Seminarski rad

Sufiksno stablo i sufiksni niz

Učenik: **Janko Šušteršič** *Profesor:* **Zoran Vasiljević**

Sadržaj

Uvod

Jedan od čestih problema u programiranju jeste rad sa stringovima, a među njima pretraživanje stringova i nalaženju određenih delova u okviru njih. Problem lako možemo rešiti u kvadratnoj složenosti, prolaskom kroz stringove i upoređivanjem odgovarajućih karaktera. Međutim problem nastaje kada je potrebno raditi sa stringovima velike dužine, ili je potrebno odgovoriti na određeni broj upita nekog tipa. Za efikasno izvršavanje ovakvih instrukcija neophodno je razviti konkretne alate, odnosno algoritme, koji to mogu uraditi u što manjoj složenosti. Vremenom su se kao najefikasnije nametnule dve strukture podataka za ovu svrhu - sufiksna stabla i sufiksni nizovi. Obe strukture podrazumevaju odgovarajuće skladištenje svih sufiksa datog stringa, uključujući i sam taj string, za njihovu kasniju upotrebu. Pod sufiksom nekog stringa podrazumeva se string koji se dobija odsecanjem prvih i ($0 < i < n$) karaktera datog stringa. U ovom radu biće prikazan princip njihovog rada, konstrukcija, kao i primena sufiksnih nizova i stabala u zadacima.

Sufiksno stablo

Sufiksno stablo predstavlja struktru podataka koja opisuje internu strukturu stringa na vrlo iscrpan način. Može se upotrebiti kako bi se rešio problem *tačnog traženja* u lineranom vremenu (postižući istu složenost u najgorem slučaju kao algoritmi KMP (Knuth-Morris-Pratt) i Bojer-Mur (Boyer-Moore)), ali njegova prednost je u mogućnosti primene u algoritmima linearne složenosti za probleme sa stringovima složenijim od tačnog traženja. Štaviše, sufiksna stabla obezbeđuju most između problema *tačnog traženja* i *približnog traženja.*

Definicija: Problem tačnog traženja je problem nalaženja svih pojavljivanja skupa stringova P u tekstu T, gde je ulaz celokupan skup P.

Definicija: Problem približnog traženja je problem traženja poklapanja stringova iz skupa P u tekstu T, pri čemu su dozvoljene izvesne ograničene različitosti u vidu zamena, umetanja i brisanja, a gde je ulaz celokupan skup.

Definicija: Problem rečnika je specijalni slučaj pretraživanja skupova stringova (koji zajedno čine rečnik) čiji je zadatak da pronađe zadati tekst u rečniku.

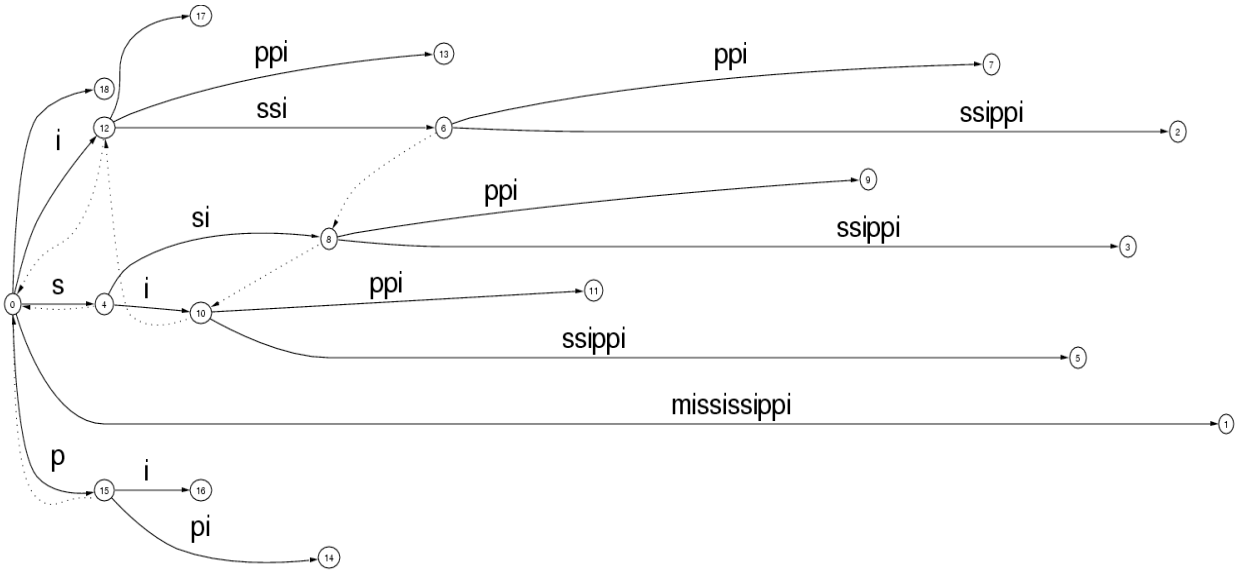
Klasičan primer primene sufiksnih stabala je *problem podstringa*:

