A simple algorithm for Lempel-Ziv factorization

VM

13 czerwca 2022

Rozważamy pewną faktoryzacje dla słowa w, która jest takim rozkładem $u_0u_1...u_k = w$, że każde u_i, za wyjątkiem możliwie ostatniego, jest albo najdłuższym prefiksem $u_iu_{i+1}...u_k$, który występuje jako podsłowo w $u_0u_1...u_i$, ale nie tylko jako sufiks, albo jest pojedynczym symbolem, gdy takiego prefiksu nie ma. Od oryginalnej faktoryzacji Lempel-Ziv'a, ta różni się tym, że w szczególności nie zawiera indeksów początków czynników, niezbędnej do odtworzenia oryginalnego słowa. Tym niemniej, autorzy odnoszą się do tej faktoryzacji pod nazwą faktoryzacji Lempel-Ziv'a.

Authorzy proponują algorytm pozwalający obliczać faktoryzację w czasie liniowym i pamięci $o(n)$. Jeszcze poprzedni wynik tych samych autorów[1] osiągał liniowy czas i pamięć, natomiast różnica pomiędzy dużym $O(n)$ tamtego algorytmu, i małym $o(n)$ dzisiejszego, jest na tyle istotna, że nowy algorytm został opublikowany.

Algorytm ten, tak jak i poprzedni, korzysta z tablicy Longest Previous Factor. Aby zrozumieć co to jest, weźmy dowolne słowo m. Aby m było najdłuższym czynnikiem poprzednim, musi ono być najdłuższym podsłowem słowa $w[1..i+|m|-1]$ spośród wszystkich możliwych prefiksów $w[i..n]$. Wtedy jego długość będzie występować w tablicy LPF na pozycji i-tej.

Gdy już posiadamy tablicę LPF, wyznaczanie faktoryzacji nie jest trudne. Łatwo zauważyć, że "najdłuższy poprzedni czynnik", to prawie dokładnie taki czynnik jakiego potrzebujemy do faktoryzacji. Wystarczy zatem przejść po tablicy LPF zwracając kolejne czynniki, pomijając przy tym czynniki pośrednie, występujące pomiędzy tymi z faktoryzacji. **Algorithm 1** jest implementacją powyższego rozumowania.

Pozostaje wyznaczenie LPF. Do tego korzystamy z tablic SA, i LCP – z uporządkowanej tablicy sufiksów i tablicy najdłuższych prefiksów między nimi. Nie będziemy projektować algorytmów do policzenia tych dwóch tablic,

[1]Praca "Computing Longest Previous Factor in linear time and applications", 2013

Algorithm 1 lempel_ziv_factorization

Require: LPF, n
Ensure: LZ

```
LZ ← [ ]
pos ← 1
while pos ⩽ n do
    push(LPF[pos], LZ)
    pos ← pos + max(1, LPF[pos])
end while
```

gdyż wiele takich istnieje. W szczególności algorytm z pracy "*Constructing suffix arrays in linear time*" autorów D.K. Kim, J.S. Sim, H. Park i K.Park dla SA, oraz "*Two space-saving tricks for linear-time LCP computation*" autora G. Manzini dla LCP.

Zwracamy uwagę na własności tablicy LPF na podstawie których bazuje algorytm jej obliczania.

Lemma 0.1. Wartości tablicy LPF są największymi wpsólnymi prefiksami między elementami tablicy SA.

Dowód. Niech w będzie słowem i SA_i będą kolejnymi tablicami prefiksów słowa w, uzupełnione o -1 na końcu. Czyli $\mathrm{SA}_i = [\mathrm{SA}[k] : \mathrm{SA}[k] \leqslant i] \cup [-1]$. Weźmy dowolne i, oraz indeks x którego wartość jest maksymalna w tablicy SA_i, czyli $x = \arg\max_x \mathrm{SA}_i[x]$. Zauważmy, że dla takiego sufiksu $w[x..n]$, odpowiedni najdłuższy czynnik poprzedni (taki, którego indeks występuje w tablicy LPF) jest największym wspólnym prefiksem tego sufiksu z poprzednim lub z kolejnym sufiksem w tablicy SA. Jest tak dlatego, że to te sufiksy są najbliżej sufiksu $w[x..n]$, zatem mają najdłuższy wspólny z nim prefix. □

