Wee LCP description

krzysztof.michalik

May 2022

1 Wstęp

Praca Johannesa Fishera zatytułowana Wee LCP przedstawia sposób reprezentowania struktury LCP w pamięci wykorzystującą $o(n) = O(\frac{n}{\log\log n})$ bitów, gdzie n jest długością początkowego słowa T dla którego obliczamy LCP.

Autor zakłada dodatkowo, że:

1. Dostępna jest tablica sufiksowa dla słowa T oraz samo to słowo.
2. Dostęp do i-tej litery słowa T zajmuje czas $O(1)$.
3. Dostęp do tablicy sufiksowej jest w czasie $t_A = \Omega(\log n^\epsilon)$, gdzie ϵ jest pewną stałą.

Ostatnie założenie bierze się stąd, że autor powołuje się na nie zwykłe tablice sufiksowe, które zajmują $n\lceil\log n\rceil$ bitów, lecz na ich skompresowaną wersję. [Nie jest podana dokładna liczba bitów zajmowana przez taką reprezentację.]

Autor przedstawia dwa główne wyniki:

1. Dostęp do LCP jest w czasie $O(t_A)$, podczas gdy sama struktura zajmuje $2n + O(\frac{n\log\log n}{\log n})$ bitów w pamięci.
2. Dostęp do LCP jest w czasie $O(t_A + \log(n)^\delta)$, podczas gdy sama struktura zajmuje $o(n) = O(\frac{n}{\log\log n})$ bitów w pamięci.

W obu przypadkach osobno dostępny jest tekst T oraz tablica sufiksowa SUF, a ograniczenia na pamięć dotyczą wyłącznie samego sposobu reprezentowania tablicy LCP.

Warto zwrócić uwagę, że mimo zwiększonego czasu odpowiedzi na zapytanie w drugim z przedstawionych wyników, jest on w pewnym sensie podobny do czasu t_A - oba są postaci $\log n^x$, gdzie w jedym przypadku $x = \epsilon$, a w drugim $x = \delta$. Oznacza to, że odpowiedni dobór δ asymptotycznie doprowadzi do tego samego czasu odpowiedzi na zapytanie do LCP.

2 Oznaczenia

Przez T będziemy oznaczać tekst, który dostajemy na wejściu.

Przez $T[i:]$ będziemy oznaczać sufiks słowa T zaczynający się na pozycji i.

Jeśli nie będziemy znali indeksu na którym zaczyna się sufiks, który jest a-ty leksykograficznie, to przez S_a będziemy oznaczać ten sufiks.

S_A będzie tablicą sufiksową, a dokładniej $S_A[i] = j$ będzie oznaczało, że $T[i :]$ jest j-tym leksykograficznie sufiksem słowa T.

Tablica S_A^{-1} będzie odwrotnością tablicy sufiksowej S_A. Tablica S_A na pozycji i-tej trzymała indeks i-tego leksykograficznie sufiksu $T[j :]$. Zatem tablica S_A^{-1} będzie trzymała na pozycji j-tej informację który leksykograficznie jest prefiks $T[j :]$.

Tablica LCP oznacza najdłuższy wspólny prefiks (Longest Common Prefix) sufiksów, które znajdują się obok siebie w porządku leksykograficznym. Dokładniej $LCP[i] = max_k(T[S_A[i] : S_A[i] + k] == T[S_A[i-1] : S_A[i-1] + k])$. $LCP[0] = 0$, ponieważ pierwszy leksykograficznie sufiks ma zero znaków wspólnych ze słowem pustym.

3 Preprocessing

Zanim osiągniemy skompresowaną wersję tablicy LCP, będziemy musieli skorzystać z kilku innych rezultatów przedstawionych w różnych pracach, aby dostać potrzebne do kompresji dane. Potrzebujemy:

1. Policzyć tablicę sufiksową SUF.
2. Policzyć odwrotność tablicy sufiksowej SUF^{-1}.
3. Policzyć tablicę LCP w postaci nieskompresowanej. Autor powołuje się na algorytm Kasaia, który podał algorytm na obliczanie tablicy LCP w czasie $O(n)$, z adaptacją Manziniego, która pozwala na obliczaniu LCP in-place.

Po otrzymaniu skompresowanej wersji tablicy LCP powyższe struktury można zapomnieć (poza tablicą SUF, która jest nadal wykorzystywana przy odpowiedziach na zapytania zgodnie z założeniami autora.)

