Bardzo dziwny algorytm szukania wzorca w słowie

Wojciech RYTTER

W artykule pokażemy, jak można „przetłumaczyć” pewien fakt matematyczny na operację algorytmiczną. Problem szukania wzorca (w najprostszej postaci) polega na znajdowaniu wszystkich wystąpień zadanego słowa x (traktowanego jako wzorzec) jako podsłowa innego słowa y. Na przykład dla $x = baa$, $y = abaabaaabaa$ wszystkimi wystąpieniami są $abaabaaabaa$, $abaabaaabaa$ i $abaabaaabaa$, co zapisujemy symbolicznie jako $1, 4, 8$, od numerów pozycji, na której zaczyna się powtórzenie (pierwsza pozycja ma numer 0). Zajmiemy się algorytmem, który działa jednocześnie w czasie liniowym (tzn. gdy problem ma rozmiar n, liczba operacji jest ograniczona przez $c \cdot n$, dla pewnej stałej c) i w pamięci stałej, nie licząc pamięci, w której przechowujemy dane wejściowe (dwie tablice x, y) i która nie jest modyfikowalna. Rozmiarem problemu jest długość n (dłuższego) słowa y. Jeśli chodzi o złożoność czasową, to liczyć będziemy tylko liczbę operacji porównywania symboli w x i y. Zaczniemy od wyszukiwania bardzo specjalnych podsłów.

Dla słowa z oznaczmy przez $p = okres(z)$ najmniejszą liczbę $s \geqslant 1$, taką że s jest okresem z, tzn. $z[i] = z[i-s]$ dla wszystkich i, dla których obie strony są zdefiniowane.

Definicja
Słowo nazwiemy leksykograficznie maksymalnym (w skrócie *lm*), gdy jest ono leksykograficznie maksymalne spośród wszystkich swoich sufiksów. Na przykład słowo $'rytter'$ nie jest *lm* (gdyż $'ytter' >' rytter'$), natomiast $'wojciech'$ jest typu *lm*.

Jak widać, termin *sufiks* (i *prefiks*) jest przez informatyków rozumiany szerzej niż przez filologów – jest to dowolny końcowy (początkowy) fragment słowa.

Dlaczego słowa o własności *lm* są interesujące? Większość szybkich algorytmów szukania podsłów korzysta z okresów p prefiksów słowa. Obliczanie tych okresów w ogólnym przypadku jest „wąskim gardłem” w projekcie algorytmu. Natomiast dla słów typu *lm* obliczanie okresów jest trywialne. Wynika to z następującego faktu, którego dowód zostawiamy Czytelnikowi.

Fakt 1 (Kluczowa własność słów typu *lm*)
Niech x będzie słowem typu *lm* i $j \geqslant 0$. Wtedy

$(*)$ $\quad [(j = 0) \vee (p = okres(x[0 \ldots j-1]) \;\&\; x[j] \neq x[j-p])] \Rightarrow okres(x[0 \ldots j]) = j + 1.$

„Przetłumaczenie” implikacji $(*)$ na odpowiednią instrukcję jest zasadniczą częścią algorytmu.

Przykład
Niech $x = bababaa$, $j = 6$. Słowo $bababa$ ma okres 2. Jednak następny symbol *„psuje”* ten okres. Zatem słowo x ma okres równy 7.

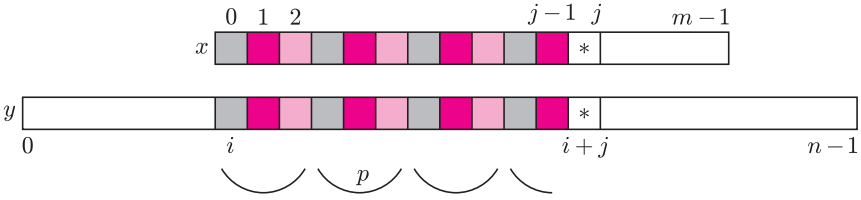
Opiszemy teraz program szukania *lm* podsłowa x w słowie y. Program wczytuje dwa teksty w tablice $x[0 \ldots m-1]$, $y[0 \ldots n-1]$, x jest typu *lm*, a wypisuje wszystkie wystąpienia x w y, tzn. wszystkie takie pozycje i, że $y[i \ldots i+m-1] = x$. Zapisujemy program w języku C++.