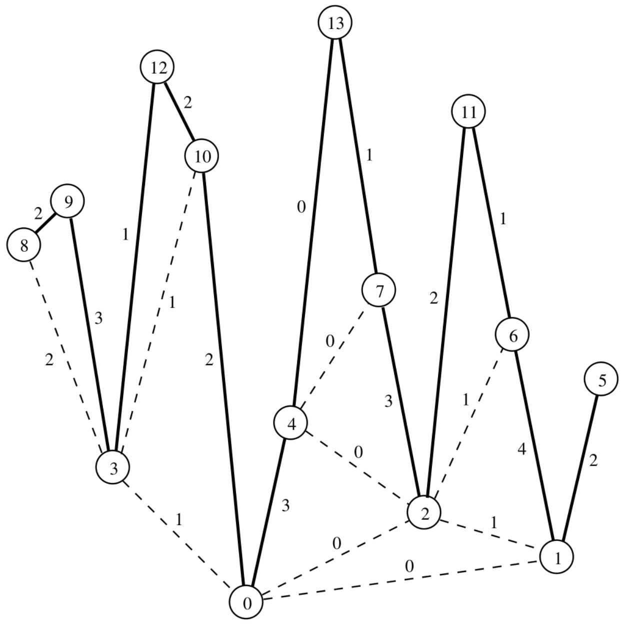
Lemma 0.2. Tablica LPF jest permutacją tablicy LCP.

Dowód. Ponieważ największe wspólne prefiksy pomiędzy sufiksami SA_n to jest po prostu tablica LCP, wystarczy pokazać, że przejście od SA_i do SA_{i-1} zawiera równoważne przejście w tablicy LCP. Czyli, gdy SA_i jest nadzbiorem SA_{i-1}, to LCP_i też jest nadzbiorem LCP_{i-1}. Zauważmy, że najdłuższy wspólny prefiks między elementem i-tym i $(i+2)$-ym jest mniejszym z najdłuższych wspólnych prefiksów między i-tym i $(i+1)$-ym a $(i+1)$-ym i $(i+2)$-im. Zatem przejście definiujemy tak, że dla największego elementu w SA_i, odpowiednie wartości LCP dla jego następnika to minimum wartości następnika i poprzednika maksymalnego elementu w LCP, natomiast wszystkie inne wartości pozostają bez zmian.

□

Mając na uwadzę takie własności, wystarczy przejść po tablicy SA w odpowiedniej kolejności i aktualizować najdłuższe wspólne prefiksy w trakcie.

Aby zobaczyć jak dokładnie bedzie zmieniać się tablica sufiksów i tablica najdłuższych prefiksów, przydatne jest przedstawić ten proces w postaci grafu.



Rysunek 1: Graf ilustrujący zmieniający się stan w algorytmie LPF gdy słowo *abbaabbbaaabab* jest na wejściu.

Rysunek 1 należy odczytywać w następujący sposób:

- wartości w wieszchołkach to są kolejne wartości z tablicy SA,
- wartości przy krawędziach zwykłych to są wartości z tablicy LCP,
- wartości przy krawędziach przerywanych to są wartości tablic LCP_i.

Algorytm obliczający LPF będzie iterować tablicę SA od lewej do prawej, i będzie rozważam takie dwa przypadki:

(1) $\mathrm{SA}[i-1] < \mathrm{SA}[i] > \mathrm{SA}[i+1]$, czyli przypadek lokalnego maksimum. Ustawiamy $\mathrm{LPF}[\mathrm{SA}[i]] = \max(\mathrm{LCP}[i], \mathrm{LCP}[i+1])$, oraz tworzymy nową krawędź zastępującą $\mathrm{LCP}[i], \mathrm{LCP}[i+1]$ o wadze $\min(\mathrm{LCP}[i], \mathrm{LCP}[i+1])$. Jeśli $i = 3$, to na przykładzie z rysunku 1 wartość $\mathrm{LPF}[\mathrm{SA}[3]]$ zostaje

ustawiona na $\max(\mathrm{LCP}[3], \mathrm{LCP}[4]) = \max(1,2) = 2$, wierzchołek o etykiecie 12 zostaje usunięty, i wartość $\mathrm{LCP}[i]$ zostaje ustawiona na $\min(\mathrm{LCP}[3], \mathrm{LCP}[4]) = \min(1,2) = 1$.

(2) $\mathrm{SA}[i-1] < \mathrm{SA}[i] < \mathrm{SA}[i+1] \wedge \mathrm{LCP}[i] \geqslant \mathrm{LCP}[i+1]$. Wykonujemy kroki analogicznie jak w pierwszym przypadku. Jeśli $i = 6$, to na przykładzie z rysunku 1 wartość $\mathrm{LPF}[\mathrm{SA}[6]]$ zostaje ustawiona na $\mathrm{LCP}[6] = 3$, wierzchołek o etykiecie 4 zostaje usunięty, i wartość $\mathrm{LCP}[6]$ zostaje ustawiona na $\mathrm{LCP}[7] = 0$.

