..P[j]] = x[j-P[j]+1..j] = y[i..i+P[j]-1]}
    begin
        j := P[j];
        while x[j+1] = y[i+j] do  j := j + 1;
        if j = m then write(i);
        i := i + przesunięcie(j); {przesunięcie >= 1}
    end
end;
```

Złożoność pesymistyczna – O(n)
Złożoność oczekiwania – O(n)

Obliczanie tablicy P

{Algorytm obliczania tablicy P)

P[j] = P^i[j-1] + 1 dla pewnego i, jeśli istnieje
P[j] = 0 w przeciwnym wypadku

```
begin
   P[0] := P[1] := 0;
   t :=0;
   for j := 2 to m do
   begin {obliczamy wartość P[j]}
            {t = P[j - 1]}
      while (t > 0) and (x[t + 1] < > x[j]) do t := P[t];
      if x[t + 1] = x[j] then t := t + 1;
      P[j] := t;
   end;
end;
```

Złożoność – O(m)

Wersja oryginalna KMP

Zamiast tablicy P używa się bardziej skomplikowanej do obliczenia tablicy NEXT, w której:

NEXT[j] jest maksymalną długością k właściwego sufiksu słowa x[1..j], będącego jednocześnie jego prefiksem i taką, że

x[k + 1] < > x[j + 1], jeśli 0 < j < m.

Przyjmijmy NEXT[0] = P[0] = 0.

Algorytm KMP' - wersja on-line algorytmu KMP; "na wejściu" jest ciąg n kolejnych symboli tekstu y, zakończony specjalnym symbolem #.

```
begin
   i := 1; j := 0; read(symbol);
   while symbol < > # do
   begin
      j := NEXT[j];
      while x[j+1] = symbol do
            begin j := j + 1; read(symbol) end;
      if j = 0 then read(symbol);
      if j = m then write(i);
      i := i + max(1,j- NEXT[j]);
   end
end; {algorytm KMP'}
```

Istotna różnica między P[] i NEXT[]
Np. dla $x = a^m$ i $y = a^{m-1}ba$ dla $j = m-1$.

Algorytm GS'
(wersja algorytmu Galila-Seiferasa dla pewnej klasy wzorców)

Wzorzec jest łatwy gdy żadne słowo niepuste postaci vvv nie jest jego prefiksem.

x = abababba nie jest łatwy

x = abbabbbab jest łatwy

Dla łatwych wzorców działa algorytm GS', w którym tablicę P z algorytmu KMP zastępuje się przez pewne przybliżone oszacowanie:

P[j] < 2j/3

przesunięcie(j) >= j/3

{Algorytm GS': wersja algorytmu Galila-Seiferasa dla łatwych wzorców}

```
begin
   i := 1;
   while i <= n – m + 1 do
   begin
      j := 0;
      while x[j+1] = y[i+j +1]  do  j := j + 1;
      if j = m then write(i);
      i := i + max(1, [j/3]); {przesunięcie >= 1}
   end;
end;
```

Złożoność czasowa liniowa, pamięciowa O(1)

Przykład niepoprawnego działania dla wzorca, który nie jest łatwy x = aaaaaaab y = aaaaaaaab.

Dla j = 7 przesunięcie będzie równe 2, a więc pozycja i = 2 zostanie pominięta.

Algorytm KR (Karpa-Rabina)

Stosujemy funkcję mieszającą h, dzięki której można obliczyć pewną wartość (kod) podsłowa y_i = y[i.. i + m - 1] tekstu wejściowego.

Jeśli h(y_i) = h(x), to z bardzo dużym prawdopodobieństwem zachodzi równość x = y_i, a więc wzorzec x występuje w y na i-tej pozycji.

Efektywność algorytmu wynika z możliwości szybkiego obliczenia wartości h(y_i+1), jeżeli wartość h(y_i) jest znana.

Działanie algorytmu polega na obliczaniu tych wartości kolejno dla i = 1, 2, ... , n - m + l.

Alfabet składa się z liter o kodach: **0,1,…,r-1**
q – duża liczba pierwsza
x – tekst o długości **m**

$$h(x) = (x[1]r^{m-1}+x[2]\, r^{m-2}+ \ldots +x[m]) \bmod q$$

Prawdopodobieństwo kolizji - **1/q**

```
{Algorytm KR (Karpa-Rabina)}
begin
    h1 := h(x);
    dM := r^(m-1); h2 := h(y[1..m]) ; {koszt O(m)}
    i := 1;
    while i <= n - m + 1 do
    begin
      if h2 = h1 then
      if x=y[i..i +m-1] then write(i) {koszt = O(m)};
      {oblicz nowe h2 = h (y[i+1..i + m]) wiedząc, że
      poprzednio h2 = h(y[i..i+m - 1]) =
      (y[i]r^(m-1)+y[i + l] r^(m-2)+ ... +y[i+m-1]) mod q}
      h2 := ((h2 - y[i] * dM) * r + y[i + m]) mod q;
      i := i + 1;
    end
 end;
```

Algorytm BM (Boyera-Moore'a)

Jest to ulepszenie zmienionej wersji algorytmu naiwnego.

Algorytm N’ (wersja algorytmu N)

