Matematički fakultet Univerziteta u Beogradu

Master rad

Sufiksno stablo i sufiksni niz

Kandidat: Ivan Ječmen
Mentor: prof. Miodrag Živković

Sadržaj

1. Uvod **5**

2. Linearni algoritmi za konstrukciju sufiksnih stabala **7**

2.1 Uvod *8*

2.2 Osnovne definicije *8*

2.3 Motivišući primer *10*

2.4 Jednostavan algoritam za konstrukciju sufiksnog stabla *12*

2.5 Konstrukcije sufiksnih stabala u linearnom vremenu *12*

2.5.1 Implicitna sufiksna stabla 13

2.5.2 Ukonenov algoritam – osnovna struktura 14

2.5.3 Implementacija i ubrzavanje 17

2.5.4 Važno pojednostavljenje implementacije 23

2.5.5 Još dva pojednostavljenja 25

2.5.6 Konverzija implicitnog u sufiksno stablo 27

2.5.7 Implementacija Ukonenovog algoritma 28

2.6 Vajnerov linearni algoritam *31*

2.6.1 Direktna konstrukcija 31

2.6.2 Popravke efiksanosti 32

2.6.3 Osnovna ideja Vajnerovog algoritma 34

2.6.4 Kompletan algoritam za konstrukciju T_i od T_{i+1} 36

2.6.5 Analiza složenosti Vajnerovog algoritma 38

3. Primene sufiksnih stabala **41**

3.1 Tačno traženje stringova *41*

3.2 Sufiksna stabla i problem tačnog traženja skupa uzoraka *41*

3.3 Problem podstringa za bazu podataka uzoraka *42*

3.4 Najduži zajednički podstring dva stringa *43*

4. Linearni algoritmi za konstrukciju sufiksnih nizova **45**

4.1 Uvod *45*

4.2 Oznake *46*

4.3 Algoritam DC3 linearne vremenske složenosti *47*

4.4 Implementacija algoritma DC3 ..... *48*

5. Primene sufiksnih nizova **53**

5.1 Motivacija – primene u analizi genoma *53*

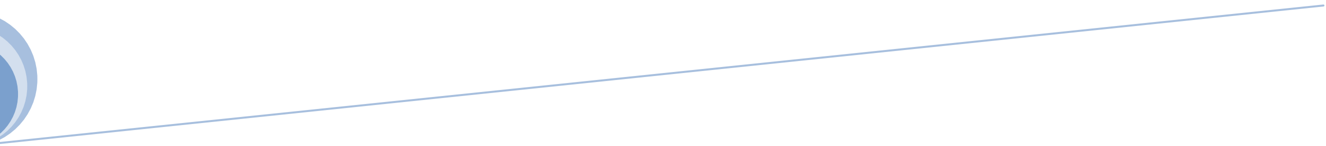
5.2 Osnovne napomene *54*

5.3 Lcp intervalno stablo sufiksnog niza *56*

5.4 Efikasna implementacija algoritma za nalaženje maksimalnih ponavljanja *59*

6. Zaključak i pravci daljeg rada **63**

7. Literatura 65



1. Uvod

Jedan od najobimnijih projekata u biologiji predstavlja proučavanje genoma živih vrsta. Biolozi su počeli da dobijaju zapaženije rezultate na ovom polju tek nakon uvođenja kompjutera u rad prilikom njihovog istraživanja, a njihovi rezultati su se poboljšavali uporedo sa razvojem bioinformatike kao posebne grane informatike. Proučavanje genoma je u najvećoj meri zasnovano na pretraživanju stringova i nalaženju određenih uzoraka u okviru njih. Samim tim jedan od osnovnih imperativa bioinformatike je bio razvoj konkretnih alata odnosno algoritama za pretraživanje stringova.

Vremenom su se kao najefikasnije nametnule dve strukture podataka za ovu svrhu - sufiksna stabla i sufiksni nizovi. Neumoljiva žeđ naučnika za kompletiranjem znanja o genomima živih vrsta, a samim tim i pronicanje u tajne života je bila glavna pokretačka sila za razvoj sve efikasnijih algoritama za rad sa stringovima. Ti algoritmi su dalje poslužili kao osnova za dalji razvoj modernih softverskih aplikacija koji savremenim biolozima omogućuju neuporedivo brže i lakše istraživanje genoma.

Tema rada koji je pred vama su osnovni algoritmi za konstrukciju sufiksnih stabala i sufiksnih nizova i primene ovih struktura podataka. Sam rad je organizovan na sledeći način. Druga glava rada daje pregled i detaljno razmatra osnovne algoritme za konstrukciju sufiksnih stabala. Literatura na osnovu koje je ova glava pisana je knjiga D. Gusfield: Algorithms on Strings, Trees and Sequences [1]. Treća glava razmatra neke od primena sufiksnih stabala, takođe na osnovu [1]. Četvrta glava razmatra algoritam DC3 [2] za konstrukciju sufiksnih nizova. Naime do 2003-će godine izgradnja sufiksnih nizova u linearnom vremenu je bila moguća samo polazeći od sufiksnog stabla. Te godine objavljena su tri različita algoritma linearne složenosti za konstrukciju sufiksnog niza. Peta glava sadrži prikaz nekih od primena sufiksnih nizova na osnovu rada [3].

2. Linearni algoritmi za konstrukciju sufiksnih stabala

Sufiksno stablo predstavlja struktru podataka koja opisuje internu strukturu stringa (niske) na vrlo iscrpan način. Sufiksno stablo može da se upotrebi da bi se rešio problem *tačnog traženja* za linerano vreme (postižući istu složenost u najgorem slučaju koju su postigli algoritmi KMP (Knuth-Morris-Pratt) i Bojer-Mur (Boyer-Moore), ali njihova 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 $\mathrm{O}(m)$, odnosno posle linernog vremena predobrade, moramo da budemo spremni da za svaki učitani string S dužine n za vreme $\mathrm{O}(n)$ 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 S 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 $\mathrm{O}(m)$. Nakon toga kad god dobijemo string dužine $\mathrm{O}(n)$ algoritam ga pronalazi u vremenu $\mathrm{O}(n)$ koristeći sufiksno stablo.

Dobijanje rezultata nakon $\mathrm{O}(m)$ vremena utrošenog na predobradu i $\mathrm{O}(n)$ vremena utrošenog na pretraživanje kod problema podstringa je veoma korisno. U tipičnim aplikacijama nakon što je izgrađeno sufiksno stablo, na ulazu imamo dugi niz zahtevanih stringova, tako da je vremenski nivo složenosti za svako pretraživanje jako važan. 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 $\Theta(m)$ (najgori slučaj) 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.

STS (Sequence-tagged-site) je DNK string dužine 200-300 nukleotida, koji se u čitavom genomu pojavljuje samo jednom.
EST (Expressed sequence tag) je STS koji je deo gena, a ne deo intergenetskog DNK.

Tekst često biva skup stringova, (na primer skupovi STS i EST) tako da je problem podstringa da odredi da li je string na ulazu podstring nekog od fiksiranih stringova. Sufiksna stabla takođe efikasno rešavaju ovaj problem. Površno gledajući, slučaj višestrukih tekstualnih stringova liči na problem rečnika koji se razmatra u kontekstu Aho-Korasikovog (Aho-Corasick) algoritma [1]. Stoga je prirodno očekivati da bi ovde mogao da se primeni Aho-Korasikov algoritam. Međutim

ovaj metod ne rešava problem podstringa u željenom vremenskom nivou složenosti jer on samo određuje da li je novi string kao celina u rečniku, a ne da li je podstring stringa u rečniku.

2.1 Uvod

Autor prvog algoritma za konstrukciju sufiksnih stabala linearne vremenske složenosti je Vajner (Weiner) 1973 godine, pri čemu je on za to stablo koristio termin *stablo pozicija*. Drugačiji i što se tiče štednje prostora prilikom gradnje sufiksnih stabala bolji algoritam, dao je MekKrejt (McCraight) nekoliko godina kasnije. Nakon toga, Ukonen (Ukkonen) je razvio konceptualno drugačiji linerani algoritam za izgradnju sufiksnih stabala koji ima sve prednosti MekKrejtovog algoritma (a detaljnija analiza pokazuje da ga se može smatrati varijantom MekKrejtovog algoritma), ali nudi mnogo jednostavnije objašnjenje.

Iako je prošlo više od 35 godina od objavljivanja Vajnerovog originalnog rada (koji je Knut (Knuth) nazivao ″algoritam iz 1973″), sufiksna stabla nisu ušla u glavne tokove nastave o računarskoj tehnologiji i generalno se malo primenjuju. Ovo je verovatno zbog toga što su ova dva originalna rada iz sedamdesetih godina imala reputaciju teško razumljivih. Ta reputacija je itekako zaslužena, ali su oni ipak dosta nesretno prošli jer ovi algoritmi iako netrivijani, nisu mnogo komplikovaniji nego mnogi drugi koji su široko predavani. Kada su dobro implementirani ovi algoritmi su praktični i omogućavaju efikasna rešenja za mnoge probleme vezane za stringove. Trenutno se u osnovi ne zna ni jedna druga struktura podataka (osim onih koji su ekvivalentni sufiksnim stablima) koje omogućavaju efikasna rešenja tako širokog spektra kompleksnih problema vezanih za stringove.