Dat je (dugačak) tekst T dužine m. Nakon O(m), odnosno posle linearnog vremena predobrade, moramo da budemo spremni da za svaki učitani string S dužine n za vreme O(n) pronađemo pojavu S u T ili da ustanovimo da S nije sadržan u T. To znači da dozvoljena predobrada traje proprocionalno dužini teksta, ali nakon toga pretraživanje mora da se uradi u vremenu proporcionalnom dužini S, nezavisno od dužine T. Ova složenost se dostiže pomoću sufiksnog stabla. Sufiksno stablo se u fazi predobrade izgrađuje za vreme O(m). Nakon toga kad god dobijemo string dužine n algoritam ga pronalazi u vremenu O(n) koristeći sufiksno stablo.

Ovakvu efikasnost nije moguće postići pomoću KMP ili Bojer-Murovog algoritama. Naime ovi metodi bi najpre predobradili svaki zahtevani string na ulazu, a tada se dobija O(m) u najgorem slučaju za traženje stringa u tekstu. S obzirom na to da m može biti ogroman u poređenju sa n, ovi algoritmi bi bili nepraktični na svim, osim na tekstovima trivijalne veličine.

Definicija: Sufiksno stablo za string S dužine m znakova je korensko orjentisano stablo sa tačno m listova numerisanih od 1 do m. Svaki unutrašnji čvor različit od korena ima najmanje dva sina i svaka grana je označena nepraznim podstringom stringa S. Nikoje dve grane koje izlaze iz čvora ne mogu da imaju oznake grane koje počinju istim znakom. Ključna osobina kod sufiksnih stabala je ta, da za svaki list *i*, konkatenacija oznaka grana na putu od korena do lista *i*, je jednaka sufiksu S koji počinje na poziciji *i*. Odnosno, da je S[i..m].

Posmatrajmo sufiksno stablo za string *mississippi:*



Slika 1. Sufiksno stablo za string *mississippi*

Primetimo da, definicija sufiksnog stabla ne garantuje da sufiksno stablo postoji za svaki string S. Problem je, da ako se sufiks poklapa sa prefiksom drugog sufiksa S, tada ne postoji sufiksno stablo koje zadovoljava datu definiciju jer se put za prvi sufiks ne završava u listu. Da bi se izbegao ovaj problem, pretpostavimo da se poslednji znak iz S ne pojavljuje nigde drugde u stringu. Tada ni jedan sufiks konačnog stringa ne može da bude prefiks ni jednog drugog sufiksa. Da bi se ovo postiglo u praksi, možemo da dodamo jedan znak za kraj stringa, koji nije u alfabetu, recimo znak $.

Implementacija sufiksnog stabla

Najjednostavniji algoritam za konstrukciju sufiksnog stabla ima kvadratnu složenost. Taj algoritam podrazumeva da stablo u početku sadrži samo jednu granu – ceo string S, a zatim, jedan za drugim dodajemo sufikse dužine m-1, m-2, …, 1 tako što u linearnom vremenu tražimo prvi karakter koji se razlikuje od svih prethodnih i tu dodamo granu sa nastavkom sufiksa. Ovaj metod je vrlo jednostavan za implementaciju, a sa druge strane jako nepraktičan za upotrebu zbog svoje velike vremenske složenosti, te mu u ovom radu nećemo posvetiti više pažnje.

Postoji više algoritama za konstrukciju sufiksnih stabala u linearnom vremenu od kojih su najznačajnija dva – Ukonenov i Vajnerov metod. Vajner je bio prvi koji je pokazao da sufiksna stabla mogu da se konstruišu u linearnom vremenu i njegov značaj je pre svega istorijski. S druge strane Ukonenov metod

je jednako brz a u praksi zauzima daleko manje prostora. Stoga se Ukonenov metod često bira kao metod za rešavanje problema koji zahtevaju konstrukciju sufiksnih stabala. Takođe se veruje da je Ukonenov metod lakši za razumevanje, stoga će u ovom radu on biti predstavljen.