4 Kompresja LCP do pamięci $2n + O(\frac{n \log\log n}{\log n})$

Generalna idea polega na tym aby reprezentować LCP jako ciąg zer i jedynek. Liczba wystąpień każdego typu bitu będzie ograniczona przez n, co da składnik $2n$. Dodatkowo będziemy potrzebowali umieć odpowiadać na zapytania postaci $select_1(S, i)$, które zwraca indeks i-tej jedynki w tekście S. Tutaj autor powołuje się na strukturę przedstawioną przez Navarro i M akinena. Pozwala ona odpowiadac na takie zapytania i używa $O(\frac{n \log\log n}{\log n})$ bitów. Razem potrzebujemy $2n + O(\frac{n \log\log n}{\log n})$ bitów.

Zanim przejdziemy do tego jak wygląda taka reprezentacja LCP, przejdziemy krok po kroku przez jej konstrukcję do której będziemy potrzebowali tablicy S_A^{-1} oraz tablicy LCP.

Autor powołuje się na następujące twierdzenie:

Theorem 1 $LCP[S_A^{-1}[i]] \geq LCP[S_A^{-1}[i-1]] - 1$, *dla każdego* $i > 1$.

Można to tez zapisać jako $LCP[S_A^{-1}[i]] - LCP[S_A^{-1}[i-1]] \geq -1$, dla każdego $i > 1$.

Intuicyjnie oznacza to, że dwa sufiksy, które zaczynają się na sąsiednich pozycjach, to krótszy z nich w tablicy LCP może mieć przypisaną wartość o nie więcej niż 1 mniejszą od swojego poprzednika. Rozważmy sufiks $T[i :]$, który jest a-ty leksykograficznie oraz sufiks $T[i+1 :]$, który jest b-ty

leksykograficznie. Wtedy w tablicy LCP są one porównywane z sufiksami S_{a-1} i S_{b-1} odpowiednio. Zauważamy, że $T[i:] = S_a$ i S_{a-1} mają pewne k wspólnych bitów jako prefiks. To oznacza, że S_{a-1} zaczyna się na pewnej j-tej pozycji w słowie (różnej od i) i $T[i : i+k] = T[j : j+k]$. Co więcej wiemy, że $T[i+1 : i+k]$ jest prefiksem dla $T[i+1:]$. Analogicznie $T[j+1 : j+k]$ będzie prefiksem dla sufiksu $T[j+1:]$. Skoro $T[i : i+k] = T[j : j+k]$, to $T[i+1 : i+k] = T[j+1 : j+k]$, zatem jeśli sufiks $T[i+1] = S_c$, to S_{c-1}, ma co najmniej wspólny prefiks z nim długości $k-1$. Warto też zwrócić uwagę, że $T[j+1:]$ znajdzie się w porządku leksykograficznym przed sufiksem $T[i+1]$ z tego samego powodu dla którego $T[j:]$ poprzedza $T[i:]$.

Teraz rozważmy ciąg $a_i = LCP[S_A^{-1}[i]] + i$. Z powyższego twierdzenia wiemy, że $a_i - a_{i-1} \geq i - (i-1) - 1 = 0$, co pokazuje, że ten ciąg jest rosnący. Jego pierwszy element nie może być mniejszy niż 1, ponieważ wartości w tablicy LCP są nieujemne. Możemy też zaobserwować, że $LCP[S_A^{-1}[i]] \leq |T[i:]| \leq n-i+1$. Co więcej $LCP[S_A^{-1}[n]] = 0$, ponieważ jest pojedynczym znakiem i będzie wcześniej alfabetycznie od wszystkich sufiksów zaczynających się na tę samą literę. Z tego możemy wywnioskować, że ciąg (a_i) jest ciągiem rosnącym w przedziale $[1, n]$.

Tego typu struktury można łatwo opisać za pomocą różnicy pomiędzy kolejnymi elementami. Dlatego wyznaczymy tablicę $DIFF[i] = a_i - a_{i-1} = LCP[S_A^{-1}[i]] - LCP[S_A^{-1}[i]] + 1$. Istotna jest następująca obserwacja - jeśli $DIFF[i] = x$, to możemy zapisać $DIFF[i]$ w następujący sposób: 0^x1, gdzie 0^x oznacza konkatenację x zer. Jeśli $DIFF[i] = x$, to przez $DIFF_S[i]$ będziemy oznaczać zapis 0^x1. Teraz rozważmy string bitowy postaci $S = \sum_{i=1}^{n} DIFF_S[i]$ (konkatenacja kolejnych stringów $DIFF_S[i]$).

Teraz możemy poczynić następujące obserwacje na temat S:

1. Liczba bitów '1' w S wynosi dokładnie n, ponieważ każdy element $DIFF_S[i]$ miał w zapisie dokładnie jedną jedynkę.

2. Liczba bitów '0' jest ograniczona od góry przez n z faktu, ze liczba zer oznaczała różnicę pomiędzy kolejnymi elementami ciągu a_i, a suma tych różnic nie mogła przekraczać n, ponieważ ciąg a_i był rosnący i przyjmował jedynie wartości z przedziału $[1, n]$.