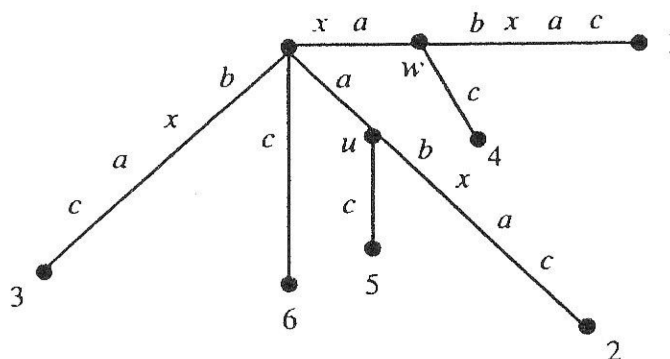
U daljem tekstu će biti prikazani Ukonenov i Vajnerov algoritam (složenosti čak $O(n^3)$). Tom prilikom će algoritmi biti predstavljeni na višem nivou posle čega će biti opisane proste i efikasne implementacije. Te implementacije će zatim biti usavršene da bi algoritam dostigao linearnu složenost.

2.2 Osnovne definicije

Prilikom opisa kako se gradi sufiksno stablo, za string koji se razmatra, koristi se osnovni string S dužine m. Ne koristi se P ili T (koji označavaju uzorak i tekst) jer su se sufiksna stabla koristila u širokom spektru primena gde je ulazni string ponekad igrao ulogu uzorka, ponekad teksta, ponekad oba a ponekad ni jednog. Pretpostavlja se da je alfabet konačan i poznat.

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 za 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 **T** za string $xabxac$. Put od korena do lista c numerisanog brojem 1 jednak je celom stringu $S = xabxac$, dok put do lista c numerisanog brojem 5 jednak je sufiksu ac, koji u S počinje na poziciji 5.



Slika 2.1 Sufiksno stablo za string xabxac. Oznake čvorova u i w za dva unutrašnja čvora biće iskorištena u nastavku teksta.

Primetimo da, definicija sufiksnog stabla ne garantuje da sufiksno stablo postoji za svaki string S. Problem je, da ako se sufiks S poklapa sa prefiksom drugog sufiksa S, tada ne postoji sufiksno stablo koje zadovoljava datu definiciju jer put za prvi sufiks ne završava u listu. Na primer, ako se poslednji znak iz $xabxac$ ukloni, time dobijamo string $xabxa$. Stoga put koji ispisuje string xa ne bi završio u listu.

Da bi se izbegao ovaj problem, pretpostavimo da se poslednji znak iz S ne pojavljuje nigde drugde u S. 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 S, koji nije u alfabetu. Ovde koristimo znak $ kao znak za kraj. Kada bude važno da istaknemo činjenicu da je ovaj znak za kraj dodat, pisaćemo to eksplicitno kao $S\$$. Pretežno ovo podsećanje neće biti neophodno, osim ukoliko nije eksplicitno naznačeno drugačije, za svaki string S se pretpostavlja da je proširen ovim znakom za kraj $, čak i ako znak nije eksplicitno prikazan.

Sufiksno stablo je srodno sa *stablom ključnih reči* (bez povratnih pokazivača). Za dati string S ako je skup P definisan tako da bude m sufiksa od S, tada sufiksno stablo od S može da se dobije iz stabla ključnih reči za P spajanjem bilo kog puta čvorova koji se ne granaju u pojedinačnu granu. Postoji prost algoritam za izgradnju stabla ključnih reči koji može da se iskoristi za konstruisanje sufiksnog stabla za S u vremenu $O(m^2)$ radije nego nivoa složenosti $O(m)$ koji ćemo uspostaviti.

Definicija: Stablo ključnih reči za skup P je korensko usmereno stablo **K** koje zadovoljava tri uslova:
1. Svaka grana je označena tačno jednim znakom;
2. Svake dve grane iz istog čvora imaju različite oznake;
3. Svaki uzorak P_i u skupu P vodi put do nekog čvora v iz **K** takvog da znaci na ovom putu od korena **K** do v tačno ispisuju P_i, i svaki list iz **K** je označen tako da bude neki uzorak u P.

Definicija: Oznaka puta od korena do nekog čvora je konkatenacija oznaka grana ovog puta. Oznaka puta čvora je oznaka puta od korena **T** do tog čvora.

Definicija: Za svaki čvor *v* u sufiksnom stablu, težinska dubina stringa za *v* je broj znakova u oznaci za *v*.

Definicija: Put koji se završava na sredini grane (u, v) deli oznaku za (u, v) u ustanovljenoj tačci. Definišimo oznaku takvog puta kao oznaku za u povezanu sa znacima na grani (u, v) na dole do ustanovljene tačke razdvajanja.

Na primer na slici 2.1 string xa označava unutrašnji čvor w (tako da čvor w ima oznaku puta xa), string *a* označava čvor u, dok string $xabx$ označava put koji se završava unutar grane $(w, 1)$ odnosno, unutar grane listova dodiruje list *1*.

2.3 Motivišući primer

Pre nego što uđemo u detalje metoda za konstrukciju sufiksnih stabala, pogledajmo kako se sufiksno stablo stringa koristi za rešavanje problema tačnog traženja.

Zadat je naredni problem: Dat je uzorak P dužine n i tekst T dužine m. Pronaći sva pojavljivanja P u T za vreme $O(n + m)$.
Već su viđena nekoliko rešenja ovog problema. Sufiksna stabla omogućuju naredni pristup:

Formira se sufiksno stablo **T** za tekst T u vremenu $O(m)$. Zatim se uporede znaci 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, svaki list u podstablu ispod tačke poslednjeg upoređivanja je numerisan startnom lokacijom P u T i svaka startna lokacija P u T numeriše jedan takav list.

Ključ za razumevanje prethodnog slučaja (kada se celi P poklapa sa delom T) je da primetimo da se P pojavljuje u T na startnoj poziciji j, ako i samo ako se P pojavljuje kao prefiks stringa $T[j..m]$. Međutim, to se dešava ako i samo ako string P označava inicijalni deo puta od korena do lista j. To je početni put koji prati algoritam upoređivanja.

Put upoređivanja je jedinstven, zato što ni jedne dve grane iz poznatog čvora ne mogu imati oznake grana koje počinju istim znakom. I pošto smo uspostavili konačan alfabet, rad na svakom čvoru trajaće konstantno vreme tako da je vreme potrebno da bi se P uporedio sa putem, proporcionalno dužini P.

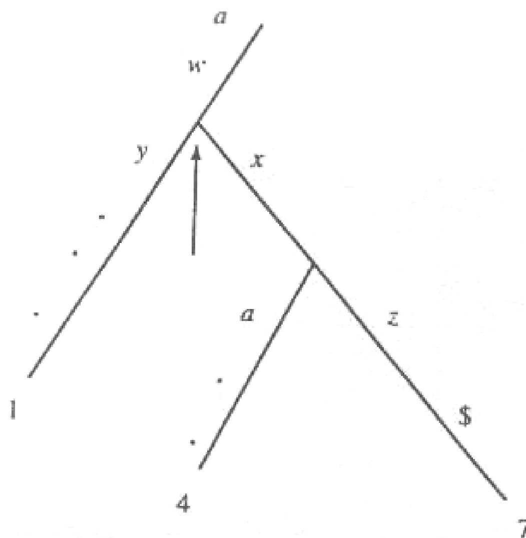
Na primer slika 2.2 pokazuje fragment sufiksnog stabla za string $T = awyawxawxz$. Uzorak P = aw se pojavljuje tri puta u T, i to na lokacijama 1, 4 i 7.

Ako se P u potpunosti poklopi sa nekim putem u stablu, algoritam može da nađe sve startne pozicije P u T obilaskom podstabla ispod kraja puta upoređivanja, prikupljajući pri tome brojeve pozicija zapisanih na listovima. Sva pojavljivanja P u T stoga mogu da se nađu u vremenu $O(n + m)$. Ova ocena složenosti ista je kao za nekoliko drugih algoritama spomenutih u uvodu, ali je raspodela rada drugačija. Prethodni algoritmi najpre potroše $O(n)$ vremena za predobradu P a zatim $O(m)$ vremena za pretraživanje. Nasuprot tome, pristupom sufiksnog stabla potroši se

$O(m)$ vremena za predobradu a zatim se potroši $O(n+k)$ vremena za pretraživanje, gde je k broj pojavljivanja P u **T**.
Da bi smo prikupili k startnih pozicija za P, obilazimo podstablo na kraju puta upoređivanja koristeći bilo koji obilazak u linearnom vremenu (recimo pretraga u dubinu) i popisujemo brojeve listova koje pri tom sretnemo. S obzirom na to da svaki unutrašnji čvor ima barem dva sina, broj listova na koji se nailazi je proporcionalan broju obiđenih grana, tako da je vreme obilaska $O(k)$, iako krajnja *težinska dubina stringa* tih $O(k)$ grana može biti proizvoljno veća od k.

