Projekt Algorytmy Tekstowe

Rafał Kilar

30 maja 2022

Wstęp

Rozważamy problem najdłuższego wspólnego podciągu. Mamy dane dwa słowa $A[1..m]$ i $B[1..n]$. Przedstawimy algorytm Hunta-Szymanskiego działający w czasie $O((r+m)log\ n+n)$ dla słów z r dopasowaniami i d dominującymi dopasowaniami. Następnie przedstawimy jego usprawnioną wersję działającą w czasie $O(m\ log\ n + d\ log(2mn/d) + n)$.

Algorytm Hunta-Szymanskiego

Algorytm i omówienie na podstawie pracy [HS77].

Definiujemy wartości progowe $T_{i,k}$ jako

$$T_{i,k} = \text{ najmniejsze } j \text{ takie że } A[i..i] \text{ i } B[1..j] \text{ posiadają podciąg długości } k$$

Możemy udowodnić kilka prostych faktów o wartościach progowych:

Lemat 1. *Jeśli $T_{i,1}, \ldots, T_{i,k}$ są zdefiniowane, to $T_{i,1} < \cdots < T_{i,k}$.*

Dowód. Zauważmy, że $B[T_{i,k}]$ jest ostatnim znakiem pewnego wspólnego podciągu $A[1..i]$ i $B[1..T_{i,k}]$. W przeciwnym przypadku $T_{i,k}$ nie byłoby minimalne. Wiemy więc, że $A[1..i]$ i $B[1..T_{i,k}-1]$ mają wspólny podciąg długości $k-1$, zatem $T_{i,k-1} \leqslant T_{i,k} - 1$.

Lemat ten przyda nam się w konstrukcji algorytmu.

Lemat 2. $T_{i,k-1} < T_{i+1,k} \leqslant T_{i,k}$

Dowód. Ponieważ $A[1..i]$ i $B[1..T_{i,k}]$ mają wspólny podciąg długości k, to mają je także $A[1..i+1]$ i $B[1..T_{i,l}]$, więc $T_{i+1,k} \leqslant T_{i,k}$. $A[1..i+1]$ i $B[1..T_{i+1,k}]$ mają wspólny podciąg długości k. Usunięcie końcowych znaków z tych słów zmniejsza długość najdłuższego wspólnego podciągu o co najwyżej 1, więc $A[1..i]$ i $B[1..T_{i+1,k}-1]$ mają wspólny podciąg długości $k-1$, czyli $T_{i,k-1} \leqslant T_{i+1,k} - 1$.

Używając tego lematu możemy wyznaczyć wartości $T_{i,k}$

Lemat 3.

$$T_{i+1,k} = \begin{cases} \text{najmniejsze } j \text{ takie że } A[i+1] = B[j] \text{ i } T_{i,k-1} < j \leqslant T_{i,k} \\ T_{i,k} \text{ jeśli takie } j \text{ nie istnieje} \end{cases}$$

Dowód. Rozważamy przypadek, że takie j nie istnieje. Z minimalności $T_{i+1,k}$, każde maksymalne wspólny podciąg $A[1..i+1]$ i $B[1..T_{i+1,k}]$ kończy się w $B[T_{i+1,k}]$. Z lematu 2. i założenia o j,

$B[T_{i+1,k}] \neq A[i+1]$, więc rozważane podciąg jest także podciągiem $A[1..i]$ i $B[1..T_{i+1,k}]$, więc $T_{i,k} \leqslant T_{i+1,k}$ i z lematu 2. $T_{i,k} = Ti + 1, k$.

Załóżmy teraz, że takie minimalne j istnieje. $A[1..i+1]$ i $B[1..j]$ mają wspólny podciąg długości k - wspólne podciąg długości $k-1$ $A[1..i]$ i $B[1..T_{i,k-1}]$ powiększone o $A[i+1]$ i $B[j]$, więc $T_{i+1,k} \leqslant j$.

Załóżmy niewprost, że $T_{i+1,k} < j$. Z lematu 2. mamy $T_{i,k-1} < T_{i+1,k}$, więc ostatni znak wspólnego podciągu $A[1..i+1]$ i $B[1..T_{i+1,k}]$ długości k nie jest równy $A[i+1]$. $A[1..i]$ i $B[1..T_{i+1,k}]$ muszą również mieć podciąg długości k, więc $T_{i,k} \leqslant T_{i+1,k}$. Z lematu 2. mamy więc równość $T_{i,k} = T_{i+1,k}$. Z założenia niewprost i wymogu $T_{i+1,k} < j \leqslant T_{i,k}$ mamy $T_{i,k} \neq T_{i+1,k}$, co daje nam sprzeczność i kończy dowód.

