Tablice prefikso-sufiksów

Wojciech RYTTER *

W roku 1971 ukazał się jeden z najważniejszych artykułów związanych z algorytmiką tekstów: *Fast pattern matching in strings* (SIAM J. Comput. 6 (1977): 323–350). Autorami byli: **Knuth, D.E.**, Morris Jr, J.H., oraz Pratt, V.R. W artykule tym głównym problemem było tzw. dopasowywanie wzorca (ang. *string-matching*). Opisano konstrukcję algorytmu znanego później jako algorytm Knutha–Morrisa–Pratta (w skrócie KMP). Kluczowymi strukturami danych w algorytmie są tablice zwane *failure tables*, które będą tematem tego artykułu. Nazywamy je tutaj tablicami prefikso-sufiksów.

Tekst to ciąg symboli zadany w tablicy $x[1 \ldots m]$, której elementami są symbole z pewnego alfabetu A. Liczbę $m = |x|$ nazywamy długością tekstu. Załóżmy, że jedyne dopuszczalne operacje to porównania dwóch symboli. Dwa teksty są równe, gdy są równe jako ciągi symboli.

Problem dopasowywania wzorca polega na wyszukiwaniu wszystkich wystąpień wzorca x w drugim tekście y. Na przykład wzorzec $x = aba$ występuje w słowie $y = abababababb$ i to kilka razy: kolejne wystąpienia wzorca kończą się na pozycjach $3, 5, 7, 9$. Piszemy, że $x = x[1 \ldots m]$ ma wystąpienie na (kończącej wystąpienie) pozycji k w tekście y, gdy $x = y[k - m + 1 \ldots k]$.

Prefiksem słowa nazywamy dowolny jego fragment początkowy, a *sufiksem* – dowolny fragment końcowy.

Prefikso-sufiksem (w skrócie **psufiksem**) danego tekstu x jest *najdłuższy* właściwy (nie będący całym x) prefiks tekstu x będący jednocześnie sufiksem x. W szczególności psufiksem może być słowo puste o długości zerowej.

Na przykład psufiksem tekstu $x = abababab$ jest $ababab$, bo $abababab = ababab ab = ab ababab$ i nie ma żadnego dłuższego słowa, które występowałoby jednocześnie na początku i na końcu x (poza całym x).

Jeśli x jest ustalone, to oznaczmy przez $P[k]$ rozmiar psufiksu słowa $x[1 \ldots k]$. Tablicę $P[0 \ldots |x|]$ nazwiemy tablicą psufiksów. Wartość $P[0]$ nie jest oznaczona, przyjmiemy $P[0] = -1$. Na przykład dla $x = abababababbaa$ mamy

$$P[0 \ldots 11] = [-1, 0, 0, 1, 2, 3, 4, 5, 6, 0, 1, 1].$$

Przedstawimy jeden z możliwych algorytmów liniowych obliczania tablicy P. Jest to iteracyjna wersja algorytmu rekurencyjnego, który można otrzymać, korzystając z następującego faktu (zachęcamy Czytelnika do zrobienia rysunku):

$$x[j] = x[P[j-1]+1] \Rightarrow P[j] = P[j-1] + 1.$$

Algorytm *Obliczanie-Psufiksów*;
  $P[0] := -1$; $t := -1$;
  for $j := 1$ **to** m **do**
    while $t \geqslant 0$ **and** $x[t+1] \neq x[j]$ **do** $t := P[t]$;
    $t := t + 1$; $P[j] := t$;

Złożoność liniowa wynika stąd, że w każdej iteracji zwiększamy wartość t co najwyżej o jeden, a wykonanie każdej operacji $t := P[t]$ zmniejsza wartość t co najmniej o jeden.

Zobaczmy, jakie nieoczekiwane zastosowania ma tablica P.

Obliczanie minimalnego okresu. Pojęciem dualnym do psufiksu jest okres słowa. Okresem tekstu x jest niezerowa liczba naturalna p, taka że $x[i] = x[i+p]$, dla każdego i, dla którego obie strony są zdefiniowane. Przez $per(x)$ oznaczmy minimalny okres x. Okres tekstu x możemy wyliczyć ze wzoru

$$per(x[1 \ldots n]) = n - P[n].$$

Problem dopasowywania wzorca *off-line*. Niech $|x| = m$, $|y| = n$ oraz niech $\#$ będzie „nowym” symbolem. Chcemy znaleźć wystąpienia wzorca x w tekście y. Obliczmy tablicę P dla słowa $x\#y$. Wtedy x występuje na (kończącej wystąpienie) pozycji k w y dokładnie wtedy, gdy $P[m + k + 1] = m$.