Definicija: Težinska dubina stringa predstavlja broj znakova na putu između dva određena čvora.

Ako se traži samo jedno pojavljivanje P a predobrada se malo produži, tada vreme pretraživanja može biti smanjeno sa $O(n+k)$ na $O(n)$. Ideja je da se piše po jedan broj na svakom čvoru (recimo najmanjem listu) u njegovom podstablu. Ovo može da se postigne u vremenu $O(m)$ u fazi predobrade pretragom u dubinu stabla **T**. Tada u fazi pretraživanja, broj napisan na čvoru, na ili ispod kraja poklapanja daje jednu startnu poziciju P u **T**.
Kasnije će se u odeljku o primenama sufiksnih stabala ponovo razmatrati relativne prednosti metoda koji predobrađuju tekst nasuprot metodima koji predobrađuju uzorak(e). U istom ovom odeljku će takođe biti pokazano kako rešiti *problem tačnog traženja* trošeći $O(n)$ vremena za predobradu i $O(m)$ vremena za pretraživanje, postižući isti nivo složenosti kao i ranije predstavljeni algoritmi u uvodu.



Slika 2.2 Tri pojavljivanja aw u awyawxawxz. Njihova početna pozicija numeriše čvorove u podstablu čvora sa oznakom puta aw.

2.4 Jednostavan algoritam za konstrukciju sufiksnog stabla

Zarad učvršćivanja definicije sufiksnog stabla i razvoja intuicije, ovde je predstavljen direktan algoritam za izgradnju sufiksnog stabla za string S. Ovaj uprošćeni metod prvo unosi jednu granu za sufiks $S[1..m]\$$ (kompletan string) u stablo; nakon toga sukcesivno unosi sufiks $S[1..m]\$$ u rastuće stablo, za 2,...,m. Neka $\mathbf{N_i}$ označava stablo koje kodira sufikse od 1 do i.
Detaljnije, stablo $\mathbf{N_1}$ se sastoji od jedne grane između korena stabla i lista označenog brojem 1. Grana je označena stringom $S\$$. Stablo $\mathbf{N_{i+1}}$ se konstruiše iz $\mathbf{N_i}$ na sledeći način: Polazeći od korena $\mathbf{N_i}$ pronađe se najduži put od korena, čija se oznaka poklapa sa prefiksom $S[i+1..m]\$$. Ovaj put je pronađen uzastopno upoređujući i poklapajući znake u sufiksu $S[i+1..m]\$$ sa znacima duž jedinstvenog puta od korena, sve dok dalja poklapanja više nisu moguća. Put poklapanja je jedinstven jer ni jedne dve grane izlazeće iz čvora ne mogu imati oznake koje počinju istim znakom. Istovremeno dalja poklapanja nisu moguća jer nema sufiksa od $S\$$ koji je prefiks nekog drugog sufiksa od $S\$$. Kada je ta tačka dostignuta, algoritam se nalazi ili u čvoru (recimo w), ili je na sredini grane. Ako se nalazi na sredini grane, recimo (u,v) tada razbija granu (u,v) na dve grane umetanjem novog čvora imenovanog sa w neposredno nakon poslednjeg znaka na grani koji se poklapao sa znakom u $S[i+1..m]$ i neposredno pre prvog znaka koji se nije poklapao. Nova grana (u,w) je označena delom oznake (u,v) koji se poklopio sa $S[i+1..m]$ a nova grana (w,v) je obeležena preostalim delom oznake (u,v). Tada bez obzira na to da li je novi čvor w bio napravljen ili takav već postoji u trenutku kada se poklapanje završilo, algoritam pravi novu granu $(w,i+1)$ počevši od w do novog lista označenog sa $i+1$ i označava novu granu sa nepoklopljenim delom sufiksa $S[i+1..m]\$$.
Stablo sada sadrži jedinstveni put od korena do lista $i+1$ i ovaj put ima oznaku $S[i+1..m]\$$. Međutim sve grane iz novog čvora w imaju oznake koje počinju različitim prvim znakom tako da induktivno sledi da nikoje dve grane koje izlaze iz čvora nemaju oznake koje počinju istim prvim znakom.
Pretpostavlja se kao i obično da je alfabet ograničene veličine, tako da jednostavan metod troši O(m^2) vremena prilikom konstrukcije sufiksnog stabla za string S dužine m.

2.5 Konstrukcije sufiksnih stabala u linearnom vremenu

U ovom odeljku će detaljno biti predstavljena dva metoda za konstrukciju sufiksnih stabala – Ukonenov metod i Vajnerov metod. Vajner je bio prvi koji je pokazao da sufiksna stabla mogu da se konstruišu u linearnom vremenu i njegov metod je ovde prezentovan iz dva razloga - zbog istorijskog značaja i zbog nekoliko različitih tehničkih ideja koje on sadrži. S druge strane Ukonenov metod je jednako brz a u praksi zauzima daleko manje prostora (na primer memorije) nego Vajnerov metod. 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 biti predstavljen prvi. Međutim s obzirom na to da razvoj Vajnerovog metoda ne zavisi od razumevanja Ukonenovog metoda, ova dva algoritma mogu da se razmatraju odvojeno (sa jednim zajedničkim delom navedenim u opisu Vajnerovog metoda).

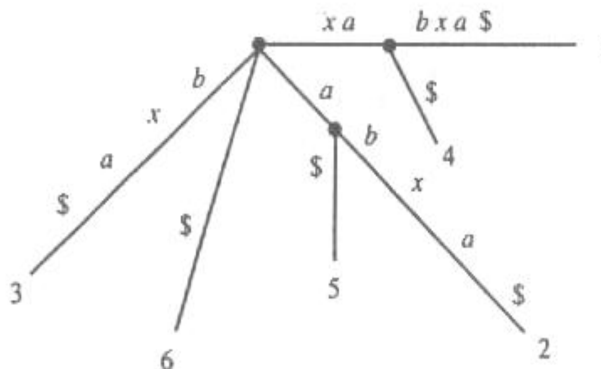
Ukonenovo sufiksno stablo u linearnom vremenu

Esko Ukonen (Esko Ukkonen) je napravio algoritam za konstrukciju sufiksnog stabla u linearnom vremenu koji je verovatno konceptualno najlakši algoritam za konstrukciju u linearnom vremenu. Ovaj algoritam ima ugrađeno unapređenje za uštedu vremena s obzirom na Vajnerov algoritam (koji je prvo bio postignut u razvoju MekKrejtovog algoritma) i ima određene "on-line" osobine koje bi mogle da budu korisne u određenim situacijama. Ove "on-line" osobine će biti opisane u nastavku, ali treba imati na umu da je glavna vrlina Ukonenovog algoritma jednostavnost njegovog opisa, dokaza i vremena za analizu. Jednostavnost dolazi otud, što algoritam može da se razvije i kao jednostavan, ali neefikasan metod, praćen "zdravorazumskim" implementacijama i trikovima koji uspostavljaju bolje vreme izvršenja najgoreg slučaja. Ovo manje direktno izlaganje bi trebalo da je razumljivije pošto je lakše usvojiti svaki korak.

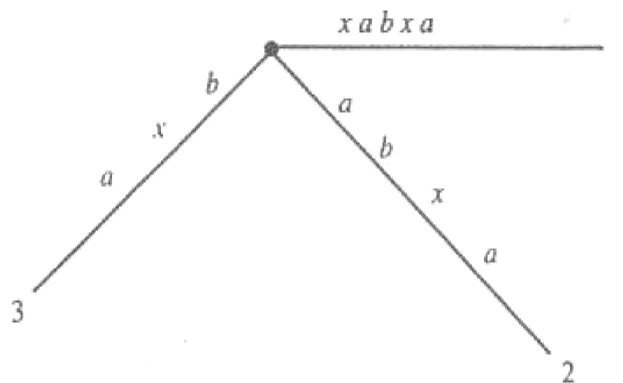
2.5.1 Implicitna sufiksna stabla

Ukonenov algoritam konstruiše niz implicitnih stabala, od kojih se poslednje konvertuje u pravo sufiksno stablo stringa S.

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.
Implicitno sufiksno stablo za prefiks $S[1..n]$ od S je slično definisano, uzimanjem sufiksnog stabla za $S[1..n]\$$ i brisanjem znaka $, grana i čvorova kao što je gore spomenuto.



Slika 2.3 ***Sufiksno stablo za string xabxa $***



Slika 2.4 *Implicitno sufiksno stablo za string xabxa*

Definicija: Označimo implicitno sufiksno stablo stringa $S[1..n]$ sa $\mathbf{I_i}$ gde i ide od 1 do m.