Ukenonov algoritam

Definicija: Implicitno sufiksno stablo za string S je stablo dobijeno iz sufiksnog stabla za S$ uklanjanjem svake kopije znaka za kraj $ sa oznake grane stabla, zatim uklanjanjem svake grane koja nema oznaku i najzad uklanjanjem svakog čvora koji nema barem dva sina. Označimo implicitno sufiksno stablo stringa S[1..m] sa $\mathbf{I}_i$ gde *i* ide od 1 do m.

Ukonenov algoritam konstruiše implicitno sufiksno stablo $\mathbf{I}_i$ za svaki prefix S[1..i] od S, počevši od $\mathbf{I}_1$ uvećava za jedan dok stablo $\mathbf{I}_m$ ne bude konstruisano. $\mathbf{I}_{i+1}$ se na osnovu $\mathbf{I}_i$ konstruiše na osnovu sledeća tri pravila:

Pravilo 1: U tekućem stablu, neki sufiks se završava u listu**.** Da bismo unapredili ovo stablo, znak S(i+1) se dodaje na kraj oznake na toj grani sa listom.

Pravilo 2: Ni jedan put od kraja stringa ne počinje znakom S(i+1), ali barem jedan označen put nastavlja od kraja sufiksa. U ovom slučaju, nova grana sa listom koja počinje od kraja mora biti napravljena i označena znakom S(i+1). Novi čvor će takođe morati tamo da bude napravljen ako se sufiks završava unutar grane. Listu na kraju nove grane sa listom je dodeljen broj j.

Pravilo 3: Neki put od kraja sufiksa počinje znakom S(i+1). U ovom slučaju string je već u tekućem stablu, tako da se dalje ne radi ništa.

Naivna implementacija ovih pravila je složenosti $O(m^3)$. Da bi se postigla željena složenost potrebno je izvršiti određena poboljšanja u vidu sufiksnih linkova i Algoritma za pojedinačno produženje (APP).

Definicija: Neka je sa $x\alpha$ označen proizvoljni string, gde x označava pojedinačni znak, a α označava (moguće prazan) podstring. Za unutrašnji čvor sa oznakom puta $x\alpha$, ako je tamo drugi čvor sa oznakom puta α, tada se pokazivač od prvog do drugog čvora naziva *sufiksni link*.

Iako definicija sufiksnih linkova ne povlači da svaki unutrašnji čvor implicitnog sufiksnog stabla ima sufiksni link koji kreće iz njega, zapravo će imati jedan. U Ukonenovom algoritmu, svaki novonapravljeni unutrašnji čvor će imati sufiksni link od sebe do kraja narednog produženja.

Algoritam za pojedinačno produženje se sastoji iz sledećih koraka:

1. Pronalazi se prvi čvor *v* u ili iznad kraja S[j-1..i] koji ili ima sufiksni link ili je koren. Ovo zahteva hod najviše za jednu granu od kraja S[j-1..i] u tekućem stablu. Označimo sa γ (moguće prazan) string između *v* i kraja S[j-1..i].
2. Ako *v* nije koren, obilazi se sufiksni link od *v* do s(*v*) i onda se ide od s(*v*) prateći put za string γ. Ako je *v* koren, tada se prati put za S[j..i] od korena.
3. Korišćenjem pravila za produženje, osigura se da je string S[j..i]S(i+1) u stablu.
4. Ako je napravljen novi unutrašnji čvor *w* u produženju j-1, po drugom pravilu za produženje, napravi se novi sufiksni link.

Definicija: Dubina čvora predstavlja broj čvorova na putu od korena do datog čvora.

Neka je dat bilo koji sufiksni link postavljen za vreme Ukonenovog algoritma. U tom momentu, dubina čvora u kome link počinje je za najviše jedan veća od dubine čvora u kome se link završava.

Koristeći sufiksne linkove u kombinaciji sa algoritmom za pojedinačno produženje, Ukonenov algoritam gradi implicitna sufiksna stabla u celokupnom vremenu O(m). Preostaje samo još da se implicitno sufiksno stablo reda i+1 konvertuje u sufiksno stablo. To se lako postiže dodavanjem znaka $ na kraj stringa i pokretanjem Ukonenovog algoritma za novodobijeni string. Kako nijedan prefiks stringa nije jednak nijednom sufiksu, to dobijeno implicitno stablo po definiciji odgovara sufiksnom stablu stringa S.

Primena sufiksnog stabla

Sufiksnim stablom se mogu rešiti problem tačnog traženja i najdužeg zajedničkog podstringa uzastopnih karaktera u linearnom vremenu.