Oznacza to, że taki string S zajmuje nie więcej niż $2n$ bitów swoim zapisem i jest to zapis tablicy LCP o który nam chodziło.

W tym momencie autor powołuje się na rezultat z innej pracy - strukturę, która dla danego stringa bitowego pozwala odpowiadać na zapytania:

1. $rank_1(S, i) = a$, gdzie a jest liczbą wystąpień bitu '1' w prefiksie $S[:i]$. Innymi słowy $rank_1$ zlicza wystąpienia '1' w $S[:i]$.

2. $select_1(S, i) = a$, gdzie $S[a]$ ='1' oraz $rank_1(S, i-1) = i-1$. Innymi słowy $select_1$ zwraca indeks i-tego bitu '1' w S.

3. Analogicznie definiowane są $rank_0(S, i)$ oraz $select_0(S, i)$.

Struktura z której korzystają wykorzystuje $O(\frac{n \log\log n}{\log n})$ bitów i pozwala odpowiadać na powyższe zapytania w $O(1)$. Oznacza to, że w całości potrzebujemy $2n + O(\frac{n \log\log n}{\log n})$ bitów, aby pomieścić string S oraz tę strukturę. Jest to dokładnie pamięć o jaką nam chodziło.

Pozostaje pokazać jak wyliczamy $LCP[i]$ korzystając z S, S_A i T. Niech $SUF[j] = i$. Wzory na to są następujące:

1. $LCP[i] = rank_0(S, select_1(S, A[i])) - A[i]$

2. $LCP[i] = select_1(S, A[i]) - 2A[i]$

Pierwszy wzór jest poprawny, ponieważ $select_1(S, A[i])$ zwraca pozycję, gdzie kończy się kodowanie $LCP[S_A[i]]$ w S. Wtedy $rank_0()$ zlicza sumę $DIFF[j]$, dla $j \leq A[i]$. Zatem odjęcie $A[i]$ (które było sztucznie dodane aby zagwarantować monotoniczność ciągu), zwróci rzeczywistą wartość $LCP[i]$.

Drugi wzór jest uproszczeniem pierwszego, ponieważ wiemy ile dokładnie występuje biót '1' w tym podstringu, możemy więc policzyć bity '0' bez odwoływania się do zapytania $rank_0()$.

5 Kompresja LCP do pamięci $O(\frac{n}{\log\log n}) = o(n)$

Zauważmy, że jedyna funkcjonalność wymagana od konstrukcji opisanej w pierwszym wyniku, to odpowiadanie na zapytania postaci $select_1(S, i)$, żeby umieć odpowiedzieć na zapytanie o wartość $LCP[i]$. Oznacza to, że wystarczy zmienić strukturę na zajmującą mniejszą asymptotycznie pamięć, która również będzie potrafić odpowiadać na tego typu zapytania, aby dostać lepszy pamięciowo rezultat.

Autor proponuje trochę 'brutalne' podejście, które sprawia, że pamięć potrzebna zostaje zredukowana do $o(n)$, a sam czas ulega wydłużeniu, ponieważ będziemy musieli za każdym razem wykonać kilka porównań w tekście T. Jednak ze względu na czas potrzebny na wyliczenie $S_A[i]$ (wynoszący A_t ze względu na kompresję), jeśli dobierzemy odpowiednie stałe, czas asymptotycznie pozostaje taki jak poprzednio.

Rozważmy podział S z poprzedniej sekcji na takie przedziały, że ostatnim elementem każdego przedziału będzie bit '1' oraz każdy z przedziałów będzie zawierał $\kappa = \lfloor \log n^2 \rfloor$ bitów '1' (być może poza ostatnim). Oznacza to, ze utworzymy w ten sposób co najwyżej $\frac{n}{\kappa}$ przedziałów.

Teraz będziemy chcieli zapamiętać 2 wartości dla każdego przedziału:

1. $N[i] = select_1(S, i\kappa)$.
2. Jeśli przedział jest dłuższy niż κ^2, czyli $\kappa^2 > N[i] - N[i-1]$, to zhadrcodujemy odpowiedzi na wszystkie możliwe zapytania $select_1()$ na tym przedziale w tablicy P i zapiszemy wskaźnik do niej.
3. Jeśli przedział jest krótszy niż κ^2, to podamy wskaźnik do tablicy N' dla niego, której opis będzie opisany w kolejnym kroku.

Rozważmy analogiczny podział każdego 'krótkiego' ($< \kappa^2$) przedziału na takie przedziały (zwane dalej miniprzedziałami), że ostatnim elementem każdego miniprzedziału będzie bit '1' oraz każdy z miniprzedziałów będzie zawierał $\lambda = \lfloor \log \kappa^2 \rfloor$ bitów '1' (być może poza ostatnim). Oznacza to, ze utworzymy w ten sposób co najwyżej $\frac{n}{\lambda}$ miniprzedziałów (sumując po wszystkich miniprzedziałach utworzonych ze wszystkich 'krótkich' przedziałów).