Podstawowym niezmiennikiem (patrz rysunek) w programie po każdej iteracji *while* jest:

(A) $j = \max\{j \geqslant 0; j = 0 \text{ lub } x[0 \ldots j-1] = y[i \ldots i+j-1]\}$,

(B) program wypisał wszystkie wcześniejsze wystąpienia $i' < i$,

(C) p jest okresem słowa $x[0 \ldots j-1]$.



Wojciech Rytter, Instytut Informatyki, Uniwersytet Warszawski

Algorytm 1 (dla wzorców x typu lm).

```
#include <iostream.h>
#include <string.h>
#define MAX_LEN 1000
  int i=0,j=0,p=1;
void przesun(void) {
  if (j<2*p) {
    i=i+p; j=0; p=1;
  }
  else {
    j=j-p; i=i+p;
  }
}
void main() {
  char x[MAX_LEN]; char y[MAX_LEN];
  cin>>x>>y;
  while (i <= strlen(y)-strlen(x)) {
    if (j==strlen(x)) {
      cout<<i<<" "; przesun();
    }
    else if (x[j]==y[i+j]) {
      if ((j==0)||(x[j]!=x[j-p])) p=j+1;
      j=j+1;
    }
    else przesun();
  }
}
```

Algorytm sprawdza, czy x zaczyna się od pozycji i-tej. Pewna liczba j symboli wzorca jest zgodna. Następnie algorytm przesuwa wzorzec o wielkość p, która jest okresem segmentu, na którym była zgodność.

Najważniejszą instrukcją w programie jest:

`if ((j==0)||(x[j]!=x[j-p])) p=j+1.`

Instrukcja ta oblicza okres kolejnego prefiksu słowa x, korzystając z Faktu 1 (zapisujemy ten fakt w języku C++). Tak więc poprawność sprowadza się do Faktu 1.

W programie korzystamy ze standardowej funkcji *strlen* zwracającej długość słowa. Operacja $cin \gg x \gg y$ polega na wczytaniu słów x, y do tablic o tych samych nazwach. Operacją dominującą jest porównywanie symboli. Algorytm wykonuje liniową (względem $n = |y|$) liczbę operacji porównywania symboli w słowach – można to udowodnić, obserwując zmiany wartości $2i + j$: wartość ta zwiększa się co najmniej o 1 po każdej iteracji. Jednocześnie $2i + j \leqslant 2n$, gdyż $i + j \leqslant n$.

Algorytm 1 można łatwo zmodyfikować tak, aby znajdował on wystąpienia dowolnego słowa (niekoniecznie typu lm) w czasie liniowym i w stałej pamięci.

Niech $x = uv$, gdzie v jest leksykograficznie maksymalnym sufiksem x. Oznaczmy $r = |u|$. Technicznie informacja o rozkładzie uv sprowadza się do pamiętania r.

Fakt 2
Niech $x = uv$ będzie rozkładem takim, jak wyżej opisany. Wtedy słowo v występuje tylko raz w słowie uv. Jeśli $i' < i$ są początkami wystąpień v, oraz $i - i' < r$, to na pozycji $i - 1$ nie kończy się wystąpienie u.

Z powyższego faktu wynika stosunkowo prosty algorytm szukania x w czasie liniowym i w pamięci stałej. Algorytm ten jest modyfikacją Algorytmu 1, w którym rolę x pełni v.

Algorytm 2

Niech v będzie leksykograficznie maksymalnym sufiksem $x = uv$;

- obliczamy za pomocą Algorytmu 1 kolejne wystąpienia v w y;
- dla każdego wystąpienia i niech i' będzie wystąpieniem poprzednim; jeśli $i - i' \geqslant |u|$, to sprawdź, czy u występuje na lewo od pozycji i; (sprawdzanie to wykonujemy w sposób *naiwny*);
- jeśli występuje, to wypisz kolejne wystąpienie całego wzorca x.

Twierdzenie
Problem szukania słowa x w słowie y można rozwiązać w czasie liniowym i w pamięci (dodatkowej) stałej, jeśli znamy początkową pozycję r leksykograficznie maksymalnego sufiksu v słowa x.

Dekompozycję uv, zapamiętaną na $r = |u|$, można obliczyć w czasie liniowym i w pamięci stałej, korzystając z modyfikacji Algorytmu 1 (odsyłamy Czytelnika po szczegóły do pracy

W. Rytter, *On maximal suffices and constant space versions of KMP algorithm*, Theoretical Computer Science 1-3(302): 211–222 (2003),

z której pochodzą Algorytm 1 i Algorytm 2).