String-matching on ordered alphabets

W oparciu o „String-matching on ordered alphabets”
Maxime Crochemore

Jan Mełech

1 Notacja

- $p(t)$ - najmniejszy okres słowa t
- $p'(t)$ - okres słowa t odpowiadający największemu silnemu prefikso-sufiksowi

2 Wstęp

Autor przedstawia liniowy algorytm do znajdowania wzorca w tekście. Przedstawiony algorytm należy do ogólniejszej klasy algorytmów skanujących tekst od lewej do prawej i wykonujących odpowiednie przesunięcia. Zaproponowany algorytm jest pierwszym algorytmem z tej klasy, który wymaga stałej ilości pamięci.

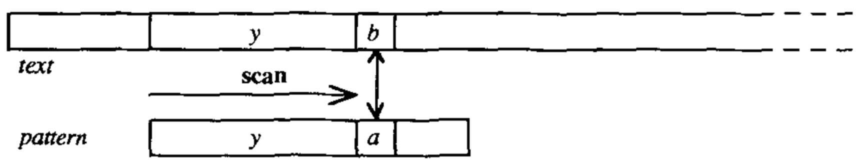
3 Główna idea algorytmu

Przyjmijmy, że mamy sytuację jak na rysunku 1. Algorytmy MP I KMP wykonują w takiej sytuacji przesunięcia odpowiednio o $p(y)$ i $p'(y)$, jednak oba podejścia wymagają pamięci wielkości proporcjonalnej do wielkości wzorca.

Crochemore stosuje nieco inne podejście. Najpierw zauważa, że najlepszym możliwym przesunięciem jest $p(yb)$. Następnie na podstawie dekompozycji słowa yb opartej na maksymalnym sufiksie oblicza przybliżoną wartość $p(yb)$. Samo obliczenie będzie wymagało stałej liczby komórek pamięci, zaś przybliżenie będzie wystarczająco dobre, aby osiągnąć liniową złożoność czasową.

4 Okresy słowa, a maksymalne sufiksy

Niech v będzie maksymalnym leksykograficznie sufiksem niepustego słowa x. Przedstawmy v jako $w^e w'$, gdzie $e \geqslant 1$, $|w| = p(v)$, zaś w' jest właściwym prefiksem w. Niech u będzie prefiksem x, takim że $x = uv = uw^e w'$. Czwórkę (u, w, e, w') będziemy nazywać MS-dekompozycją słowa x.



Rysunek 1: Niedopasowanie tekstu ze wzorcem

Na początku przywołajmy lemat, że element w w MS-dekompozycji nie może mieć właściwego prefikso-sufiksu:

Lemma 4.1. Niech $uw^e w'$ będzie MS-dekompozycją niepustego słowa x. Wtedy zachodzi $p(w) = |w|$.

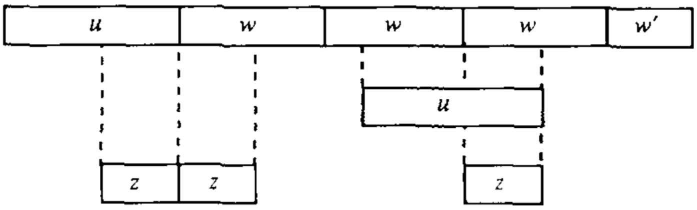
Następnie pokażemy w jaki sposób możemy oszacować najmniejszy okres słowa za pomocą jego MS-dekompozycji:

Lemma 4.2. Niech $uw^e w'$ będzie MS-dekompozycją niepustego słowa x. Wtedy zachodzą następujące własności:

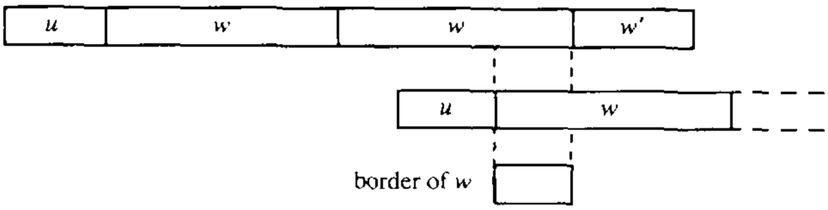
1. jeśli u jest sufiksem w, to $p(x) = p(v)$,
2. $p(x) > |u|$,
3. jeśli $|u| \geqslant |w|$, to $p(x) > |v| = |x| - |u|$,
4. jeśli u nie jest sufiksem w oraz $|u| < |w|$, to $p(x) > min(|v|, |uw^e|)$.

Dowód.

