A simple algorithm for Lempel-Ziv factorization

VM

21 maja 2022

Faktoryzacja Lempel-Ziv'a dla słowa w jest takim rozkładem $u_0u_1...u_k = w$, że każde u_i, za wyjątkiem możliwie ostatniego, jest albo najdłuższym prefiksem $u_iu_{i+1}...u_k$ i występuje jako podsłowo w $u_0u_1...u_i$, ale nie tylko jako sufiks, albo jest pojedynczym symbolem, gdy takiego prefiksu nie ma.

Authorzy proponują algorytm pozwalający obliczać faktoryzację w czasie liniowym i pamięci $o(n)$. Jeszcze poprzedni wynik tych samych autorów 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, oraz zamieniając wszystkie zera na jedynki w tablicy LPF, ponieważ faktoryzacja nie zawiera słów pustych. **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, 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.

Algorithm 1 lempel_ziv_factorization

Require: LPF, n
Ensure: LZ
  LZ $\leftarrow$ [ ]
  pos $\leftarrow$ 1
  while pos $\leqslant$ n **do**
    push(max(1, LPF[pos]), LZ)
    pos $\leftarrow$ pos + max(1, LPF[pos])
  end while

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.

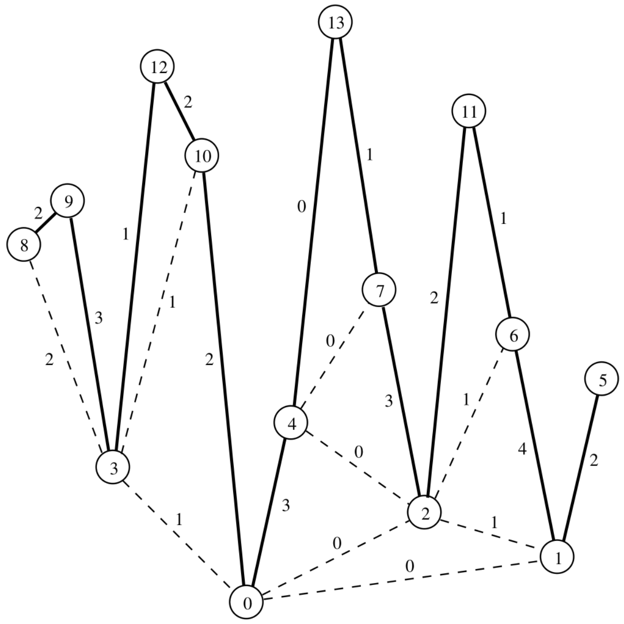
Dowód. Niech w będzie słowem i SA_i będą kolejnymi tablicami prefiksów słowa w, uzupełnione o -1 na końcu. Czyli $\mathrm{SA}_i = [\mathrm{SA}[k] : \mathrm{SA}[k] \leqslant i] \cup [-1]$. Weźmy dowolne i, oraz indeks x którego wartość jest maksymalna w tablicy SA_i, czyli $x = \max_y \{x : \mathrm{SA}_i[x] = y\}$. Zauważmy, że dla takiego sufiksu $w[x..n]$, odpowiedni najdłuższy czynnik poprzedni (w sensie definicji 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 eleementem i-tym i $(i + 2)$-ym jest mniejszy prefiks 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. Trzy przypadki są do rozważenia:

(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 $\min(\mathrm{LCP}[i], \mathrm{LCP}[i{+}1])$.

(2) $\mathrm{SA}[i-1] < \mathrm{SA}[i] < \mathrm{SA}[i+1] \wedge \mathrm{LCP}[i] \geqslant \mathrm{LCP}[i+1]$. Analogicznie do przypadku (1).

(3) $\text{SA}[i-1] > \text{SA}[i] > \text{SA}[i+1] \wedge \text{LCP}[i] \leqslant \text{LCP}[i+1]$. Ponieważ my idziemy od lewej do prawej, to przypadek (1) zapobiega występowaniu takiego scenariusza.

Algorytm będzie próbował stosować reguły (1) i (2) na każdym kroku iteracji. Aby zachować odpowiednią kolejność, 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 $\text{SA}[i_1] < \text{SA}[i_2] < ... < \text{SA}[i_k]$ i

LCP$[i_1]$ < LCP$[i_2]$ < ... < LCP$[i_k]$. Rozważmy dowolne LCP$[i_j]$. Zawiera ono najdłuższy wspólny prefiks sufiksów i_{j-1}-go i i_j-go.

Pokażemy teraz, że $i_{j+2} - i_j \geqslant LCP[i_{j+1}]$. Załóżmy, że tak nie jest. Wtedy czynnik $w[i_j ... i_{j+1}+$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 $w[i_j ... i_j + LCP[i_{j+1}] - 1]$ ma 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 $i_{j+2} - i_j \geqslant$ LCP$[i_{j+1}]$ i LCP$[i_{j+1}]$, to implikuje, że rozmiar stosu, czyli k, jest w obrębie $O(\sqrt{n})$. A to znaczy, że maksymalny rozmiar stosu miejści się w $o(n)$.

□