Implicitno sufiksno stablo će za svaki string S imati manje listova nego sufiksno stablo za string S$ ako i samo ako barem jedan od sufiksa za S, je prefiks nekog drugog sufiksa. Znak za kraj $ je bio dodat na kraj S upravo da bismo izbegli ovu situaciju. Međutim ako se S završava znakom koji se ne pojavljuje nigde drugde u S, tada će implicitno sufiksno stablo za S imati list za svaki sufiks i stoga će biti pravo sufiksno stablo.

Kao primer razmotrimo sufiksno stablo na slici 2.3. Sufiks xa je prefiks sufiksa $xabxa$ i slično sufiks a je prefiks $abxa$. Stoga u sufiksnom stablu za $xabxa$ ivice koje vode ka listovima 4 i 5 su označene samo sa $. Uklanjanjem ovih grana prave se dva čvora sa samo jednim sinom a ovi zatim takođe bivaju uklonjeni. Rezultujuće implicitno stablo za $xabxa$ je pokazano na slici 2.4. Kao naredni primer slika 2.1 pokazuje stablo izgrađeno za string $xabxac$. S obzirom na to da se znak c pojavljuje samo na kraju tog stringa, stablo na toj slici je uporedo i sufiksno i implicitno stablo za ovaj string.
Iako implicitno sufiksno stablo može da nema list za svaki sufiks, ono kodira sve sufikse od S, odnosno svaki sufiks je ispisan znacima na nekom putu od korena implicitnog sufiksnog stabla. Međutim ako se put ne završava u listu, tada neće biti markera da ukaže na kraj puta. Stoga su implicitna sufiksna stabla sama po sebi malo manje informativna nego prava sufiksna stabla. Upravo ta stabla će biti upotrebljena kao alat u Ukonenovom algoritmu da bi se najzad dobilo pravo sufiksno stablo za S.

2.5.2 Ukonenov algoritam – osnovna struktura

Ukonenov algoritam konstruiše implicitno sufiksno stablo $\mathbf{I_i}$ za svaki prefiks $S[1..n]$ za S, počevši od $\mathbf{I_1}$ uvećava i za jedan dok stablo $\mathbf{I_m}$ ne bude konstruisano.
Pravo sufiksno stablo za S je konstruisano iz $\mathbf{I_m}$ a vreme potrošeno za ceo algoritam je $O(m)$. Ukonenov algoritam će biti objašnjen najpre prezentovanjem metoda u vremenu $O(m^3)$ da bi se konstruisala sva stabla $\mathbf{I_i}$ a zatim da bi se optimizovanjem njegove implementacije dobio spomenuti vremenski nivo složenosti.

Opis Ukonenovog algoritma na višem nivou

Ukonenov algoritam je podeljen u m faza. U fazi $i+1$, stablo $\mathbf{I}_{i+1}$ je konstruisano iz $\mathbf{I}_i$. Svaka faza $i+1$ je dalje podeljena u $i+1$ produženja, po jedno za svaki od $i+1$ sufiksa za $S[1..i+1]$. U produženju j u fazi $i+1$, algoritam najpre nalazi kraj puta od korena označenog podstringom $S[j..i]$. Tada algoritam produžava podstring dodajući znak $S(i+1)$ na njegove krajeve, sve dotle dok se $S(i+1)$ već ne pojavi tamo. Tako da je u fazi $i+1$, string $S[1..i+1]$ prvi stavljen u stablo, praćen stringovima $S[2..i+1]$, $S[3..i+1]$, ...(u produženjima $1, 2, 3, \ldots$ respektivno).

Definicija: Sufiks S$[i..j]$ je prazan sufiks ako je $i > j$.

Produženja $i+1$ u fazi $i+1$ produžuju *prazan sufiks* za $S[1..i]$ odnosno postavljaju jedan znak stringa $S(i+1)$ u stablo (osim ako već nije tamo). Stablo $\mathbf{I}_1$ predstavlja samo jedna grana označena znakom $S(1)$. Proceduralno, algoritam se predstavlja na sledeći način:

Osnovna struktura Ukonenovog algoritma

Konstruiše stablo $\mathbf{T}_1$
for $i = 1$ to $m-1$
begin {faza $i+1$}
  for $j = 1$ to $i+1$
    begin {produženje j}
    Polazeći od korena nalazi se kraj puta označenog sa $S[j..i]$ u tekućem stablu.
    Ako je potrebno, taj put se produžava dodavanjem znaka $S(i+1)$,
    i time osigurava da je string $S[j..i+1]$ u stablu.
  end;
end;

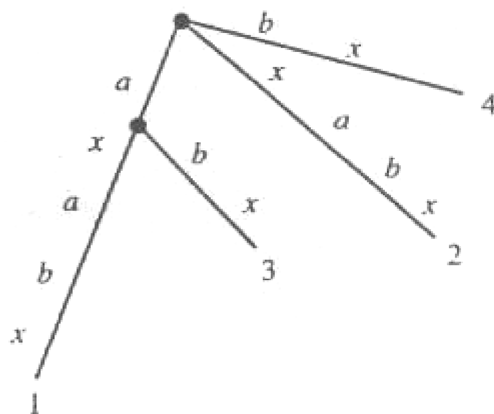
Pravila za produženje sufiksa

Da bi se transformisao ovaj opis višeg nivoa u algoritam, mora tačno da se odredi kako da se predstavi *sufiksno produženje*. Pustimo $S[j..i] = \beta$ da bude sufiks od $S[1..i]$. U produženju j, kada algoritam nađe kraj β u tekućem stablu, ono produžuje β da bi se osiguralo da je sufiks $\beta\, S(i+1)$ u stablu. Ono radi po narednom od sledeća tri pravila:

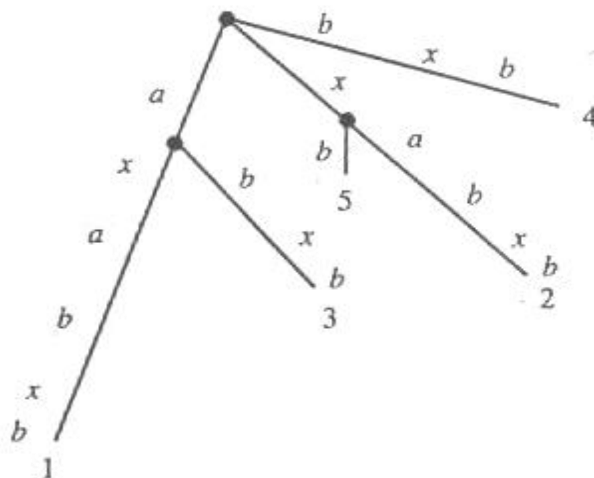
Pravilo 1 U tekućem stablu, put β se završava u listu**.** To znači da se put od korena označenog sa β, produžava do kraja neke grane sa listom. Da bismo unapredili ovo stablo, znak $S(i+1)$ se dodaje na kraj oznake na toj grani sa listom.

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

Pravilo 3 Neki put od kraja stringa β počinje znakom $S(i+1)$. U ovom slučaju string $\beta S(i+1)$ je već u tekućem stablu, tako da se dalje ne radi ništa (setimo se da u implicitnom sufiksnom stablu kraj sufiksa ne mora da bude eksplicitno označen).



Slika 2.5 Implicitno sufiksno stablo za string axabx pre nego što je šesti znak b, dodat.



Slika 2.6 Produženo implicitno sufiksno stablo nakon dodavanja znaka b.

Kao primer, razmotrimo implicitno sufiksno stablo za $S = axabx$ prikazano na slici 2.5. Prva četiri sufiksa se završavaju u listovima, ali jedan znak - sufiks x se završava unutar grane. Kada je šesti znak b dodat stringu, prva četiri sufiksa bivaju produžena primenom pravila 1, peti sufiks biva produžen pravilom 2 a šesti pravilom 3. Rezultat je prikazan na slici 2.6.

2.5.3 Implementacija i ubrzavanje

Koristeći pravila za produženje sufiksa, jednom kada je kraj sufiksa $\beta\, S[1..i]$ pronađen u tekućem stablu, potrebno je samo konstantno vreme da bi se izvršila pravila za produženje sufiksa (da bi se osiguralo da se sufiks $\beta\, S[1..i]$ nalazi u stablu). Ključna stavka u implementaciji Ukonenovog algoritma je zapravo kako pronaći krajeve svih $i+1$ sufiksa za $S[1..i]$.
Ideja je pronaći kraj svakog od $i+1$ sufiksa β u vremenu $O(|\beta|)$ idući od korena tekućeg stabla. Ovim pristupom, produženje j u fazi $i+1$ bi trajalo $O(i+1-j)$ vremena, $\mathbf{I_{i+1}}$ bi mogao da se napravi iz $\mathbf{I_i}$ u vremenu $O(i^2)$ dok bi $\mathbf{I_m}$ mogao da se napravi u vremenu $O(m^3)$. Ovaj algoritam izgleda malo budalasto s obzirom na to da je već poznat direktan algoritam za izgradnju sufiksnog stabla u vremenu $O(m^2)$, ali je lakše opisati Ukonenov $O(m)$ algoritam kao ubrzanje za gore spomenuti metod $O(m^3)$.