1. Gdy u jest sufiksem w, nietrudno zauważyć, że $|w|$ jest najmniejszym okresem x. Zatem $p(x) = |w| = p(v)$.
2. Gdyby $p(x) \leqslant |u|$, to w x istniałoby drugie wystąpienie v w x niebędące sufiksem x. Zatem x moglibyśmy zapisać jako $u'vv'$, gdzie $|u'| < |u|$ oraz $|v'| > 0$. Z drugiej strony, $vv' > v$ co stoi w sprzeczności, że v jest maksymalnym sufiksem x.
3. Załóżmy przeciwnie, że $p(x) \leqslant |v|$. Zatem w x można znaleźć drugie wystąpienie u w x niebędące prefiksem x, które przecina v. Skoro $|u| \geqslant |w|$, to istnieje z sufiks u, który jest jednocześnie prefiksem w, jak pokazane na rysunku 2. Teraz możemy zapisać $v = zz'$. Skoro v jest maksymalnym sufiksem to dostajemy $zv < v = zz'$. Z tego wnioskujemy, że $v < z'$, sprzeczność z maksymalnością v.
4. Załóżmy przeciwnie, że $p(x) \leqslant min(|v|, |uw^e|)$. Z $p(x) \leqslant |v|$ wynika, że istnieje drugie wystąpienie u w x niebędące prefiksem x. Natomiast z $p(x) \leqslant |uw^e|$ możemy wysnuć wniosek, że to drugie wystąpienie musi się przecinać z w^e. Jeżeli przecinałoby się z granicą dwóch kolejnych



Rysunek 2: Sufiks u jako prefiks w



Rysunek 3: w nie może mieć właściwego prefikso-sufiksu

słów w w w^e lub z granicą pomiędzy ostatnim w i w' to z poprzedniego podpunktu otrzymujemy sprzeczność. Zatem u musi być podsłowem w różnym od jego sufiksu (z założenia). Z okresowości po u musi nastąpić w, co implikuje sytuację widoczną na 3, w której dostajemy, że w ma właściwy prefikso-sufiks, co daje nam sprzeczność z lematem 4.1

□

Corollary 4.2.1. Niech uw^ew' będzie MS-dekompozycją niepustego słowa x. Jeżeli u jest sufiksem w, to $p(x) = p(x) = |w|$. W przeciwnym przypadku, $p(x) > max(|u|, min(|v|, |uw^e|)) \geqslant |x|/2$.

Dowód. Jeżeli u jest sufiksem w to na podstawie pierwszego podpunktu lematu 4.2 dostajemy $p(x) = p(x) = |w|$. W przeciwnym przypadku, na podstawie pozostałych podpunktów lematu 4.2 otrzymujemy nierówność $p(x) > max(|u|, min(|v|, |uw^e|))$. Jeżeli $|u| \geqslant |x|/2$ to od razu dostajemy tezę. Z drugiej strony, jeżeli $|u| < |x|/2$, to $|v| \geqslant |x|/2$. Dodatkowo, $|uw^e| = |x| - |w'| > |x|/2$, ponieważ $2|w'| < |w'| + |w| \leqslant |v| < |x|$. □

5 Obliczanie MS-dekompozycji

MS-dekompozycję słowa x będziemy przechowywać za pomocą MS-czwórki (i, j, k, p). MS-czwórka (i, j, k, p) to cztery liczby charakteryzujące w pełni MS-dekompozycję x:

$$i = |u|, j = |uw^e|, k = |w'| + 1, p = |w| = p(v)$$

gdzie $x = uw^e w'$.

5.1 Optymalizowanie obliczania MS-dekompozycji

Gdyby obliczać w każdej iteracji MS-dekompozycję słowa yb to dla pary $t = a^n, w = a^m, n > m$ nasz algorytm zajmowałby $O(|t||w|)$ czasu. Rozwiązaniem tego problemu jest wykorzystywanie MS-dekompozycji obliczonych w poprzednich iteracjach pod pewnymi warunkami zawartymi w następującym lemacie:

Lemma 5.1. Niech (u, w, e, w') będzie MS-dekompozycją niepustego słowa $x = uw^e w'$. Załóżmy, że $p(x) = |w|$. Wtedy, jeżeli $e > 1$, to $(u, w, e-1, w')$ jest MS-dekompozycją słowa $x' = uw^{e-1}w'$.

5.2 Pseudokod

Do obliczania MS-dekompozycji będziemy używać pomocniczej metody next_maximal_suffix. Na wejściu mamy słowo w długości m or MS-czwórkę (i, j, k, p) dla pewnego słowa z będącego prefiksem w. Sama metoda polega na iterowaniu się po całym słowie w zaczynając od prefiksu z i odpowiednim aktualizowaniu poszczególnych wartości MS-czwórki w zależności od tego czy znaleźliśmy nowy maksymalny sufiks czy nie.

