Omówienie pracy
“A Linear Space Algorithm for the LCS Problem“
autorstwa S. Kiran Kumar oraz C. Pandu Rangan

Bartłomiej Jachowicz

1 Główna idea algorytmu

Algorytm zaprezentowany w pracy opiera się na zastosowaniu techniki *dziel i zwyciężaj*. Pozwala ona znaleźć szukany najdłuższy wspólny podciąg w złożoności $O(n(m-p))$ oraz przy wykorzystaniu liniowego rozmiaru pamięci. W poniższym opisie A oraz B oznaczają słowa o długości odpowiednio m oraz n, które są argumentami algorytmu, natomiast C to szukany najdłuższy wspólny podciąg (LCS) o długości p. Zapis $T(i:j)$ oznacza podsłowo słowa T z początkiem o indeksie i oraz końcem o indeksie j. W opisie pracy zakładam że słowa indeksowane są od cyfry 1, implementacja algorytmu ze względu praktycznych wykorzystuje indeksowanie słowa od 0 - przy wywołaniach rekurencyjnych na podsłowach trzeba byłoby przekazywać za każdym razem słowa o jeden znak dłuższe.

Pierwszą rzeczą, którą wykonuje algorytm jest obliczenie długości najdłuższego wspólnego podciągu. Znając jego długość możemy obliczyć *perfekcyjne cięcie*, które definiowany jest w następujący sposób:

Definicja 1.1. Parę (u, v) nazywamy *perfekcyjnym cięciem* dla słów A, B jeżeli:

1. (u, v) jest prawidłowym cięciem dla słów A, B; to znaczy jeśli $LCS(A, B)$ możemy przedstawić jako $C = C_1C_2$ wtedy C_1 to $LCS(A(1:u), B(1:v))$ oraz C_2 to $LCS(A(u+1:m), B(v+1:n))$,
2. $W(A(1:u), B(1:v)) = (m-p)/2$, gdzie $W(A, B) = |A| - |C|$

Perfekcyjne cięcie pozwala podzielić zadany problem zgodnie z definicją na dwa podproblemy: *LCS(A(1:u), B(1:v))* oraz *LCS(A(u+1:m), B(v+1:n))*, a z otrzymanych wyników utworzyć rozwiązanie.
Głównym problemem algorytmu jest zatem znalezienie perfekcyjnego cięcia. Do rozwiązania tego problemu wykorzystywane są dwie procedury: *calmid* oraz *fillone*. By lepiej zrozumieć ich ideę wprowadźmy najpierw następujące definicje:

Definicja 1.2. $L_i(k)$ oznacza największe h dla którego słowa *A(i:m), B(h:n)* mają najdłuższy wspólny podciąg długości k.

Definicja 1.3. $L_i^*(k)$ oznacza najmniejsze h dla którego słowa *A(i:m), B(h:n)* mają najdłuższy wspólny podciąg długości k.

W pracy przedstawiona została zależność pomiędzy *perfekcyjnym cięciem (u, v)* a wartościami $L_i(k)$ oraz $L_i^*(k)$. Pozwala ona znaleźć perfekcyjne cięcie słów A, B jeśli zna się wartości jeśli zna się wartości $L_i(k)$ oraz $L_i^*(k)$ dla odpowiednich i, k:

Twierdzenie 1.1. perf-cut Para (u, v) jest *perfekcyjnym cięciem* wtedy i tylko wtedy gdy $L_{u+1}(m-u-w') \geqslant v \geqslant L_u^*(u-w)$, gdzie $w = \lfloor \frac{m-p}{2} \rfloor$ oraz $w' = \lceil \frac{m-p}{2} \rceil$.

