A simple algorithm for Lempel-Ziv factorization

VM

20 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.

Algorithm 1 lempel_ziv_factorization

Require: LPF, n
Ensure: LZ
  LZ ← [ ]
  pos ← 1
  while pos ⩽ n **do**
    push(max(1, LPF[pos]), LZ)
    pos ← pos + max(1, LPF[pos])
  end while

W szczególności, do policzenia tablicy SA proponowano jest użyć którykolwiek z poniższych algorytmów:

1. "Simple linear work suffix array construction" autorów J. Kärkkäinen i P. Sanders,

2. "Linear-time longest-common-prefix computation in suffix arrays and its applications" autorów T. Kasai, G. Lee, H. Arimura i S. Arikawa,

3. i "Constructing suffix arrays in linear time" autorów D.K. Kim, J.S. Sim, H. Park i K.Park,

pamięciowa złożoność których jest usprawniona w pracy "Space efficient linear time construction of suffix arrays" autorstwa P. Ko i S. Aluru.

W przypadku LCP, algorytm z pracy "Two space-saving tricks for linear-time LCP computation" autora G. Manzini, też jest wystarczająco dobry aby nie wpłynąć na złożoność algorytmu LPF.

Zanim przejdziemy do właściwego obliczania LPF, zauważmy kilka własności związanych z jej wartościami.

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 $SA_i = [SA[k] : SA[k] \leqslant i] \cup [-1]$. Weźmy dowolne i, oraz indeks x którego wartość jest maksymalna w tablicy SA_i, czyli $x = \max_y\{x : SA_i[x] = y\}$. 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. Aby dowieść tezę, 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 też jest.

□