Drzewa i tablice sufiksowe

referat z seminarium *zaawansowane struktury danych* prowadzonego przez mgr Pawła Rzechonka w semestrze zimowym 2010/2011

Grzegorz Myszkier

14 marca 2011

1 Wstęp

Wyszukiwanie wzorca jest zadaniem znanym z dziedziny algorytmów tekstowych. Oparte na automatach skończonych algorytmy Boyera-Moore'a, Morrisa-Pratta, Knutha-Morrisa-Pratta i ich dalsze modyfikacje dobrze spełniają swoją rolę pod warunkiem, że zależy nam na wyszukaniu zadanego, pojedynczego wzorca w tekście. Gdy interesuje nas wyszukiwanie wielu wzorców w tym samym tekście, porównywanie tekstów lub określanie samopodobieństwa tekstu, te algorytmy wciąż są użyteczne. Istnieją natomiast struktury danych, które mogą nam pomóc w wyszukiwaniu z lepszą złożonością czasową, takie jak drzewa i tablice sufiksowe. Są one dwiema różnymi strukturami danych, które nadają się do przeprowadzania użytecznych zapytań związanych z wyszukiwaniem w tekście na wejściu.

2 Definicje

Przypomnijmy i zdefiniujmy podstawowe pojęcia:

Σ – *alfabet*
$\cdot$ – operator *konkatenacji*, zwyczajowo $a \cdot b$ oznaczamy przez ab
$|w|$ – długość słowa w
$\preccurlyeq$ – porządek leksykograficzny
$T = t_0 t_1 t_2 \ldots t_{n-1}$ – *tekst* długości n, w którym $t_i \in \Sigma$
$T_i = t_i t_{i+1} t_{i+2} \ldots t_{n-1}$ – i-ty *sufiks* T

Aby ustalić ścisły porządek na sufiksach danego tekstu, będzie wygodnie dołożyć na koniec tekstu znak #. Przyjmujemy, że jest leksykograficznie mniejszy od dowolnego symbolu z Σ.

3 Drzewa sufiksowe

Drzewo sufiksowe dla tekstu T jest drzewem *trie* przechowującym każdy sufiks T_i.

3.1 Wyszukiwanie

3.2 Budowanie drzewa sufiksowego

4 Tablice sufiksowe

Tablica sufiksowa (ang. *suffix array*) SA_T dla tekstu T to tablica zawierająca indeksy sufiksów T posortowane według porządku leksykograficznego tych sufiksów. Weźmy dla przykładu słowo

$$T = \begin{array}{cccccccccccc} \text{M} & \text{I} & \text{S} & \text{S} & \text{I} & \text{S} & \text{S} & \text{I} & \text{P} & \text{P} & \text{I} & \# \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \end{array}$$

Porządek sufiksów przedstawia poniższa tabela

i	$SA_T[i]$	$T_{SA_T[i]}$
0	11	#
1	10	I#
2	7	IPPI#
3	4	ISSIPPI#
4	1	ISSISSIPPI#
5	0	MISSISSIPPI#
6	9	PI#
7	8	PPI#
8	6	SIPPI#
9	3	SISSIPPI#
10	5	SSIPPI#
11	2	SSISSIPPI#

Czasami będziemy potrzebować informacji na temat podobieństw dwóch sufiksów sąsiadujących w tablicy SA_T. Może być wtedy przydatny dostęp do tablicy LCP_T najdłuższych wspólnych prefiksów (ang. *longest common prefix*).

Definicja 1. $LCP_T[i] = max\{|w| : w \text{ jest prefiksem } T_{SA_T[i]} \text{ i } T_{SA_T[i+1]} \}$

Dla naszego słowa MISSISIPI# będzie to

$$LCP_T = (0, 1, 1, 4, 0, 0, 1, 0, 2, 1, 3) .$$

4.1 Wyszukiwanie wzorca

Dany jest wzorzec $P = p_0p_1p_2 \ldots p_{m-1}$. Wyszukiwanie wzorca M w tekście T można wykonać za pomocą algorytmu 1, który realizuje wyszukiwanie binarne. Nie jest to najszybszy algorytm służący do znajdowania wystąpienia wzorca. Jest jednak szybszy od algorytmu Knutha-Morrisa-Pratta dla T znacznie dłuższych od P. Łatwo udowodnić jego złożoność czasową.

W algorytmie stosujemy porównywanie dwóch słów w porządku $\preccurlyeq$. Jedno takie porównanie trwa proporcjonalnie długo do długości wzorca m. Binarne przeszukiwanie zakresu dostępnych indeksów tekstu gwarantuje, że porównań słów będzie $O(\log n)$. Całość dokona się zatem w czasie $O(m \log n)$.