Gore spomenuti $O(m^3)$ nivo složenosti može da se redukuje do vremena $O(m)$ sa nekoliko zapažanja i trikova implementacije. Svaki trik po sebi izgleda kao senzibilni heuristički metod ubrzanja algoritma, ali delujući pojedinačno ovi trikovi ne moraju obavezno da redukuju nivo složenosti najgoreg slučaja. Međutim zajedno, oni dostižu nivo složenosti najgoreg slučaja. Najvažniji element ubrzanja je korištenje sufiksnih linkova.

Sufiksni linkovi: prva implementacija ubrzanja

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

Ponekad će se sufiksni link od v do $s(v)$ biti imenovan kao par $(v, s(v))$. Na primer na slici 2.1 neka je v čvor sa oznakom puta xa i neka je $s(v)$ čvor čija je oznaka puta pojedinačni znak a. Tada ovde postoji sufiksni link od čvora v do čvora $s(v)$. U ovom slučaju, α je dug svega jedan znak.
Kao specijalni slučaj, ako je α prazan, tada sufiksni link iz unutrašnjeg čvora sa oznakom puta $x\alpha$ ide do korenog čvora. Koreni čvor se sam po sebi ne smatra unutrašnjim i iz njega ne izlazi 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 stvari, biće ustanovljeno nešto daleko čvršće u narednim lemama i zaključcima.

Lema 2.1.1 Ako je dodat novi unutrašnji čvor v sa oznakom puta $x\alpha$ tekućem stablu u produženju j neke faze $i+1$, tada se put označen sa α završava već na unutrašnjem čvoru tekućeg stabla ili će unutrašnji čvor na kraju stringa da se napravi (pomoću pravila o produženju) u produženju $j+1$ u istoj toj fazi $i+1$.

Dokaz: Novi unutrašnji čvor v je napravljen u produženju j (u fazi $i+1$) samo kada se primeni drugo pravilo o produženju. To znači da u produženju j, put označen sa $x\alpha$, je nastavljen nekim znakom različitim od $S(i+1)$, recimo c. Stoga u produženju $j+1$, postoji put označen sa α u

stablu i sigurno ima nastavak sa znakom c (moguće i sa drugim znacima). Tada postoje dva slučaja za razmatranje: ili put označen sa α nastavlja samo sa znakom c, ili nastavlja sa nekim drugim znakom. Kada je α nastavljeno samo sa c, drugo pravilo o produženju će napraviti čvor $s(v)$ na kraju puta α.

Kada se α nastavi sa dva različita znaka, na kraju puta α već mora da postoji čvor $s(v)$. Time je lema dokazana u oba slučaja. □

Posledica 2.1.1 U Ukonenovom algoritmu, svaki novonapravljeni unutrašnji čvor će imati sufiksni link od sebe do kraja narednog produženja.

Dokaz: Dokaz je induktivne prirode i istinit je za stablo $\mathbf{I_1}$, s obzirom na to da $\mathbf{I_1}$ ne sadrži unutrašnje čvorove. Pretpostavimo da je tvrđenje istinito do kraja faze i, i uzmimo u obzir pojedinačnu fazu $i+1$. Prema lemi 2.1.1, kada je novi čvor napravljen u produženju j, ispravan čvor $s(v)$ koji završava sufikni link iz v biće pronađen ili napravljen u produženju $j+1$. Ni jedan novi unutrašnji čvor neće biti napravljen u poslednjem produženju faze (produženje koje se bavi pojedinačnim znakom sufiksa $S(i+1)$, tako da svi sufiksni linkovi iz unutrašnjih čvorova napravljenih u fazi $i+1$ su poznati do kraja faze i stablo $\mathbf{I_{i+1}}$ ima sve njegove sufiksne linkove. □

Posledica 2.1.2 U svakom implicitnom sufiksnom stablu $\mathbf{I_i}$, ako unutrašnji čvor v ima oznaku puta $x\alpha$ tada se tamo nalazi čvor $s(v)$ iz $\mathbf{I_i}$ sa oznakom puta α.

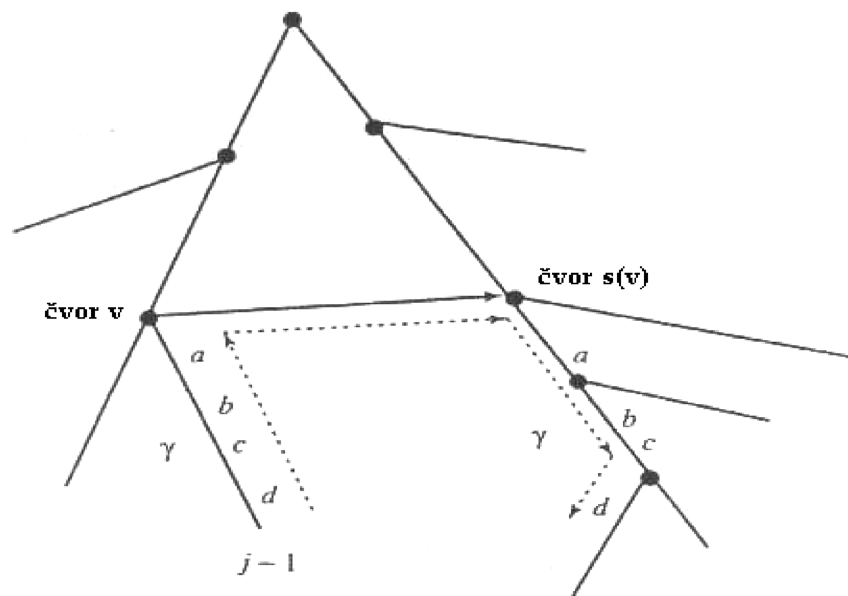
Prema posledici 2.1.1 svi unutrašnji čvorovi u rastućem stablu će imati sufiksne linkove koji izlaze iz njih, osim unutrašnjeg čvora koji je poslednji dodat, koji će da dobije svoj sufiksni link do kraja narednog produženja. U tekstu koji sledi biće pokazano kako se sufiksni linkovi koriste za ubrzanje implementacije.

Praćenje traga sufiksnih linkova da bi se izgradilo stablo $\mathbf{I_{i+1}}$

Prisetimo se da u fazi $i+1$ algoritam nalazi sufikse $S[j..i]$ za $S[1..i]$ u produženju j, za j rastuće od 1 do $i+1$. U principu ovo je postignuto upoređivanjem stringa $S[j..i]$ duž puta od korena u tekućem stablu. Sufiksni linkovi mogu da skrate ovaj put i svako produženje. Prva dva produženja (za $j=1$ i $j=2$) u svakoj fazi $i+1$ su najlakši za opisivanje.
Kraj punog stringa $S[1..i\,]$ mora da se završi u listu $\mathbf{I_i}$ s obzirom na to da je $S[1..i\,]$ najduži string predstavljen u stablu. To olakšava nalaženje kraja tog sufiksa (pošto su stabla konstruisana, može da se zadrži pokazivač na čvor koji odgovara tekućem punom stringu $S[1..i\,]$), a njegovo sufiksno produženje biva obrađeno prvim pravilom za produženje sufiksa. Tako da je prvo produženje bilo koje faze jedinstveno i traje konstantno vreme s obzirom na to da algoritam ima pokazivač na kraj tekućeg punog stringa.
Neka string $S[1..i]$ bude $x\alpha$, gde je x pojedinačni znak a α je (moguće prazan) podstring, i neka $(v,1)$ bude grana stabla koja ulazi u list 1. Sledeći algoritam mora da pronađe kraj stringa $S[2..i]=\alpha$ u tekućem stablu izvedenom iz $\mathbf{I_i}$. Ključ je u tome da se čvor v nalazi ili u korenu ili je unutrašnji čvor za $\mathbf{I_i}$. Ako je koren, tada da bi pronašao kraj za α, algoritam treba samo da siđe niz stablo prateći put označen sa α, kao u jednostavnom algoritmu. Međutim, ako je v unutrašnji

čvor za $\mathbf{I_i}$, onda po lemi 2.1.2 (s obzirom na to da je v bio u $\mathbf{I_i}$), v odatle ima sufiksni link do čvora $s(v)$. Ovo ima za posledicu, da prilikom traženja kraja za α u tekućem stablu, algoritam ne mora da krene od korena, već samo od čvora $s(v)$. To je zapravo glavni smisao uključivanja sufiksnih linkova u algoritam.