Algorithm 1 Algorytm HS

```
for i = 1..m do
    MATCHLIST[i] = {j_1, j_2, ..., j_p} such that j_1 > ··· > j_p and A[i] = B[j_k]
    THRESH[i] = n + 1
end for
THRESH[0] = 0
LINK[0] = 0
for i = 1..m do
    for j in MATCHLIST[i] do
        find k such thatTHRESH[k − 1] < j < THRESH[k]
        if j < THRESH[k] then
            THRESH[k] = j
            LINK[k] = newnode(i, j, LINK[k − 1]
        end if
    end for
end for
recover LCS in reverse using the LINK array
```

W naszym algorytmie będziemy wyznaczać kolejne wiersze $T_{i,1}, \ldots$.

Algorytm będzie iterował się po kolejnych znakach A. Na początku każdej iteracji tablica $THRESH$ będzie zawierała kolejne wartości $T_{i,1}, T_{i,2}, \ldots$. Lista $MATCHLIST[i]$ zawiera wszystkie indeksy j takie, że $B[j] = A[i]$ w malejącej kolejności.

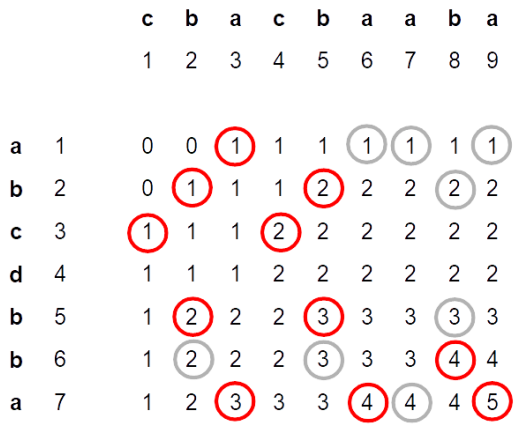
Załóżmy, że dla w iteracji i algorytm rozważa dopasowania $A[i]$ do $B[j_1], \ldots, B[j_p]$ pomiędzy wartościami progowymi, to jest $THRESH[k-1] = T_{i-1,k-1} < j_1 < \cdots < j_p \leqslant T_{i-1,k} = THRESH[k]$. Z lematu 3. $T_{i,k} = j_1$. Ponieważ rozważamy indeksy w malejącej kolejności, poprawimy wartość $THRESH[k]$ na kolejne $j_p, j_{p-1}, \ldots$ aż skończymy na j_1.

Ponieważ lemat 1 mówi nam, że wartości w $THRESH$ są w kolejności rosnącej, możemy wyszukiwać k używając wyszukiwania binarnego w czasie $O(log\ n)$. Jeśli oznaczymy ilość wszystkich dopasowań pomiędzy A i B, to wykonanie głównej pętli zajmuje $O(r\ log\ n)$. Wyznaczenie list $MATCHLIST$ możemy wykonać sortując A i B pamiętając indeksy i wykonać merge na tych ciągach w łącznym czasie $O(n\ log\ n)$. Potrzebujemy $O(n)$ pamięci na wykorzystywane listy i tablice oraz $O(r)$ pamięci na linki.

Łączna złożoność czasowa to $O((r+n)log\ n)$ i pamięciowa $O(r+n)$.

Usprawniony algorytm

Przedstawimy teraz usprawnioną wersję powyższego algorytmu zaproponowaną w [Apo86]. Zanim to zrobimy wprowadzimy kilka pojęć. Powiemy, że dopasowanie $[i, j]$ (para taka, że $A[i] = B[j]$) ma rank k jeśli $A[1..i]$ i $B[1..j]$ mają najdłuższy wspólny podciąg długości k. Dopasowanie $[i, j]$ nazwiemy k-dominującym jeśli $[i, j]$ ma rank k i dla każdego dopasowania $[i', j']$ o ranku k zachodzi albo $i' > i \wedge j' \leqslant j$ albo $i' \leqslant i \wedge j' > j$. Liczbę wszystkich dominujących dopasowań oznaczymy przez d. Na poniższym rysunku dominujące dopasowania oznaczone są czerwonymi okręgami.



Aby przyśpieszyć algorytm 1 zmodyfikujemy proces wyznaczania k-dominujących dopasowań. Nie będziemy sprawdzać wszystkich dopasowań. Zamiast tego, przechowujemy dla każdego symbolu σ listę $MATCHLIST[\sigma]$ listę aktywnych indeksów, to jest takich, które nie wyznaczają wartości progowych. Będziemy wyznaczać jedynie dominujące dopasowania. W tym celu posłużą nam operacje na słownikach:

