Manber, Myers: zapytania do tablicy sufiksowej w czasie $O(m + \log n)$

Jan Havránek

20.06.2020

Wprowadzenie

Tablica sufiksowa SA zawiera początkowe indeksy leksykograficznie uporządkowanych sufiksów słowa t. Oznaczmy taki sufiks wskazany przez indeks a w SA jako $t_a = t[SA[a] \dots n]$. Mając tablicę sufiksową, można ją wykorzystać do indeksowania tekstu – poprzez zwykłe wyszukiwanie binarne można odnaleźć pierwszą (l_w) i ostatnią (r_w) pozycję w SA taką, że w jest prefiksem $t_i, i \in l_w \leq i \leq r_w$. Wtedy $SA[i]$ wskazują wszystkie wystąpienia w w t. Jednak to podejście wymaga $O(m \log n)$ czasu.

Manber i Myers w 1993 r. zaproponowali prosty algorytm przyśpieszający takie zapytania do $O(m + \log n)$. Wykorzystywana jest do tego struktura LCP-LR zawierająca najdłuższe wspólne prefiksy (*lcp*) sufiksów w.

Zapytania do SA z wykorzystaniem LCP-LR

Załóżmy, że mamy tablicę SA oraz LCP-LR (jej konstrukcja zostanie omówiona w następnej sekcji). Informację o *lcp* sufiksów t można wykorzystać do przyśpieszenia wyszukiwania binarnego, ograniczając liczbę wykonywanych porównań symboli.

Oznaczmy jako l i r lewą i prawą granicę rozpatrywanego przedziału SA w wyszukiwaniu binarnym. $c = (l + r)/2$ jest zatem pozycja oznaczająca sufiks t_c, z którym porównywamy wyszukiwane słowo w. Dalej oznaczmy $L = lcp(t_l, w)$ i $R = lcp(t_r, w)$. Podczas biegu algorytmu będą wartości L i R aktualizowane (niezmiennik), i to z wykorzystaniem minimalnej ilości porównań. Niech $H = \max(L, R)$ i $C = lcp(t_c, w)$.

Bez straty ogólności załóżmy, że w danym cyklu wyszukiwania $H = L \geq R$. Mogą zajść trzy przypadki (w praktyce można pierwsze dwa implementować razem):

1. $lcp(t_l, t_c) > L$
   Wtedy $C = L$.
   w ma pierwszych L znaków zgodnych z t_l i różni się $L+1$-szym. Skoro t_l i t_c zgadzają się przynajmniej w pierwszych $L+1$ znakach, w i t_c będą również mieli pierwszych L znaków identycznych i $L+1$-szy inny, więc $C = L$.

2. $lcp(t_l, t_c) = L$
   W takim przypadku musimy porównać $L+1, L+2 \ldots$-ty symbol w i t_c póki nie znajdziemy i takie, że $w[L+i] \neq t_c[L+i]$. Wtedy $C = L+i-1$. Podobnie jak w przypadku 1, skoro w i t_l są identyczne w pierwszych L znakach oraz t_l i t_c są też identyczne w pierwszych L znakach, to pierwszych L znaków w i t_c będzie również identycznych. Jednak $lcp(w, t_c)$ może być dłuższy od L, więc musimy zrobić skan po znakach dalej.

3. $lcp(t_l, t_c) < L$
   Wtedy $C = lcp(t_l, t_c)$.
   Analogicznie do przypadku 1, tylko na odwrót.

Kiedy znamy C, wystarczy porównać $C+1$-szy symbol w i t_c, żeby zdecydować, czy przesunąć lewą lub prawą granicę przeszukiwania oraz czy zaktualizować wartość L lub $R = C$. W przypadku, kiedy $C = m$, decydujemy według tego, czy szukamy l_w (przesuwamy r) albo r_w (przesuwamy l).

Twierdzenie. *Algorytm poprawnie znajduje l_w/r_w.*

Dowód. Algorytm działa na takiej samej zasadzie, co podstawowe wyszukiwanie binarne, tylko z szybką identyfikacją pierwszego różniącego się symbolu. □

Twierdzenie. *Algorytm działa w czasie $O(m + \log n)$.*

Dowód. Tak samo, jak w wyszukiwaniu binarnym, będziemy musieli wykonać w najgorszym przypadku $O(\log n)$ cykli. Zauważmy, że poza jednym obowiązkowym porównaniem znaku w każdym cyklu, dodatkowe porównywania są wykonywane tylko w opcji 2 z wyżej wymienionych. Biorąc pod uwagę, że

jest to jedyna opcja, przy której się zmienia H (jeżeli zachodzi opcja 3, wtedy szukane l_w/r_w znajduje się w lewej połowie i w związku z tym aktualizujemy R, czyli mniejszą z wartości L, R), że H może tylko rosnąć (przybliżamy się do szukanej pozycji) oraz że H jest z definicji ograniczone przez m, takich porównań będzie maksymalnie m. Łącznie więc otrzymujemy $O(m + \log n)$ porównań. □

LCP-LR i jej konstrukcja

LCP-LR pozwala na zapytania o *lcp* dwu sufiksów t w czasie $O(1)$. Jeżeli mielibyśmy przechowywać tę informację o każdych dwu sufiksach t, *LCP-LR* zajmowałaby $O(n^2)$ pamięci, jednak ograniczenie się do par potrzebnych w trakcie wyszukiwania binarnego redukuje wymagania pamięciowe do $O(n)$.

Manber i Meyers proponują implementację *LCP-LR* jako dwu tablic *Llcp* i *Rlcp*, gdzie dla każdej trójki pozycji (r, c, l), która może zostać rozpatrzona w trakcie wyszukiwania binarnego, $Llcp[c] = lcp(t_l, t_c)$ i $Rlcp[c] = lcp(t_r, t_c)$. Alternatywnie można zastosować słownik (hash table) indeksowany dwójkami $LCP\text{-}LR[(a, b)] = lcp(t_a, t_b)$.

Mając tablicę LCP (obliczoną np. algorytmem Kasai), potrzebne *lcp* można łatwo uzyskać z wykorzystaniem rekursji:

$$lcp(t_l, t_r) = \begin{cases} LCP[r] & \text{jeżeli } r = l + 1 \\ min(lcp(t_l, t_c), lcp(t_c, t_r)) & \text{wpp} \end{cases}$$

gdzie $c = (l + r)/2$.

Twierdzenie. *Powyższy algorytm zwraca poprawne wartości LCP-LR dla wszystkich potrzebnych* (a, b).

Dowód. Algorytm wybiera pary (a, b) w dokładnie ten sam sposób co wyszukiwanie binarne, uzyskamy więc dokładnie te same (a, b). Biorąc pod uwagę, że sufiksy w SA są uporządkowane leksykograficznie, $lcp(t_a, t_b) = \min(lcp(t_a, t_{a+1}), lcp(t_{a+1}, t_{a+2}) \ldots lcp(t_{b-1}, t_b)) = \min_{a+1 \le i \le b} LCP[i]$, co gwarantuje poprawność wyników. □

Twierdzenie. *LCP-LR uzyskujemy w czasie* $O(n)$.

Dowód. Ilość porównań jest z góry ograniczona przez

$$\sum_{i=0}^{\log_2 n} \frac{n}{2^i} = 2n \in O(n)$$

□