Da bi ovo drugo produženje bilo detaljnije opisano, označimo sa γ granu $(v, 1)$. Da bi se pronašao kraj za α, krene se od lista 1 do čvora v; prati se sufiksni link od v do $s(v)$; i ide se od $s(v)$ niz put (koji može da bude više od jedne grane) označen sa γ. Kraj puta je kraj α (videti sliku 2.7). Na kraju puta za α, stablo je unapređeno po pravilima za produženje sufiksnih linkova. Ovo u potpunosti opisuje prva dva produženja faze $i + 1$.
Da bi se produžio svaki string $S[j..i]$ do $S[1..i+1]$ za $j > 2$, ponovi se ista opšta ideja: počevši od kraja stringa $S[j-1..i]$ u tekućem stablu, ide se najviše jedan čvor ili do korena ili do čvora v koji ima sufiksni link iz njega; neka je γ oznaka te grane; pretpostavlja se da v nije koren, obilazi se sufiksni link od v do $s(v)$; tada se siđe dole niz stablo od $s(v)$, prateći put označen sa γ do kraja $S[j..i]$; na kraju se produži sufiks do $S[j..i+1]$ prema pravilima o produženju.
Postoji jedna mala razlika između produženja za $j > 2$ i prva dva produženja. Uopšteno, kraj $S[j-1..i]$ može da bude u čvoru u kojem već postoji sufiksni link, i u tom slučaju algoritam obilazi taj sufiksni link. Primetimo da čak i kada se primeni drugo pravilo za produženje u produženju $j-1$ (tako da je kraj za $S[j-1..i]$ na novonapravljenom unutrašnjem čvoru w) ako roditelj za w nije koren, tada roditelj za w već ima sufiksni link, kako je garantovano lemom 2.1.1. Stoga u produženju j, algoritam se nikada ne penje više od jedne grane.



Slika 2.7 Produženje j > 1 u fazi i+1. Penje se za jednu granu (označenu sa γ) od kraja puta označenog sa S[j-1..i] do čvora v; zatim se prati sufiksni link do s(v); zatim se silazi niz put određujući podstring γ; zatim se primenjuje odgovarajuće pravilo za produženje za umetanje sufiksa S[j..i+1].

Algoritam za pojedinačno produženje: APP

Sastavljajući ove delove zajedno, kada su implementirani korišćenjem sufiksnih linkova, produženje $j \geq 2$ u fazi $i+1$ je:

Algoritam za pojedinačno produženje:

Početak

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 (kao u jednostavnom algoritmu).
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), tada po lemi 2.1.1. string α mora da se završi u čvoru $s(w)$, završnom čvoru za sufiksni link iz w. Sada se napravi sufiksni link $(w, s(w))$ od w do $s(w)$.

Kraj.

Pretpostavlja se da algoritam čuva pokazivač na tekući pun string $S[\,j..i\,]$, tako da prvo produženje u fazi $i+1$ nema potrebe za hodom na gore ili na dole. Dalje, za prvo produženje u fazi $i+1$ se uvek primenjuje prvo pravilo za produženje sufiksa.

Šta je do sada postignuto?

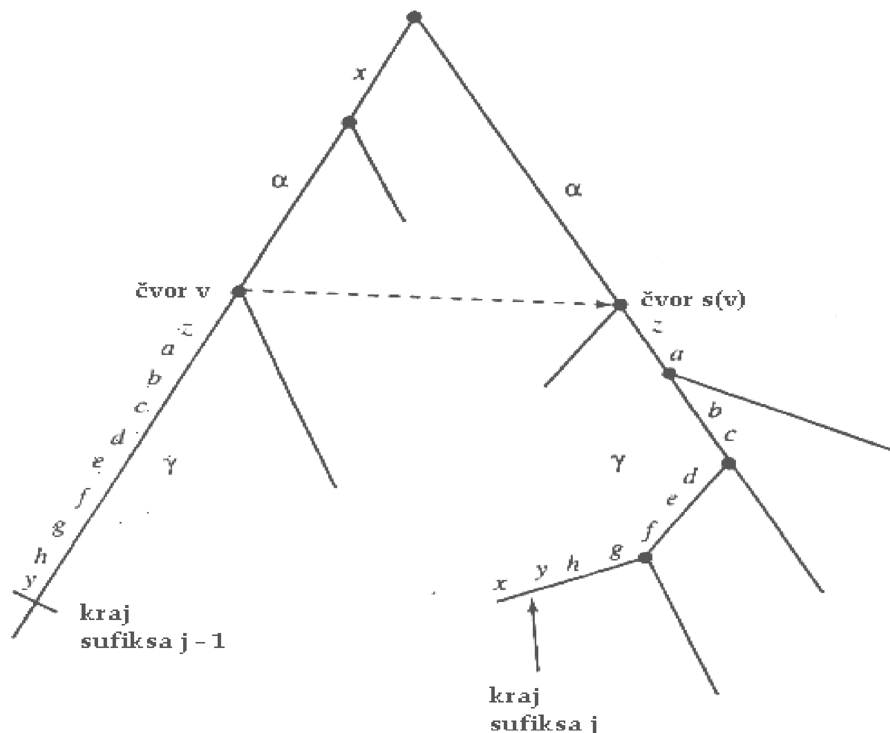
Upotreba sufiksnih linkova je jasno i praktično unapređenje u odnosu na hod od korena u svakom produženju, kao što je urađeno u jednostavnom algoritmu. Ali, da li njihova upotreba unapređuje izvršenje u vremenu najgoreg slučaja?
Odgovor je, kao što je već opisano, da upotreba sufiksnih linkova ne unapređuje vremenski nivo složenosti. Međutim, ovde može da se predstavi trik koji će smanjiti vreme najgoreg slučaja za algoritam $\mathrm{O}(m^2)$. Ovaj trik će takođe biti centralni trik i u drugim algoritmima za izgradnju i upotrebu sufiksnih stabala.

Trik broj 1: preskoči / prebroj trik

U koraku 2 za produženje $j+1$ algoritam silazi od čvora $s(v)$ duž puta označenog sa γ. Prisetimo se da tamo zasigurno mora da postoji takav put γ od $s(v)$. Direktno implementirano, ovaj hod duž γ uzima vreme proporcionalno $|\gamma|$, broju znakova na tom putu. Ali, jednostavan trik nazvan preskoči / prebroj trik, će redukovati vreme obilaska do neke veličine proporcionalne broju čvorova na tom putu. Tada sledi da je vreme za sva silaženja u jednoj fazi najviše $\mathrm{O}(m)$.

Trik1: Označimo sa g dužinu γ i prisetimo se da nikoje dve oznake grana iz $s(v)$ ne mogu da počnu istim znakom, tako da prvi znak γ mora da se pojavi kao prvi znak na tačno jednoj grani;

Neka g' predstavlja broj znakova na toj grani. Ako je g' manje od g tada algoritam više ne mora dalje da razmatra znake. Jednostavno preskače do čvora na kraju grane. Vrednost g zamenjuje se sa $g - g'$, promenljivoj h dodeljuje se vrednost $g' + 1$ i pretražuje izlazeće grane tražeći ispravnu narednu granu na putu (čiji se prvi znak poklapa sa h iz γ). Uopšteno, kada algoritam pronađe narednu granu na putu, on upoređuje trenutnu vrednost za g sa brojem znakova g' na toj grani. Kada je $g > g'$, algoritam skače na čvor na kraju grane, zamenjuje g sa $g - g'$, a h sa $h + g'$ i nalazi granu čiji prvi znak je znak h iz γ i onda ponavlja sve to. Kada je dostignuta grana gde je $g \leq g'$, tada algoritam skače do znaka g na grani i staje, siguran da se put γ od $s(v)$ završava u grani tačno g znakova niz granu. (videti sliku 2.8).



Slika 2.8 Preskoči / prebroj trik. U fazi i+1, podstring γ ima dužinu deset. Ovde imamo kopiju podstringa γ iz čvora s(v); Nađena su tri znaka niz poslednju granu, nakon što su izvršena četiri preskakanja čvorova.

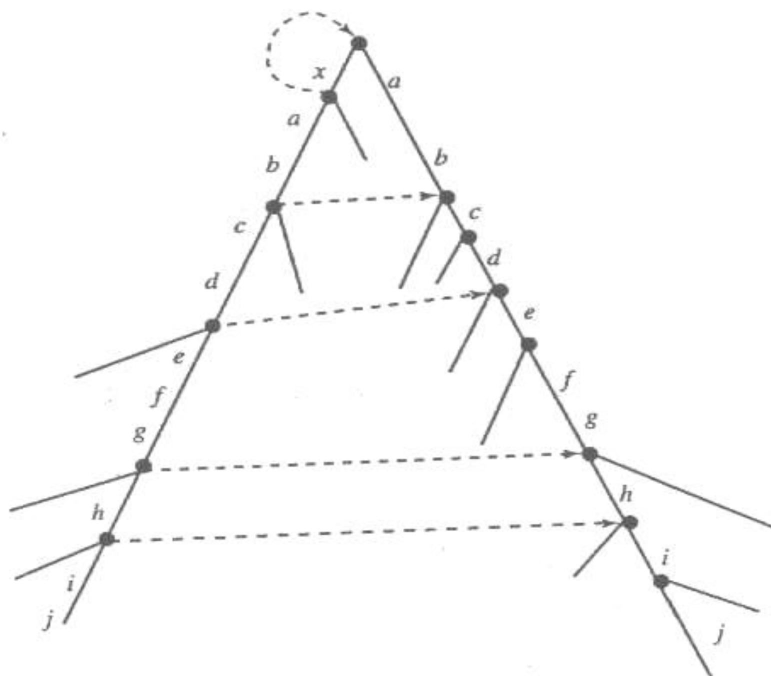
Pretpostavljajući jednostavne i očigledne detalje implementacije (kao što je poznavanje broja znakova na svakoj grani i sposobnost da se u konstantnom vremenu odstrani znak iz S na bilo kojoj poziciji) efekat korišćenja preskoči / prebroj trika je sposobnost kretanja po putu γ od jednog do drugog čvora u konstantnom vremenu[1]. Celokupno vreme za obilazak puta je onda proporcionalno broju čvorova na njemu, radije nego broju znakova na njemu. Ovo je korisna heuristika, ali šta se time dobija u domenu nivoa složenosti najgoreg slučaja? Naredna lema 2.1.2 vodi direktno ka odgovoru na to pitanje.