Problem tačnog traženja se može rešiti na sledeći način:

Formira se sufiksno stablo **T** za tekst T u vremenu O(m). Zatim se uporede znaci reči P uzduž jedinstvenog puta **T** dok se ne iscrpi P ili dok dalja upoređivanja više nisu moguća. U poslednjem slučaju, P se ne pojavljuje nigde u T. U prvom slučaju broj pojavljivanja P u T jednak je težini poslednjeg čvora.

Problem najdužeg zajedničkog podstringa uzastopnih karaktera svodi se na konstrukciju zajedničkog sufiksnog stabla za oba stringa pri čemu se svaki čvor označava sa 1, odnosno 2, ako postoji sufiks stringa 1, tj. 2, koji počinje u datom čvoru, a zatim i pronalaženja najdužeg niza čvorova obeleženih i sa 1 i sa 2. S obzirom da je konstrukcija stabla linearna, kao i da se sam obilazak stabla se obavlja u linearnom vremenu, celokupan algoritam je linearan.

Sufiksni niz

Sufiksni niz i sufiksno stablo su praktično ekvivalentane strukture podataka u pogledu na probleme koje mogu rešiti, sa malom razlikom u tome što je sufiksni niz lakše implementirati, a i zauzima manje memorije nego stablo.

Definicija: Sufiksni niz stringa je niz indeksa svih leksikografski uređenih sufiksa.

Razmotrimo string "abrakadabra" dužine 11 znakova. Ovaj string ima 11 sufiksa: "abrakadabra", "brakadabra", "rakadabra", ..., "a". Sortirani leksikografskim poretkom, ovi sufiksi su:
10: a
7: abra
0: abrakadabra
5: adabra
3: akadabra
8: bra
1: brakadabra
6: dabra
4: kadabra
9: ra
2: rakadabra

Implementacija sufiksnog niza

Najtrivijalnija implementacija sufiksnog niza je reda $O(n^2 \log n)$. Kako je potrebno sortirati sve sufikse dovoljno je u C++ napraviti funkciju koja u linearnom vremenu poredi dva sufiksa koja može izgledati ovako:

```
bool cmp ( int a, int b )
{
   while ( ch[a] == ch[b] )
   {
      if ( a == length ) return 1;
      if ( b == length ) return 0;
      a++; b++;
   }
   return (ch[a]<ch[b]);
}
```

, što u kombinaciji sa sortiranjem daje složenost $O(n^2 \log n)$.

Međutim ovde nismo iskoristili činjenicu da mi ne sortiramo bilo koje stringove, već das u to sve sufiksi jednog stringa, te ćemo u tom pravcu vršiti poboljšanja.

Složenost $O(n \log^2 n)$

Gore pomenutu činjenicu ćemo iskoristiti na sledeći način:

1. Prvo, sufikse poredimo tako što poredimo prva dva njihova karaktera.
2. Zatim sufikse poredimo tako što uporedjujemo prva četiri njihova karaktera. Tu nastupa činjenica da već postoje sufiksi koje smo uporedili posmatrajući prva dva karaktera takva das u to druga dva karaktera sufiksa u drugom poređenju, stoga nije potrebno prolaziti kroz ceo string već samo iskoristiti prethodno izračunate podatke. Time se postiže složenost $O(n \log n)$ u najboljem i $O(n \log^2 n)$ u najgorem slučaju.

Jedan od načina da se implementira navedena ideja:

```
void buildSA ( )
{
    for ( int i = 0; i < n; i++ )
    {
        SA[ i ] = i;
        poz[ i ] = s[ i ];
    }
    for ( t = 1; ; t*=2 )
    {
        sort( SA, SA + n, suff_compare );
        for ( int i = 0;  i < n - 1; i++) tmp[ i + 1] = tmp[ i ] + suff_compare( SA[ i ], SA[ i + 1 ] );
        for ( int i = 0; i < n; i++ ) poz[ SA[ i ] ] = tmp[ i ];
        if (tmp[ n – 1 ] == n - 1) break;
    }
}
```


gde je:

```
bool suff_compare(int i, int j)
{
   if (poz[i] != poz[j]) return (poz[i] < poz[j]);
   i += t;  j += t;
   if (i < n && j < n) return (poz[i] < poz[j]);
   else return (i > j);
}
```

Prednost ove ideje jeste to što je laka za razumevanje i najjednostavnija za implementaciju i praktičnu upotrebu.