- $SEARCH(key, LIST)$ zwraca najmniejszy element listy $LIST$ nie mniejszy niż key lub $n + 1$ jeśli taki element nie istnieje. $SEARCH(n + 1, LIST)$ od razu zwraca $n + 1$.
- $INSERT(key, LIST)$ i $DELETE(key, LIST)$ odpowiednio dodają i usuwają wartość key z listy $LIST$. Jeśli $key = n + 1$, to nic nie robią.
- $first(LIST)$ zwraca najmniejszy element w liście $LIST$

Algorytm przedstawiono poniżej (2). W zewnętrznej pętli iterujemy się po kolejnych znakach A. W każdej iteracji wewnętrznej pętli algorytm rozpatruje kolejne dominujące dopasowania $[i, j]$, wyszukuje próg T, który ma zastąpić j. Następnie aktualizuje listę $THRESH$ i listy aktywnych dopasowań $AMATCHLIST[\sigma]$ i zapamiętuje linki. Następnie wyszukuje kolejne dominujące dopasowanie $[i, j]$ dla $j >= T$ w liście aktywnych dopasowań $AMATCHLIST[A[i]]$.

W wewnętrznej pętli zachowujemy następujący niezmiennik: jeśli $j \neq n + 1$, wtedy $[i, j]$ jest k-dominującym dopasowaniem i pierwsze $k - 1$ pozycji w $THRESH$ zawiera poprawne wartości dla i-tego wiersza. Po i-tej iteracji zewnętrznej pętli następujące niezmienniki są zachowane:

- $THRESH$ zawiera wartości $T_{i,1}, \ldots, T_{i,l}$
- $AMATCHLIST[\sigma]$ zawiera indeksy σ w B, które nie są w $THRESH$

Algorithm 2 Algorytm HS1

```
for i = 1..m do
    σ = A[i]
    j = first(AMATCHLIST[σ])
    FLAG = true
    while FLAG do
        T = SEARCH(j, THRESH)
        k = rank(T)
        if T = n + 1 then
            FLAG = false
        end if
        INSERT(j, THRESH)
        DELETE(T, THRESH)
        LINK[k] = newnode(i, j, LINK[k − 1])
        σ′ = B[T]
        DELETE(j, AMATCHLIST[σ]
        j = SEARCH(T, AMATCHLIST[σ])
        INSERT(T, AMATCHLIST[σ′]
    end while
end for
recover LCS in reverse using the LINK array
```

Czas działania zależy od implementacji list $THRESH$ i $MATCHLIST[\sigma]$. Skorzystamy ze struktury danych nazwanej C-drzewo opisanej w pracy [AG87]. Reprezentuje ona uporządkowaną listę elementów z ustalonego uniwersum U jako statyczne drzewo binarne. Kolejne liście odpowiadają kolejnym wartością z U. Dodatkowo utrzymujemy wskaźniki na niedawno odwiedzone liście. Pozwala to na szybsze wyszukiwanie kolejnych wartości - możemy zacząć od zapamiętanego liścia i przejść najpierw w górę i później w dół po najkrótszej ścieżce do szukanego liścia. Po zakończeniu operacji przestawiamy ten wskaźnik.

Możemy pokazać, że jeśli wykonujemy operacje na liściach $i_0 < i_1 < \cdots < i_k$, gdzie $b_j = i_j - i_{j-1}$ operacje te wymagają czasu $O(log\ m + \sum_{j=1}^{k} log\ b_j)$. Pierwsza operacja działa w czasie $O(log\ m)$, gdyż wskaźnik może wskazywać na późniejszy liść. Musimy wtedy przestawić wskaźnik na pierwszy liść. Następne operacje przechodzą po najkrótszej ścieżce między liśćmi, mają one długość $O(log\ b_j)$.

W i-tej iteracji zewnętrznej pętli znajdujemy d_i dominujących dopasowań. Operacja na określonej liście zajmują $O(log\ n + \sum_{j=1}^{d_i} log\ b_j)$ czasu. Ponieważ rozmiar list to co najwyżej n, to $\sum_{j=1}^{d_i} b_j \leqslant 2n$. We wszystkich iteracjach wykorzystywany czas wynosi $O(m\ log\ n + \sum_{j=1}^{d} b_j)$, gdzie $\sum_{j=1}^{d} b_j \leqslant 2nm$. Czas jest maksymalizowany gdy $b_j = 2nm/d$.

Czas działania algorytmu wynosi więc $O(m\ log\ n + d\ log(2nm/d) + n)$ gdy uwzględnimy liniowy czas potrzebny na inicjalizację.

Literatura

[AG87] Alberto Apostolico and Concettina Guerra. The longest common subsequence problem revisited. *Algorithmica*, 2(1):315–336, 1987.

[Apo86] Alberto Apostolico. Improving the worst-case performance of the hunt-szymanski strategy

for the longest common subsequence of two strings. *Information Processing Letters*, 23(2):63–69, 1986.

[HS77] James W Hunt and Thomas G Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353, 1977.