Definicija: Dubina čvora (čvora u), predstavlja broj čvorova na putu od korena do čvora u.

[1] Opet pretpostavljamo da imamo alfabet fiksirane veličine.

Lema 2.1.2 Neka je $(v, s(v))$ bilo koji sufiksni link postavljen za vreme Ukonenovog algoritma. U tom momentu, dubina čvora za v je za najviše jedan veća od dubine čvora za $s(v)$.

Dokaz: Kada je grana $(v, s(v))$ obiđena, svaki unutrašnji naslednik za v, koji ima oznaku puta recimo $x\beta$, ima sufiksni link do čvora sa oznakom puta β. Ali $x\beta$ je prefiks puta do v, tako da je β prefiks puta do $s(v)$ i sledi da sufiksni link od svakog unutrašnjeg naslednika od v ide do naslednika za $s(v)$. Još više ako je β neprazan, tada je čvor označen sa β unutrašnji čvor. Sada, iz razloga što dubine čvorova svaka dva naslednika za v moraju da se razlikuju, svaki naslednik za v ima sufiksni link do različitog naslednika od $s(v)$. Sledi da je dubina čvora od $s(v)$ najmanje jedan (za koren), plus broj unutrašnjih naslednika za v koji imaju oznake puta dužine više od jednog znaka. Jedini poseban naslednik kojeg v ima (bez odgovarajućeg naslednika od $s(v)$) je unutrašnji naslednik čija oznaka puta ima dužinu jedan (ima oznaku x). Stoga, v može da ima dubinu čvora najviše za jedan više nego $s(v)$. (videti sliku 2.9) □



Slika 2.9 Za svaki čvor v na putu xα odgovarajući čvor s(v) je na putu α . Međutim dubina čvora za s(v) može da bude za jedan manja nego dubina čvora za v, može da bude jednaka ili može da bude veća. Na primer, čvor označen sa xab ima dubinu čvora dva, dok je dubina čvora za ab jedan. Dubina čvora označenog sa xabcdefg je četiri, dok dubina čvora za abcdefg je pet.

Definicija: Prilikom rada algoritma, *tekuća dubina čvora* algoritma je dubina čvora kojeg je algoritam poslednjeg posetio.

Teorema 2.1.1: Koristeći preskoči / prebroj trik, bilo koja faza Ukonenovog algoritma uzima $O(m)$ vremena.

Dokaz: U fazi i postoji $i + 1$ produženja. U pojedinačnom produženju algoritam obilazi najviše jednu ivicu da bi našao čvor sa sufiksnim linkom, obiđe jedan sufiksni link, siđe dole za neki broj čvorova, primeni pravila o produženju sufiksa i možda doda sufiksni link. Već je uspostavljeno da sve operacije osim silaska traju konstanto vreme po produženju, tako da sve što je porebno da se uradi, jeste da se analizira vreme potrebno za silazak. Ovo se radi proučavanjem kako se menja tekuća dubina čvora kroz fazu.
Penjanje u svakom produženju smanjuje tekuću dubinu čvora najviše za jedan (s obzirom na to da se kreće na gore najviše za jedan čvor), svaki obilazak sufiksnog linka smanji dubinu čvora za najviše još jedan (po lemi 2.1.2) i svaka grana koja je obiđena prilikom silaska ide do čvora veće dubine. Stoga kroz čitavu fazu tekuća dubina čvora je smanjena najviše $2m$ puta i s obzirom da ni jedan čvor ne može da ima dubinu veću od m, celokupno moguće uvećanje tekuće dubine čvora ima nivo složenosti $3m$ kroz čitavu fazu. Sledi da kroz čitavu fazu, celokupan broj prelazaka grana tokom silaska ima nivo složenosti 3m. Korišćenjem preskoči / prebroj trika vreme po obilasku na dole je konstantno, tako da je celokupno vreme za sve silaske u fazi $O(m)$ i time je teorema dokazana. □

Pošto imamo m faza, neizbežna je naredna posledica:

Posledica 2.1.3 Ukonenov algoritam može da se implementira pomoću sufiksnih linkova da bi radio u vremenu $O(m^2)$.

Primetimo da je nivo složenosti $O(m^2)$ za algoritam dobijen množenjem nivoa složenosti $O(m)$ u jednoj fazi sa m (s obzirom na to imamo m faza). Ovo grubo množenje je bilo neophodno jer je vreme za analizu usmereno samo na jednu fazu. Ono što je potrebno jesu neke promene u implementaciji koje će da dozvole da vreme potrebno za analizu pređe nivo složenosti faze. Ovo će biti urađeno uskoro.

U ovom trenutku ovo može da izgleda pomalo varljivo jer izgleda kao da nije napravljen nikakav napredak, s obzirom na to smo krenuli sa jednostavnim $O(m^2)$ metodom. Čemu sav taj rad samo da bi smo se vratili na isti nivo složenosti? Odgovor je, da iako nije napravljen veliki napredak na vremenskom nivou složenosti, napravljen je veliki konceptualni napredak tako da će sa primenom samo nekoliko lakih detalja vreme spasti na $O(m)$. Zapravo, biće potrebno jedno pojednostavljenje implementacije i još dva mala trika.

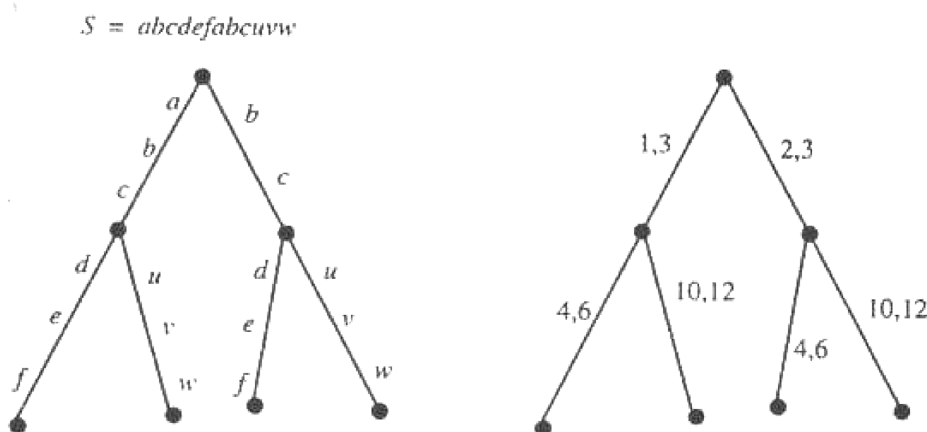
2.5.4 Važno pojednostavljenje implementacije

Sledeće što se uspostavlja je nivo složenosti veličine $O(m)$ za izgradnju sufiksnog stabla. Međutim na putu do tog cilja postoji jedna prepreka - sufiksno stablo može da zahteva $\Theta(m^2)$ prostora. Kao što je do sada opisano, oznake grana sufiksnog stabla mogu sveukupno da sadrže više od $\Theta(m)$ znakova. S obzirom na to je vreme za algoritam barem isto toliko veliko koliko i njegov izlaz, tako mnogo znakova čini da nivo složenosti $O(m)$ postaje nemoguć. Posmatrajmo string $S = abcdefghijklmnopqrstuvwxyz$. Svaki sufiks počinje različitim znakom; Stoga tu

imamo 26 grana iz korena i svaki je označen kompletnim sufiksom, zahtevajući sveukupno po 26x27/2 znakova. Za stringove duže od veličine alfabeta, neki znaci će se ponavljati, ali jedan može da napravi stringove značajne veličine m tako da rezultujuće oznake grana imaju sveukupno više od $\Theta(m)$ znakova. Stoga vreme $O(m)$ algoritma za izgradnju sufiksnih stabala stabala potražuje neku alternativnu šemu koja će da predstavi oznake grana.