*Instytut Informatyki, Uniwersytet Warszawski

W tym problemie szukamy szablonu, który można przyłożyć kilka razy do ściany, tak aby po „pociągnięciu" sprayem otrzymać napis x. Chcemy, żeby szablon był możliwie oszczędny (krótki). Na przykład do uzyskania słowa $x = abaabaa$ wystarczy szablon $z = abaa$ – trzeba go przyłożyć na pozycjach 4 i 7.

Problem najkrótszego słowa pokrywającego. Wystąpienie słowa z na pozycji k słowa x pokrywa pozycje $[k - |z| + 1 \ldots k]$ w słowie x. Mówimy, że słowo z *pokrywa tekst*, gdy każda pozycja w x jest pokryta przez pewne wystąpienie z w x. Chcemy znaleźć *najkrótsze* słowo z, które pokrywa tekst x. Takie z musi być jednocześnie prefiksem i sufiksem x. Wystarczy policzyć jego długość. Minimalną długość z pokrywającego $x[1 \ldots k]$ oznaczamy przez $Pokr[k]$. Korzystamy pomocniczo z wartości $Z[k]$: jest to długość najdłuższego fragmentu $x[1 \ldots i]$, który można pokryć słowem, będącym najkrótszym pokryciem słowa $x[1 \ldots k]$. Jeśli mamy obliczoną tablicę P słowa x, to następujący algorytm liczy wszystkie wartości $Pokr[k]$:

Algorytm *Najkrótsze-Pokrycie*
  for $i := 1$ **to** n **do**
    $Z[i] = i,\ Pokr[i] = i$

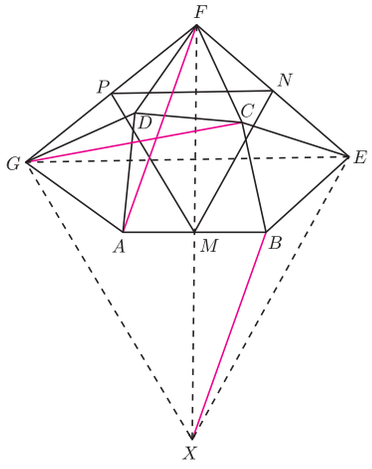
  for $i := 2$ **to** n **do**
    if $P[i] > 0$ **and** $i - Z[Pokr[P[i]]] \leqslant Pokr[P[i]]$ **then**
      $Pokr[i] := Pokr[P[i]],\ Z[Pokr[P[i]]] := i;$

Co to znaczy *on-line*? Możemy wykorzystać jedynie $O(|x|)$ pamięci, a tekst y wczytujemy „litera po literze", na bieżąco udzielając odpowiedzi (tekst y może być za długi, aby zmieścił się w pamięci, albo możemy nie znać z góry jego długości).

Problem dopasowania wzorca *on-line*. Załóżmy, że wyszukujemy wzorzec x w tekście $y = a_1 a_2 \ldots a_n$ (zakończonym symbolem #), którego symbole wczytujemy kolejno z wejścia. Na wyjściu mamy wypisywać 1, gdy po wczytaniu symbolu a_i pozycja i jest jednym z kończących wystąpień x, a 0 w przeciwnym przypadku.

Algorytm *Dopasowanie-on-line*
  $j := 0$
  while *true* **do**
    czytaj następny symbol s tekstu y;
    if $s =$ # **then** STOP
    $j := następny(j, s)$;
    if $j = m$ **then write** 1 **else write** 0;

Rozwiązanie zadania M 1193.
Niech X będzie takim punktem, że trójkąt EGX jest równoboczny, jak pokazano na rysunku. Rozpatrzmy obrót R_D o środku D i kącie $-60°$. Obrót ten przeprowadza punkty A i F odpowiednio na punkty G i C. Następnie rozpatrzmy obrót R_E o środku E i kącie $60°$. Obrót ten przeprowadza punkty G i C odpowiednio na punkty X i B.