Algorithm 1 next_maximal_suffix($w[1]...w[m], (i, j, k, p)$)

```
t_pos ← 0, w_pos ← 1
(i, j, k, p) ← (0, 1, 1, 1)
while j + k ⩽ m do
  if w[i + k] = w[j + k] then
    if k = p then
      j ← j + p, k ← 1
    else
      k ← k + 1
    end if
  else if w[i + k] > w[j + k] then
    j ← j + k, k ← 1, p ← j − i
  else
    i ← j, j ← i + 1, k ← 1, p ← 1
  end if
end while
return (i, j, k, p)
```

6 Główny algorytm

Przedstawmy główny algorytm string_matching. Wpisuje się on w klasyczny schemat algorytmów skanujących tekst od lewej do prawej i wykonujących przesunięcia. Omówmy najważniejszą część tego algorytmu, czyli obliczenie przesunięcia. Oznaczmy jak wcześniej y - prefiks wzorca już dopasowanego, b - pierwszy znak w tekście, który jest różny od odpowiadającego mu znaku we wzorcu (lub kolejny znak w tekście gdy znaleźliśmy dopasowanie). Najpierw obliczamy MS-czwórkę słowa yb używając metody next_maximal_suffix, dostając $yb = uw^e w'$. Następnie na bazie wniosku 4.2.1 sprawdzamy czy u jest sufiksem słowa w:

- jeżeli tak, to $p(yb) = |w|$ i o tyle możemy przesunąć tekst. Dodatkowo sprawdzamy czy $j - i > p$ (co jest równoważne sprawdzeniu $e > 1$) , aby w miarę możliwości nie obliczać następnej MS-czwórki od początku (na podstawie lematu 5.1).
- jeżeli nie, to przesuwamy o $max(|u|, min(|v|, |uw^e|))$.

6.1 Dowód poprawności

W obu przypadkach wartość przesunięcia nie przekracza $p(yb)$ zatem nic nie przeoczymy na przesunięciach, a na sprawdzanych pozycjach w oczywisty sposób poprawnie zwrócimy wystąpienia wzorca.

6.2 Skrót dowodu złożoności

Najpierw zauważmy, że podczas sprawdzania czy u jest sufiksem w porównamy co najwyżej $i = |u|$ znaków z tekstu $t[text\_pos + 1]...t[text\_pos + i]$. W każdej iteracji tekst przesuniemy o co najmniej i, zatem porównań znaków wykonanych w linii 14 będzie co najwyżej $|t|$ podczas całego algorytmu.

Nastęnie autor pokazuje, że każde pozostałe porównanie w algorytmie string_matching zwiększy wartość wyrażenia $5 \cdot text\_pos + w\_pos + i + j + k$. Na końcu wyrażenie osiągnie wartość $5|t| + 8$, co da nam liniową złożoność czasową.

Możemy prześledzić jak zachowuje się wyrażenie $5 \cdot text\_pos + w\_pos + i + j + k$:

- można zauważyć, że każde porównanie znaków w metodzie next_maximal_suffix zwiększa wartość wyrażenia $i + j + k$ o co najmniej 1,
- pomyślne porównanie znaków w trakcie skanu zwiększa w_pos o 1,
- niepomyślne porównanie znaków rozkłada się na 3 przypadki, każdy z nich jest dość techniczny - pozwoliłem je sobie pominąć.

Algorithm 2 string_matching(t, w, n, m)

```
t_pos ← 0, w_pos ← 1
(i, j, k, p) ← (0, 1, 1, 1)
while t_pos ⩽ n − m do
    while w_pos ⩽ m and t[t_pos + w_pos] = w[w_pos] do
        w_pos ← w_pos + 1
    end while
    if w_pos = m + 1 then
        return  t_pos + 1
    end if
    if t_pos = n − m then
        break
    end if
    (i, j, k, p) ← next_maximal_suffix(w[1]...[w_pos − 1]t[t_pos +
    w_pos], (i, j, k, p))
    if w[1]...w[i] is suffix of prefix of length p of w[i+1]...[w_pos−1]t[t_pos+
    w_pos] then
        t_pos ← t_pos + p, w_pos ← w_pos − p + 1
        if j − i > p then
            j ← j − p
        else
            (i, j, k, p) ← (0, 1, 1, 1)
        end if
    else
        t_pos ← t_pos + max(i, min(w_pos − i, j)) + 1, w_pos ← 1
        (i, j, k, p) ← (0, 1, 1, 1)
    end if
end while
```