```
begin
   i := 1;
   while i <= n – m + 1 do
   begin
      j := m; while x[j] = y[i+j-1] do  j := j - 1;
      if j = 0 then write(i);
      i := i +1; {przesunięcie równe 1}
   end;
end; {algorytm N’}
```

Tracona wartość **j** spełnia warunki:

(a) x[j + 1 .. m] = y[i + j..i + m - 1];
{do tekstu pasuje końcowa część wzorca, począwszy od pozycji j + 1 we wzorcu}

(b) x[j] < > y[i + j- 1];

Niech s będzie przesunięciem.

Jeśli wzorzec zaczyna się na pozycji i + s, to:

war1(s,j):
przesunięty wzorzec jest zgodny z jego częścią "pasującą" ostatnio do tekstu y;
formalnie: dla każdego j < k <= m zachodzi
s >= k lub x[k - s] = x[k];

war2(s, j):
s >= j lub x[j - s] < > x[j] (warunek ten wynika z niezgodności ostatnio czytanych symboli tekstu i wzorca).

Niech:

d1(j) = min {s >= 1| zachodzi war1(s, j)}
d2(j) = min {s >= 1| zachodzi war1(s, j) i war2(s, j)}

Przykład:

y = aaaaaaaaaababababa
x = cababababa.

Mamy d1(9) = 2,
d2(9) = 8,

a więc d2 daje większe przesunięcia

Modyfikujemy algorytm N’ tak, że przyjmujemy wartość przesunięcia s = d2(j)

Algorytm BM (wersja algorytmu N’)

```
begin
    i := 1;
    while i <= n – m + 1 do
    begin
        j := m; while x[j] = y[i+j-1] do  j := j - 1;
        if j = 0 then write(i);
        i := i +d2(j); {przesunięcie równe d2(j)}
    end;
end; {algorytm BM}
```

Całkowita liczba porównań jest liniowa.

Obliczenie tablicy d2[ ] jest liniowe.

Niech x = cababababa i
y = aaaaaaaaaababababa.

Liczba porównań symboli w algorytmie BM jest równa 12.

Gdybyśmy jako przesunięcia użyli d1(j) a nie d2(j), to wykonalibyśmy 30 porównań.

Okazuje się, że w wypadku użycia d1(j) liczba porównań jest rzędu n^2.

Można to zobaczyć dla $x = ca(ba)^k$ i $y = a^{2k+2}(ba)^k$.

Algorytm FP (Fishera-Patersona)

Problem WW' (wyszukiwanie wzorca z symbolami nieznaczącymi)

Złożoność problemu WW' jest (asymptotycznie) niewiększa niż złożoność problemu mnożenia dwóch liczb całkowitych.

φ – symbol nieznaczący (pasuje do każdego symbolu)

≡ - relacja „pasowania"

Dla symboli s i z:

s ≡ z wttw s = z lub s = φ lub z = φ

dla tekstów x i y:

x ≡ y wttw x[i] = y[i] dla każdej pozycji i.

Relacja ≡ nie jest **przechodnia** !!!

Zatem nie wszystkie algorytmy dla WW działają dla WW' (gdzie sprawdzanie = jest zastąpione przez sprawdzanie ≡)

Jeśli jedyną informację o słowach otrzymujemy za pomocą porównań ≡, to potrzeba ich Ω(n^2).

Kodując teksty x i y jako liczby binarne, można osiągnąć złożoność rzędu niższego niż n^2 (koszt mnożenia liczb).

Nie jest znany algorytm o złożoności liniowej.

Mnożenie tekstów u i v (**):

$w[k] = \bigwedge\{i+j = k+1\} (u[i] \equiv v[j])$

/\ - koniunkcja

```
     baab
≡      ab
-----------
     1001
/\  0110
-----------
   01001
```

Zasada algorytmu:
Wzorzec x występuje w tekście y od pozycji i, gdzie 1 <= i <= n-m+1 wttw
z[i + m – 1] = 1, gdzie z = x^R ** y.

Przykład:

x = baa

y = baaba

x^R = aab

aab ** baaba =

```
      baaba
        aab
      -------
      10010
     01101
    01101
   ------------
   0010010
```

n = 5, m = 3, n – m + 1 = 3

z[3] = z[6] = 1

i + 2 = 3 → i = 1

i + 2 = 6 → i = 4 (ale 1 <= i <= 3)

x występuje w y tylko od pozycji 1.

Mnożenie logiczne „/\,\/” definiujemy podobnie do **, ale używając /\, \/ zamiast ≡,/\

logwektor_a(v) jest wektorem logicznym, którego i-tą składową jest true wttw v[i] = a. (v – tekst)

X_{a,b}(u,v) = logwektor_a(u) „/\,\/” logwektor_b(v)

v ** u = negacji alternatywy logicznej wszystkich wektorów X_{a,b}(u,v), gdzie (a,b) są wszystkimi parami wzajemnie różnych symboli różnych od φ.

WW’ ma ten sam rząd złożoności co iloczyn logiczny wektorów.

Obliczenie iloczynu logicznego wektorów o długości n jest tego samego rzędu co obliczenie iloczynu arytmetycznego dwóch liczb binarnych (o długości nlogn).

Istnieją algorytmy mnożenia liczb binarnych w czasie O(n^r), gdzie r < 2, to problem WW’ można rozwiązać w czasie O((nlogn)^r).