Zatem złożenie $R_E \circ R_D$ jest przesunięciem, które przeprowadza punkty A i F odpowiednio na punkty X i B. Wobec tego czworokąt $AFBX$ jest równoległobokiem, skąd wynika, że punkt M – środek odcinka AB, jest również środkiem odcinka FX.

Rozpatrzmy jednokładność o środku F i skali $\frac{1}{2}$. Jednokładność ta przeprowadza trójkąt równoboczny XEG na trójkąt MNP, a więc trójkąt MNP jest również równoboczny.

Funkcję *następny* możemy zaimplementować następująco, korzystając z tablicy P słowa x:

funkcja *następny*(t, s)
  if $t = m$ **then** $t := P[m]$;
  while $t \geqslant 0$ **and** $x[t+1] \neq s$ **do** $t := P[t]$;
  return $t + 1$;

Jeśli $x = aba$, $y = abababababb\#$, to algorytm wypisze ciąg 0 0 1 0 1 0 1 0 1 0 0.

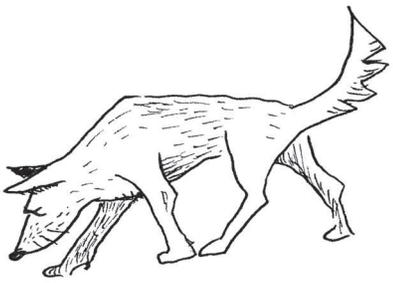
Algorytm Dopasowanie-on-line z tak działającą funkcją *następny* działa w czasie liniowym i jest wersją algorytmu KMP. Oryginalną wersję algorytmu KMP otrzymamy, gdy zamiast tablicy psufiksów użyjemy tablicy P' *silnych psufiksów*. Niech $|x| = m$. Definiujemy $P'[m] = P[m]$ oraz dla $j < m$ definiujemy $P'[j]$ jako największą liczbę naturalną $k \geqslant 0$, taką że $x[1 \ldots k]$ jest sufiksem $x[1 \ldots j]$ oraz dodatkowo $x[j+1] \neq x[k+1]$. Jeśli takiej liczby nie ma, to $P'[j] = -1$. Wartości tablicy P' mogą być znacznie mniejsze niż wartości tablicy P. Na przykład, dla $x = abaab$ mamy

$$P[0 \ldots 5] = [-1,\ 0,\ 0,\ 1,\ 1,\ 2\ ];\quad P'[0 \ldots 5] = [-1,\ 0,\ -1,\ 1,\ 0,\ 2\ ].$$

Oznaczmy przez *następny′* funkcję *następny*, w której zamiast tablicy P używamy P'.

Zauważmy na koniec, że wartości funkcji *następny* można zawczasu *stablicować* w tablicy *NASTĘPNY[j,s]=następny(j,s)*:

Algorytm Obliczanie tablicy *NASTĘPNY*
  {Niech $x = a_1 a_2 \ldots a_m$}
  $NASTĘPNY[0, a_1] := 1$; $NASTĘPNY[0, s] := 0$ dla $s \neq a_1$;
  for $i := 1$ **to** $m - 1$ **do**
    $NASTĘPNY[i, a_{i+1}] := i + 1$;
    $NASTĘPNY[i, s] := NASTĘPNY[P[i], s]$ dla $s \neq a_{i+1}$;
  $NASTĘPNY[m, s] := NASTĘPNY[P[m], s]$ dla każdego s;



Po co funkcje *następny* i *następny′*, skoro możemy skorzystać od razu z tablicy *NASTĘPNY*? Jeśli rozmiar alfabetu (liczba symboli s) jest duży, to tablica *NASTĘPNY* ma rozmiar kwadratowy względem m i nie uzyskamy liniowych złożoności. Natomiast tablica *NASTĘPNY* jest świetna dla małych alfabetów, w szczególności dla alfabetu binarnego.

Zakończymy, stawiając kilka problemów.

Problem 1. Udowodnij, że liczba miejsc niezerowych w tablicy *NASTĘPNY* dla tekstu x nigdy nie przekracza $2\,|x|$.