Procedura *calmid(A, B, m, n, x, LL, r)* oblicza wartości $L_{m-x+1-j}(j)$ za pomocą procedury *fillone(A, B, m, n, R1, R2, r, s)* i zapisuje je w tablicy LL. Zmienna r po wykonaniu funkcji oznacza natomiast największe j dla którego dana wartość jest poprawnie zdefiniowana. Procedura *fillone(A, B, m, n, R1, R2, r, s)* używa dwóch tablic by obliczyć wartości $L_{s+1}(0), L_s(1), L_{s-1}(2), ...$ Obliczenia te wykonywane są iteracyjne, z wykorzystaniem poprzednich wartości. Poprawność tych funkcji oraz dokładne ich działanie opiszę przy dowodzie ich poprawności.

2 Opis dowodu poprawności oraz złożoności algorytmu

Prezentowany w pracy dowód poprawności algorytmu podzielony został na dowód poprawności dla każdej z procedur zaprezentowanych przez autorów. Na początek przedstawmy dowody poprawności dla procedur *fillone* oraz *calmid* których działanie jest kluczowe dla większości pozostałych funkcji algorytmu. Następnie przyjrzymy się metodzie znajdowania *perfekcyjnego cięcia*. Pozostałe procedury działają w sposób dość jasny do zrozumienia.

2.1 Procedura *fillone*

Tak jak zostało już wspomniane procedura *fillone(A, B, m, n,* R_1, R_2*, r, s)* używa dwóch tablic by obliczyć wartości $L_{s+1}(0), L_s(1), L_{s-1}(2), ...$ Sposób w jaki uzyskiwane są kolejne wartości wynika z następującego lematu:

Lemat 2.1. Niech h będzie największą liczbą taką że $A[i] = B[h]$ oraz $h < l_{i+1}(j-1)$, wtedy liczba $L_i(j)$ jest definiowana przez h oraz $L_{i+1}(j)$:

$$y = \begin{cases} h & \text{gdy } L_{i+1} \text{ jest zdefiniowane} \\ L_{i+1}(j) & \text{gdy } h \text{ nie istnieje} \\ max(h, L_{i+1}(j) & \text{gdy obie wartości są zdefiniowane} \end{cases}$$

W procedurze wykorzystywana jest zmienna lokalna $lower_b$, która odpowiada za to, by nie przekroczyć wartości dla których h oraz $L_{i+1}(j)$ z powyższego lematu dla danego i, j są niezdefiniowane.
Dodatkowo wprowadzona zostaje zmienna $over$, która odpowiada za to by przerwać działanie funkcji gdy wszystkie szukane wartości zostały już obliczone.
Aby uzasadnić liniową złożoność pamięciową funkcji, wystarczy zauważyć że jedyne czego używa procedura *fillone* to liniowej wielkości tablice R_1 oraz R_2, zatem złożoność pamięciowa

to $O(n+m)$. Złożoność czasowa natomiast wynika z tego że maksymalna liczba porównań którą możemy wykonać w pętli *while* to n, ponieważ nigdy nie zwiększamy zmiennej pos_b, zatem złożoność to $O(n)$.

2.2 Procedura *calmid*

Procedura składa się z dwóch części. Pierwsza to pętla for, która w i-tej iteracji odpowiada za wyliczanie wartości $L_{m-i+2-j}(j)$ i zapisanie jej do $R_1[j]$. Przepisywanie wyników z tablicy R_2 do tablicy R_1 na koniec każdej iteracji zapewnia, że będą znajdowały się tam poprawne wartości dla wszystkich $0 \leqslant j \leqslant r$. Poprawność tych wartości wynika z poprawności działania procedury *fillone*
Druga część to przepisanie ostatecznych wartości $L_{m-x-j+1}(j)$ które zapisane są w tablicy $R_1(j)$ do zwracanej tablicy $LL(j)$.