Algorithm 1 Szukanie wzorca w tekście

```
1: procedure FIND(String P, String T, Integer[] SA)
2:     if P ≼ T_{SA_T[0]} then
3:         return 0
4:     else if P ≻ T_{SA_T[n−1]} then
5:         return n − 1
6:     else
7:         l ← 0
8:         r ← n − 1
9:         while r − l > 1 do
10:            m ← (l + r)/2
11:            if P ≼ T_{SA_T[m]} then
12:                r ← m
13:            else
14:                l ← m
15:            end if
16:        end while
17:    end if
18:    return r
19: end procedure
```

4.2 Budowa tablicy sufiksowej

Zaprezentowany niżej algorytm Kärkkäinena-Sandersa buduje tablicę sufiksową w czasie i pamięci $O(n)$ Pozwolę sobie na niezałączenie pseudokodu procedur RadixSortThrees, RadixSortPairs i Merge i pozostanę przy opisie słownym. Nie są one na tyle ciekawe, by zapełniać treść tej pracy. Aby czytelnik zrozumiał działanie tej metody, trzeba przybliżyć pojęcie rangi. Ranga sufiksu T_i określa jego pozycję w porządku sufiksów. Tablica rang R jest w pewnym sensie odwrotnością tablicy sufiksowej. Ściślej rzecz ujmując $R[i] = j \Leftrightarrow SA[j] = i$.

Algorytm *KS* jest algorytmem rekurencyjnym. Spójrzmy na zapis procedury ComputeRanks z algorytmu 2. Na wstępie dzieli indeksy na dwa zbiory względem reszty z dzielenia indeksu przez 3. Indeksy podzielne przez 3 zostają spisane w tablicy S_0 a pozostałe w S_{12}.

Wykorzystajmy S_{12} do utworzenia nowego słowa.Niech

$$\overline{S_{12}} = (t_1t_2t_3)(t_2t_3t_4)(t_4t_5t_6)(t_5t_6t_7)\dots(t_kt_{k+1}t_{k+2}),$$

gdzie k jest rozmiarem tablicy S_{12} po przejściu kroków 2-9. Jak widać, jest to konkatenacja kolejnych trójek słowa T pomijając te zaczynające się na litery o indeksach podzielnych przez 3. Warto zwrócić uwagę na jeden szczegół techniczny. Jeśli pierwszy lub drugi symbol ostatniej trójki jest ostatnim symbolem T, wówczas na kolejne pozycje tej (i być może wcześniejszej) trójki dokładamy brakujące #. Każdą trójkę traktujemy jako pojedynczy symbol pewnego nowego alfabetu.

Na tym etapie chcielibyśmy znać lokalne rangi słowa $\overline{S_{12}}$. Będą one użyte do określenia rang słowa T. Aby to osiągnąć, wykonujemy sortowanie pozycyjne trójek. Pozycje trójek w posortowej tablicy mogą stanowić lokalne rangi, pod

warunkiem, że każde dwie trójki rożnią się. Jeśli tak nie jest, wywołujemy ComputeRanks z argumentem R_{12}. To nam gwarantuje, że po wykonaniu kroków 10-12 R_{12} jest tablicą różnowartościową.

W kolejnym kroku chcielibyśmy określić lokalne rangi dla pozycji podzielnych przez 3. Na tym etapie jest już łatwiej. Możemy skorzystać z tablicy R_{12}. Niech

$$\overline{S_0} = (t_0, R_1)(t_3, R_4)(t_6, R_7)\dots(t_j, R_{j+1}).$$

Mając dane lokalne rangi R_{12} i symbole bez problemu przeprowadzimy sortowanie pozycyjne par słowa $\overline{S_0}$. Podobnie jak wcześniej, otrzymujemy tablicę R_0.

Algorithm 2 Kärkkäinena-Sandersa

```
1:  procedure ComputeRanks(String T)
2:      j ← 0, k ← 0
3:      for i = 0 ... |T| − 1 do
4:          if i ≡3 0 then
5:              S0[j++] ← i
6:          else
7:              S12[k++] ← i
8:          end if
9:      end for
10:     R12 ← RadixSortThrees(S12)
11:     if ∃i ≠ j . R12[i] = R12[j] then
12:         R12 ← ComputeRanks(R12)
13:     end if
14:     R0 ← RadixSortPairs(S0, R12)
15:     R ← Merge(R0, R12)
16:     return R
17: end procedure

18: procedure SuffixArray(String T)
19:     R ← ComputeRanks(T)
20:     for i = 0 ... |T| − 1 do
21:         SA[R[i]] = i
22:     end for
23:     return SA
24: end procedure
```