Algorytm będzie próbował stosować reguły (1) i (2) na każdym wierzchołku. Gdy nie jest to możliwe, będzie on próbował zastosować je na kolejnych wierzchołkach, aż sytuacja się nie zmieni. Aby zachować odpowiednią kolejność i uniknąć faktycznego tworzenia grafu, utrzymywany będzie stos indeksów do tablic SA i LCP. Kod algorytmu jest przedstawiony niżej.

Algorithm 2 compute_lpf

```
Require: SA, LCP, n
Ensure: LPF
  SA[n + 1] ← -1
  LCP[n + 1] ← 0
  L ← [1]
  for i = 1 to n + 1 do
    while L ≠ ∅ ∧
        (SA[i] < SA[TOP(L)] ∨
        (SA[i] > SA[TOP(L)] ∧ LCP[i] ⩽ LCP[TOP(L)])) do
      if SA[i] < SA[TOP(L)] then
        LPF[SA[TOP(L)]] ← max(LCP[TOP(L)], LCP[i])
        LCP[i] ← min(LCP[TOP(L)], LCP[i])
      else
        LPF[SA[TOP(L)]] ← LCP[TOP(L)]
      end if
      pop(L)
    end while
    if i ⩽ n then
      push(i, L)
    end if
  end for
```

Liniowa złożoność czasowa algorytmu wynika z tego faktu iż każdy indeks jest dokładany na stos co najwyżej raz. Pozostaje pokazanie złożoności pamięciowej algorytmu.

Lemma 0.3. Algorithm 2 używa $o(n)$ pamięci.

Dowód. Zauważmy, że w każdym momencie pozycje na stosie są uporządkowane od największej na jego szczycie. Dodatkowo, odpowiednie wartości SA i LCP też są w takim porządku. Czyli, jeśli wartości na stosie to $i_1 < i_2 < ... < i_k$, to z tego wynika, że $\mathrm{SA}[i_1] < \mathrm{SA}[i_2] < ... < \mathrm{SA}[i_k]$ i $\mathrm{LCP}[i_1] < \mathrm{LCP}[i_2] < ... < \mathrm{LCP}[i_k]$. Rozważmy dowolne $\mathrm{LCP}[i_j]$. Zawiera ono najdłuższy wspólny prefiks sufiksów i_{j-1}-go i i_j-go.

Pokażemy teraz, że $i_{j+2}-i_j \geqslant LCP[i_{j+1}]$. Od tego momentu argumentacja będzie polegać na znanych własnościach napisów w tzw. kombinatoryce napisów, która jest zbyt głęboka, aby powtarzać w tym miejscu. Dla odniesienia, kombinatoryka napisów została wprowadzona w "Algebraic Combinatorics on Words" (2002). Załóżmy, że teza jest falszywa. Wtedy czynnik $w[i_j...i_{j+1} + \mathrm{LCP}[i_{j+1}-1]]$ ma okres o długości $i_{j+1}-i_j$ (dzięki temu, że występuje nakładanie się czynników $w[i_j...i_j+LCP[i_{j+1}]-1]$ i $w[i_{j+1}...i_{j+1}+LCP[i_{j+1}]-1]$). Ponieważ $LCP[i_{j+2}] > LCP[i_{j+1}]$ i $w[i_j...i_j + LCP[i_{j+1}] - 1]$ mają wspólne podsłowo z $w[i_{j+1}...i_{j+1} + LCP[i_{j+1}] - 1]$ o długości co najmniej $i_{j+1} - i_j$, to pierwiaski pierwotne dwóch okresów powinne się synchronizować. W takim razie, okres $i_{j+1} - i_j$ kończy się dalej niż $i_{j+1} + LCP[i_{j+1}] - 1$ w słowie w. Czyli, $LCP[i_{j+1}]$ jest ściśle mniejszy od prawdziwych długości najdłuższych wspólnych prefiksów.

Ponieważ $i_{j+2} - i_j \geqslant \mathrm{LCP}[i_{j+1}]$ i $\mathrm{LCP}[i_{j+1}] > \mathrm{LCP}[i_j]$, to implikuje, że rozmiar stosu, czyli k, jest w obrębie $O(\sqrt{n})$. A to znaczy, że maksymalny rozmiar stosu miejści się w $o(n)$.

□

Proszę zwrócić uwagę na to, że algorytm liczy jedynie faktoryzację. Aby dokonać właściwej kompresji należałoby odzyskać indeksy początków kolejnych czynników. Do tego można spróbować wyznaczyć jawnie permutację między LPF a LCP, natomiast nie jest to łatwo zrobić z uwagi na konstrukcję algorytmu, bo tablica LCP zmienia się w taki sposób, że wartości w niej nie koniecznie odnoszą się do sąsiednich elementów w tablicy SA.