Podobnie jak przy procedurze *fillone* złożoność pamięciowa oraz czasowa są dosyć proste do uzasadnienia. Wykorzystanie pamięci to ponownie jedynie linowego rozmiaru tablice R_1, r_2, LL, zatem złożoność pamięciowa to $O(n+m)$.
Złożoność czasowa to natomiast $(x+1)$ wywołanie procedury *fillone* w pętli for. Przypominając że złożoność procedury *fillone* wynosi $O(n)$, zatem złożoność czasowa procedury *calmid* to $O((x+1)n)$.

2.3 Znajdowanie perfekcyjnego cięcia

Procedura znajdowania *perfekcyjnego cięcia* polega na dwukrotnym wywołaniu procedury calmid odpowiednio dla odwróconych słów A, B oraz dla nieodwróconych. Po zakończeniu tych wywołań otrzymujemy w tablicach LL_1, LL_2 wyliczone wartości odpowiednio: $n+1-L_{m-w+1-k}(k)$ dla $k \leqslant r_1$ oraz $L_{w+k+1}(m-(w+k)-w')$ $(= L_{m-w'+1-(p-k)}(p-k))$ dla $p-k \leqslant r_2$. Z twierdzenia 1.1 wynika zatem, że wystarczy zatem dobrać $u = k+w$ oraz $v = LL_1(k)$ by dostać poprawne perfekcyjne cięcie.
Złożoność czasowa i pamięciowa to ta sama, która miała procedura calmid, gdyż dodatkowo wykonywana jest jedynie pętla *for* by dobrać odpowiednie (u, v), która ma złożoność $O(r_1) \leqslant O(n)$.

2.4 Złożoność czasowa i pamięciowa algorytmu

Patrząc na główną procedurę *Longest common subsequence*, która działa zgodnie z techniką *dziel i zwyciężaj* opisaną na początku dokumentu możemy obliczyć złożoność czasową algorytmu. Mamy do rozpatrzenia dwa przypadki:

- Przypadku bazowego: jego rozwiązanie to wywołanie funkcji *calmid* z parametrem $x = m-p$, gdzie $m-p \leqslant 2$, zatem jest to złożoność $O(n)$ oraz dodatkowe operacje w pętlach *while*, których maksymalnie zostanie wykonanych $O(m)$. Zatem całkowita złożoność przypadku bazowego to $O(n+m)$.

- Znajdowanie *perfekcyjnego cięcia* oraz dwa wywołania rekurencyjne: pierwsze to dwa wywołania procedury *calmid* z parametrem $x = w$ lub $x = w'$ odpowiednio, zatem ich złożoność to $O(n(w+1))$ i $O(n(w'+1))$. Złożoność ta to inaczej $O(n(m-p))$, ponieważ tak dobrany był parametry w i w'. Dodając dwa wywołania rekurencyjne zgodne z *perfekcyjnym cięciem* (u, v) dostajemy złożoność:

$$T(m, n, p) = O(n(m-p)) + T(u, v, u-w) + T(m-u, n-v, m-u-w')$$

która po przeliczeniach okazuje się wynosić $O(n(m-p))$.

Jedyną operacją wykonywaną poza procedurą *Longest common subsequence* jest znalezienie na początku algorytmu długości najdłuższego wspólnego podciągu, która również wykonywana jest w czasie $O(n(m-p))$. Podsumowując całkowita złożoność czasowa algorytmu to:

$$O(n(m-p)) + O(n+m) + O(n(m-p)) = O(n(m-p))$$

Złożoność pamięciowa algorytmu jest natomiast liniowa względem rozmiaru słów A i B, czyli wynosi $O(n+m)$. By uzyskać taką złożoność przy wywołaniach rekurencyjnych zamiast podsłów przekazujemy jedynie indeksy początku i końca podsłowa. Tablice LL_1 oraz LL_2 wykorzystywane do obliczania i przechowywanych wartości mają zawsze wielkość liniową. Maksymalna głębokość rekursji jaka może wystąpić w algorytmie to $O(log(m-p))$, zatem całkowita złożoność pamięciowa algorytmu pozostaje liniowa.