Pozostało scalić R_0 i R_{12}. Nie jest to trudne, gdy na potrzeby scalania wyliczymy sobie ich odwrotności. Dzięki temu wyznaczymy częściową kolejność wybierania elementów. Trzeba jeszcze zdefiniować porządek na pozycjach z dwóch różnych klas. Należy przyjąć, że dla $i \equiv_3 0$

$$\begin{cases} (t_i, R_{i+1}) \prec (t_j, R_{j+1}) \Leftrightarrow t_i \prec t_j \vee (t_i = t_j \wedge R_{i+1} < R_{j+1}) & j \equiv_3 1 \\ (t_i, t_{i+1}, R_{i+2}) \prec (t_j, t_{j+1}, R_{j+2}) \Leftrightarrow \begin{array}{l} (t_i \prec t_j)\vee \\ (t_i = t_j \wedge t_{i+1} \prec t_{j+1})\vee \\ (t_i t_{i+1} = t_j t_{j+1} \wedge R_{i+2} < R_{j+2}) \end{array} & j \equiv_3 2 \end{cases}.$$

Prześledźmy czas wykonania jednego wywołania procedury ComputeRanks. Przebiegnięcie tablicy w krokach 3-9 zajmuje czas $O(n)$. Wykonanie sortowania pozycyjnego w linii 10 trwa $O(n)$ czasu, podobnie jak w linii 14. Utworzenie dwóch odwrotnych tablic i scalenie w linii 15 odbywa się również w czasie liniowym. Zauważmy, że przy rekursji przekazujemy w argumencie tablicę o rozmiarze $\frac{2}{3}$ rozmiaru tablicy wejściowej, z dokładnością do jednego elementu. Stąd wzór rekurencyjny na złożoność procedury ComputeRanks $T(n) = T(\frac{2}{3}n) + O(n)$ a to prowadzi do wniosku, że $T(n) = O(n)$.

4.3 Tworzenie tablicy LCP

Tablicę LCP możemy obliczyć, gdy mamy już daną tablicę sufiksową dla danego tekstu. Zanim obejrzymy algorytm warto najpierw rozważyć niezbyt skomplikowaną tezę.

Lemat 1. *Niech R_T i LCP_T oznaczają odpowiednio tablice rang i najdłuższych wspólnych prefiksów dla danego tekstu T. Wówczas dla $i \in [1 \dots n-2]$ zachodzi*

$$LCP_T[R_T[i]] \geqslant LCP_T[R_T[i-1]] - 1$$

Dowód. Gdy $LCP_T[R_T[i-1]] - 1 = 0$, wówczas nierówność zawsze zachodzi. Przyjmijmy, że $LCP_T[R_T[i-1]] = l > 1$. Wtedy

$$\exists T[j..k)\ .\ T[j..j+l) = T[i-1..i+l-1).$$

Czyli $T[j+1..j+l) = T[i..i+l-1)$. □

Przejdźmy do algorytmu.

Algorithm 3 obliczanie LCP

```
1:  procedure ComputeRanks(String T)
2:      l ← 0
3:      for i = 0 ... n − 1 do
4:          if R[i] ⩾ 1 then
5:              j ← SA[R[i] − 1]
6:              while S[i + l] = S[j + l] do
7:                  l + +
8:              end while
9:              LCP[R[i]] ← l
10:             l ← max{l − 1, 0}
11:         end if
12:     end for
13: end procedure
```

Poprawność algorytmu wynika z lematu 1. Jeśli policzyliśmy wcześniej długość prefiksu dwóch sufiksów, możemy to wykorzystać przy liczeniu długości dla kolejnego sufiksu. Algorytm obliczający tablicę najdłuższych wspólnych prefiksów przebiega wszystkie i od 0 do $n-1$. Wydaje się, że liniowy czas działania jest zaburzony przez działającą wewnątrz pętlę *while* inkrementującą wartość l. Ale wystarczy tylko zauważyć, że l nie może inkrementować więcej niż n razy a liczba dekrementacji jest też nie większa od n. Dzięki temu algorytm 3 działa w czasie $O(n)$;

5 Podsumowanie

Literatura

[1] M. Crochemore, W. Rytter: *Jewels of stringology*, World Scientific Publishing Co. Pte. Ltd, 2002

[2] J. Kärkkäinen, P. Sanders: *Simple Linear Work Suffix Array Construction*

[3] J. Karkkainen, P. Sanders, S. Burkhardt: *Linear Work Suffix Array Construction*

[4] P. Ko, S. Aluru: *Space Efficient Linear Time Construction of Suffix Arrays*