Složenost O(n)

Problem konstruisanja sufiksnog niza u linearnom vremenu je sličan prethodnom s tim što se koristi 2/3 rekurzija umesto ½, što se postiže u tri koraka:

1. Konstruiše se sufiksni niz za sufikse koji počinju na pozicijama i, takvim da je i mod 3 <> 0. Ovo je urađeno rekurzijom, odnosno svođenjem na konstrukciju sufiksnog niza za niz dužine jednake 2/3 dužine polaznog stringa.
2. Konstruiše se sufiksni niz od preostalih sufiksa koristeći rezultat prvog koraka.
3. Dva sufiksna niza se spajaju u jedan.

Prvi korak se implementira tako što se string podeli na delove od po 3 karaktera tako što se za početne karaktere uzmu oni čiji indeks daje ostatak pri deljenju sa 3 različit od 0, a zatim se sortiraju radix sortom. Stringovi koj su isti uporedjuju se na osnovu njihovog nastavka čije je uporedjivanje već izvršeno.

Iznenađujuće, upotreba 2/3 umesto 1/2 sufiksa u prvom koraku čini poslednji korak gotovo trivijalnim: naime, dovoljno je jednostavno objedinjavanje zasnovano na upoređivanjima. Na primer za upoređivanje sufiksa Si i Sj, takvih da je i mod 3 = 0 i j mod 3 = 1, najpre se upoređuju prvi znaci, pa ako su isti, upoređuju se sufiksi Si+1 i Sj+1, čiji je relativni poredak već poznat iz prvog koraka.

Primena sufiksnog niza

Kao što smo rekli sufiksni nizovi se mogu koristiti pri obradi dugačkih stringova upotpunosti zamenjujući sufiksna stabla jer zauzimaju manje memorije i jednostavniji su za implementaciju. Sufiksni niz kao struktura sama po sebi nema neku konkretnu upotrebu već se ona može menjati od primera do primera što se može videti u zadacima koji će uslediti.

Cyclical Quest

izvor: http://codeforces.com/problemset/problem/235/C

Tekst zadatka

Dat je string s, $|s|<10^6$ i broj $n<10^5$ koji označava broj stringova x_i koji se unose u svaki u novom redu. Za svaki string x_i ispisati broj uzastopnih podstringova stringa s koji su ciklično izomorfni stringu x_i.

Dva stringa su ciklično izomorfna ako se jedan može dobiti od drugog jednim sečenjem stringa na dva dela i spajanjem njegovih krajeva. Npr. string “deabc” se može dobiti od “abcde” sečenjem na “abc” i “de” i spajanjem u “deabc”.

Svi stringovi se sastoje iz malih slova engleskog alfabeta.

Input	Output
baabaabaaa	7
5	5
a	7
ba	3
baa	5
aabaa	
aaba	

Little Elephant and Strings

izvor: http://codeforces.com/problemset/problem/204/E

Tekst zadatka

Dat je niz stringova od n elemenata $n<10^5$, a_i ukupne dužine do 10^5 i $k<10^5$. Za svaki string odrediti broj parova brojeva l i r takvih da je $a_i[l..r]$ podstring najmanje k drugih stringova.

Svi stringovi se sastoje iz malih slova engleskog alfabeta.

Input
7 4
rubik
furik
abab
baba
aaabbbababa
abababababa
zero

Output
1 0 9 9 21 30 0

LCP niz

LCP niz ili Longest Common Prefix Array je niz usko povezan sa sufiksnim, jer on sadrži dužinu najdužeg zajedničkog prefiksa dva uzastopna sufiksa. Implementacija je trivijalna ukoliko smo na prethodni način izgradili sufiksni niz. Složenost datog algoritma je O(n).

```
void buildLCP()
{
   for (int i=0, k=0; i<n; i++)
   {
      if (poz[i] != n-1)
      {
         int j = SA[poz[i]+1];
         while (s[i+k] == s[j+k]) k++;
         LCP[poz[i]] = k;
         if (k) k--;
      }
   }
}
```

Fence

izvor: http://codeforces.com/problemset/problem/232/D

Tekst zadatka

Data je ograda dužine n, $n<10^5$, napravljena od lestvica širine 1 i visine $h_i<10^9$. Deo ograde od l do r je niz brojeva h_i čiji su indeksi od l do r, uključujući i njih. Dva dela ograde iste dužine se zovu spajanje, ako važi $h_{l1+i} + h_{l2+i} = h_{l1} + h_{l2}$ i njihov presek je prazan skup. Za $q \le 10^5$ upita tipa l_i i r_i ($1 \le l_i \le r_i \le n$) odgovoriti koliko različitih spajanja postoji za taj deo ograde.

Input
```
10
1 2 2 1 100 99 99 100 100 100
6
1 4
1 2
3 4
1 5
9 10
10 10
```

Output
```
1
2
2
0
2
9
```