Problem 2. Niech $t \geqslant 2$. Udowodnij, że liczba porównań symboli (instrukcji $x[t+1] \neq s$) w funkcji *następny′*(t, s) (gdzie stosujemy tablicę silnych psufiksów) jest rzędu $\Theta(\log t)$, podczas gdy liczba ta w funkcji *następny*(t, s) (stosującej tablicę P) jest rzędu $\Theta(t)$. Inaczej mówiąc, *opóźnienie* w algorytmie Dopasowanie-on-line między wczytaniem kolejnego symbolu z wejścia i dostaniem odpowiedzi jest logarytmiczne, gdy używamy silnych psufiksów, a liniowe, gdy normalnych.

Problem 3. Udowodnij, że algorytm Dopasowanie-on-line z implementacją *następny′* lub *następny* działa w czasie liniowym.

Problem 4. Zmodyfikuj algorytm *Obliczanie-Psufiksów* tak, aby w czasie liniowym liczył tablicę P' silnych psufiksów.

Problem 5. Napisz liniowy algorytm tablicowania funkcji *następny′*.

Problem 6. Udowodnij poprawność algorytmu *Najkrótsze-Pokrycie*.

Problem 7(!!). Napisz program, który dla niedużych n szybko liczy liczbę wszystkich ciągów liczbowych długości n, które są tablicami P dla pewnego tekstu.

O notacji asymptotycznej w analizie algorytmów

Łukasz KOWALIK [*]

Załóżmy, że mamy dwa algorytmy dla tego samego problemu i oba są poprawne. Który z nich jest szybszy? Od razu powstaje pytanie: jak zmierzyć czas działania algorytmu? Choć informacja, że nasz algorytm działa tyle a tyle minut na pewnym komputerze dla pewnych danych wejściowych, daje nam jakieś pojęcie o jego skuteczności, jest to jednak informacja bardzo niepełna. Po pierwsze, nie mamy gwarancji, że na innych danych (nawet tego samego rozmiaru), które mogą się pojawić, nasz algorytm nie będzie działał znacznie wolniej. Po drugie, nie mamy żadnego wyobrażenia na temat działania algorytmu dla większych danych. Po trzecie wreszcie, wraz z postępem technologii pojawiają się kolejne generacje komputerów i tego typu informacja szybko się dezaktualizuje.

Rozważmy konkretny przykład algorytmu. Dana jest tablica liczb naturalnych $a[1 \ldots n]$, uporządkowana niemalejąco. Należy sprawdzić, czy a zawiera daną liczbę x. Użyjemy klasycznego algorytmu wyszukiwania binarnego. Z grubsza rzecz biorąc, naśladuje on wyszukiwanie numeru w książce telefonicznej, ale takiej, w której nie mamy żadnej pewności, czy istnieją np. jakiekolwiek nazwiska zaczynające się na „A”. Zaczynamy od porównania x z $a[\lfloor n/2 \rfloor]$. Jeśli mieliśmy szczęście, znaleźliśmy x i kończymy. Jeśli $x < a[\lfloor n/2 \rfloor]$, wiemy, że możemy ograniczyć poszukiwania do $a[1 \ldots \lfloor n/2 \rfloor - 1]$, a w przeciwnym przypadku do $a[\lfloor n/2 \rfloor + 1 \ldots n]$. Powiedzmy, że zdarzył się ten drugi przypadek. Mamy do czynienia z takim samym problemem, tylko o połowę mniejszym. Postępujemy więc tak samo jak poprzednio, czyli porównujemy x z elementem środkowym ciągu $a[\lfloor n/2 \rfloor + 1], \ldots, a[n]$. Ponownie albo znajdujemy x, albo redukujemy przestrzeń poszukiwań o połowę, itd. W końcu może się zdarzyć, że przestrzeń poszukiwań stanie się pusta – wtedy kończymy ze stwierdzeniem, że x nie ma w tablicy.

Jeśli chcemy uniezależnić naszą miarę czasu działania algorytmu od konkretnych danych, to – innymi słowy – chcemy rozważyć wszystkie możliwe zestawy danych. Zwykle algorytmy działają dłużej dla większych danych. Stąd naturalny pomysł: jako miarę złożoności (szybkości) algorytmu przyjmiemy maksymalny czas działania algorytmu dla danych rozmiaru n. Jest to tzw. *złożoność pesymistyczna*. Widzimy, że jest to funkcja postaci $T : \mathbb{N} \to \mathbb{N}$. Spróbujmy znaleźć funkcję T dla wyszukiwania binarnego, ale zamiast czasu będziemy

[*]Instytut Informatyki, Uniwersytet Warszawski