Analogicznie będziemy chcieli zapamiętać co najwyżej 2 wartości dla każdego miniprzedziału zawierającego się w przedziale P:

1. $N'[i] = select_1(P, i\lambda)$.
2. Jeśli miniprzedział jest dłuższy niż $\log n^\delta$, czyli $\log n^\delta > N'[i] - N'[i-1]$, to zhadrcodujemy odpowiedzi na wszystkie możliwe zapytania $select_1()$ na tym miniprzedziale w tablicy P' i zapiszemy wskaźnik do niej.

3. Jeśli przedział jest krótszy niż $\log n^\delta$, to nie musimy nic zapisywać.

Teraz zastanówmy się jak możemy odpowiadać na zapytania typu $select_1(S, i)$. Mamy kilka dość intuicyjnych przypadków:

1. Długość przedziału w którym jest i-ta jedynka ($N[\lfloor \frac{i}{\kappa} \rfloor + 1] - N[\lfloor \frac{i}{\kappa} \rfloor]$) jest dłuższa niż κ^2 - wtedy nasza odpowiedź jest zhadrcodowana i wystarczy ją odczytać w czasie $O(1)$.

2. Długość przedziału jest mniejsza niż κ^2, ale długość miniprzedziału jest większa od $\log n^\delta$. Wtedy odpowiedź również jest zhardcodowana, jako suma dwóch wartości. Odczytujemy je również w czasie $O(1)$.

3. Długość miniprzedziału jest mniejsza niż $\log n^\delta$. Wtedy nasza wartość nie jest zhardcodowana, ale możemy popatrzyć się na początek miniprzedziału i podać wartość dla niego jako naszą wstępną odpowiedź (wiemy, że aktualna odpowiedź na zapytanie $select_1()$ będzie nie mniejsza). Dokładną wartość poznajemy porównując w czasie $O(\log n^\delta)$ fragmenty T.

Warto jeszcze odpowiedzieć jak używamy tekstu T do dokładnego policzenia $LCP[i]$ z podpunktu nr 3 powyżej. Niech $j = A[i-1]$. Z racji, że $LCP[i] \geq$ wartość zapisana na początku minibloku i wiemy, że nie może wzrosnąć o ponad $\log n^\delta$. Porównujemy zatem sufiksy $T[i+m:]$ oraz $T[j+m:]$ i po co najwyżej $\log n^\delta$ krokach pewien znak nie będzie się zgadzał. Powiedzmy, że pomyłka były przy $y+1$-szym porównaniu. Teraz możemy zakończyć obliczanie wartości i po prostu zwrócić ją $y+m$.

Pozostało pokazać, że pamięć z której korzystamy wynosi $o(n)$.

Policzmy ile miejsca zajmują kolejne tablice, które tworzyliśmy:

1. Pamięć zajmowana przez N to po prostu $O(\frac{n}{\kappa}\log n) = O(\frac{n}{\log n}) = o(n)$, ponieważ zapamiętujemy wartości dla każdego przedziału i każda wartość zapamiętana zajmuje $\log n$ bitów.

2. Pamięć zajmowana przez $P = O(\frac{n}{\kappa^2} * \kappa * \log n) = O(\frac{n}{logn}) = o(n)$, ponieważ długich przedziałów może być nie więcej niż $\frac{n}{\kappa^2}$, w każdym hardcodujemy κ wartości, każda długości $\log n$.

3. Pamięć zajmowana przez wszystkie tablice N' to $O(\frac{n}{\lambda}\log\kappa) = O(\frac{n}{\log\log n}) = o(n)$, ponieważ zapamiętujemy wartości dla każdego przedziału i każda wartość zapamiętana zajmuje $\log\kappa$ bitów.

4. Pamięć zajmowana przez wszystkie tablice $P' = O(\frac{n}{\log n^\delta} * \lambda * \log\kappa) = O(\frac{n\log^3\log n}{logn}) = o(n)$, ponieważ długich miniprzedziałów może być nie więcej niż $\frac{n}{\log n^\delta}$, w każdym hardcodujemy λ wartości, każda długości $\log\kappa$.

Każda z tablic zajmuje nie więcej niż $O(\frac{n}{\log\log n}) = o(n)$ miejsca, zatem wszystkie razem również zajmują $O(\frac{n}{\log\log n}) = o(n)$ miejsca.

W swojej pracy autor rozważa również przypadek gdy tekst jest skompresowany i czas dostępu do niego nie wynosi $O(1)$, lecz wymaga asymptotycznie większego czasu. Korzysta tam jednak głównie z wyników innych prac i nie przedstawia jak wygląda ich implementacja.