Kompresija oznake grane

Postoji jednostavna alternativna šema za označavanje. Umesto eksplicitnog pisanja podstringa na grani stabla, samo ispišemo par indicija na grani, određujući početak i kraj pozicije tog podstinga u S (videti sliku 2.10). S obzirom na to da algoritam ima kopiju stringa S, može da nađe svaki određeni znak u stringu S na osnovu date pozicije u stringu u konstantnom vremenu. Zato možemo da opišemo svaki određeni algoritam za sufiksno stablo kao da su oznake grana eksplicitno date a opet algoritam se implementira samo sa konstantnim brojem simbola ispisanim na bilo kojoj grani (par indeksa koji označava početak i kraj pozicija podstringa).
Na primer u Ukonenovom algoritmu kada se traži duž grane, algoritam koristi par indeksa ispisanih na grani da bi se vratili potrebni znaci iz S i onda da urade poređenje na tim znacima. Pravila za produženje se takođe implementiraju s lakoćom pomoću šeme označavanja. Kada se drugo pravilo za produženje primeni u fazi $i+1$, označimo novonapravljene grane pomoću para indeksa $(i+1, i+1)$ a kada se primeni prvo pravilo za produženje (na grani listova), promeni se par indeksa na toj grani listova sa (p, q) na $(p, q+1)$. Lako je induktivno videti da je q trebalo da bude i, te stoga nova oznaka $(p, i+1)$ predstavlja tačan novi podstring za tu granu listova.
Koristeći par indeksa da odredimo oznaku grane, samo dva broja su napisana na svakoj grani i s obzirom na to da je broj grana najviše $2m-1$, sufiksno stablo koristi samo $O(m)$ znakova i zahteva samo $O(m)$ prostora. Ovo čini još verovatnijim da stablo zapravo može da se konstruiše u vremenu $O(m)$. Iako puna implementacija algoritma neće eksplicitno da ispiše podstring na grani, ipak potrebno je spomenuti da se govori „o podstringu ili oznaci na grani puta" kao da je tamo bio napisan eksplicitni podstring.



Slika 2.10 Levo stablo je fragment sufiksnog stabla za string S = abcdefabcuvw, sa oznakama grana pisanih eksplicitno. Desno stablo pokazuje kompresovane oznake grana. Primetimo da je grana sa oznakama 2, 3 takođe mogla da se označi i sa 8, 9.

2.5.5 Još dva pojednostavljenja

Predstavićemo još dva trika za implementaciju koja dolaze iz dva zapažanja o načinu na koji pravila za produženje uzajamno deluju u uzastopnim produženjima i fazama. Ovi trikovi će, zajedno sa lemom 2.1.2 na kraju dovesti do željenog linearnog nivoa složenosti.

Zapažanje1: Pravilo 3 je zaustavljač procesa U svakoj fazi, ako treće pravilo o produženju sufiksa primenimo u produženju j, ono će se primeniti takođe i u svim naredim produženjima (od $j+1$ do $i+1$) sve do kraja faze. Razlog tome je taj što kada se primeni pravilo o produženju sufiksa, put označen sa $S[i..j]$ u tekućem stablu mora da se nastavi znakom $S(i+1)$ a isto to radi i put sa $S[j+1..i]$ a treće pravilo o produženju sufiksa se opet primenjuje u produženjima $j+1, j+2, ..., i+1$.
Kada se primeni treće pravilo o produženju sufiksa, ništa više ne preostaje da se uradi s obzirom na to da je sufiks koji nas zanima već u stablu. Šta više, u stablo je nakon produženja potrebno dodati novi sufiksni link u kojem je primenjeno drugo pravilo o produženju sufiksa. Ove činjenice i zapažanje 1 vode ka narednom triku za implemenataciju.

Trik2: Zaustavlja svaku fazu $i+1$ prvi put kada je primenjeno treće pravilo o produženju. Ako se ovo dogodi u produženju j, tada više nema potrebe da se eksplicitno nalazi kraj svakog stringa $S[k..i]$ za $k > j$. Za produženja u fazi $i+1$ koja su "završena" nakon prvog izvršenja trećeg pravila o produženju se kaže da su urađena *implicitno*. Ovo je suprotno u odnosu na svako produženje j gde je kraj za $S[j..i]$ nađen eksplicitno. Produženje te vrste se naziva *eksplicitno produženje*.
Jasno je da je trik 2 dobra heuristika, koja redukuje količinu rada, ali nije jasno da li vodi do boljeg vremenskog nivoa složenosti najgoreg slučaja. Za to nam je potrebno još jedno zapažanje i jedan trik.

Zapažanje 2: Jednom list, zauvek list Odnosno, ako je u nekoj tački u Ukonenovom algoritmu napravljen list koji je označen sa j, (za sufiks koji počinje na poziciji j u S) tada će list ostati u svim stablima naslednicima napravljenim tokom algoritma. Ovo je istina, jer algoritam nema mehanizme za produženje grane listova dalje od njegovog trenutnog lista. Detaljnije, kada je jednom list označen sa j, prvo pravilo za produženje će uvek da se primeni do produženja j u svakoj narednoj fazi. Tako da, jednom list - zauvek list.
Sada je list 1 napravljen u fazi 1, tako da u svakoj fazi i postoji inicijalni niz za naredna produženja (počevši od produženja 1) gde se primenjuju prvo i drugo pravilo za produženje. Neka j_i označava poslednje produženje u ovoj sekvenci. S obzirom da svaka primena drugog pravila za produženje pravi novi list, iz zapažanja 2 sledi da $j_i \leq j_{i+1}$. Odnosno inicijalni niz produženja gde se primenjuju prvo i drugo pravilo o produženju ne može da se smanji u narednoj fazi. Ovo sugeriše trik za implementaciju koji u fazi $i+1$ izbegava sve eksplicitne nastavke od 1 do j_i. Umesto toga će za ova produženja implicitno da se traži samo konstantno vreme.
Zarad opisa trika prisetimo se da oznaka za svaku granu u implicitnom sufiksnom stablu (ili sufiksnom stablu) može da se predstavi sa dva indikatora p, q navođenjem podstringa $S[p..q]$. Prisetimo se takođe da za svaku granu lista za $\mathbf{I_i}$, indeks q je jednak i a u fazi $i+1$ indeks q dobija uvećanje do $i+1$, koji reflektuje dodavanje znakova $S(i+1)$ na kraj svakog sufiksa.

Trik 3: U fazi $i+1$**,** kada je napravljena prva grana sa listom, normalno bi bila označena sa podstringom $S[p..i+1]$, umesto ispisivanja indikatora $(p, i+1)$ po grani, pišemo (p, e), gde je e znak koji označava trenutni kraj. Znak e je *globalni indeks* koji je postavljen na $i+1$ jednom u svakoj fazi. U fazi $i+1$, s obzirom da algoritam zna da će prvo pravilo da se primeni u produženjima od 1 do barem j_i, nije potreban dalji dodatni eksplicitni rad da bi se implementirala ta j_i produženja. Umesto toga, potreban je samo konstantni rad da bi se uvećala promenljiva e, i tada se primenjuje eksplicitan rad za (neka) produženja počevši od produženja j_i+1.

Smisao

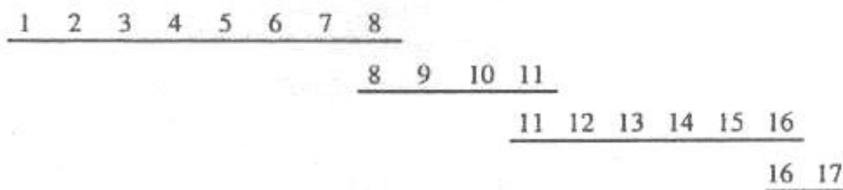
Pomoću trikova 2 i 3, eksplicitna produženja u fazi (koristeći APP) su jedina koja se zahtevaju od produženja j_i+1 do prvog produženja gde se primenjuje treće pravilo o produženju (ili dok produženje $i+1$ nije završeno). Sva druga produženja (pre ili posle tih eksplicitnih produženja) su urađena implicitno. Sumirajući ovo, faza $i+1$ je implementirana na sledeći način:

Algoritam za pojedinačno produženje: APP

Početak
1. Uvećaj indeks e do $i+1$. (Trikom 3 korektno se implementiraju sva implicitna produženja od 1 do j_i)
2. Eksplicitno se izvrše naredna produženja (koristeći algoritam za pojedinačno produženje) počevši od j_i+1 dok se ne dosegne prvo produženje j^* gde se primenjuje treće pravilo o produženju ili dok sva produženja u ovoj fazi nisu završena. (Trikom 2 ovo korektno implementira sva dodatna implicitna produženja od j^*+1 do $i+1$.
3. Postavi se j_{i+1} u j^*-1, da bi bilo spremno za narednu fazu.

Kraj.

Korak 3 pravilno postavlja j_{i+1}, jer inicijalni niz produženja je mesto gde se primenjuju prvo i drugo pravilo o produženju koja moraju da se završe u tačci gde se primenjuje treće pravilo o produženju.
Ključna stavka APP algoritma je da će faza $i+2$ početi sa izvršavanjem eksplicitnih produženja sa produženjem j^*, gde je j^* bilo poslednje eksplicitno produženje izvršeno u fazi $i+1$. Stoga dve naredne faze dele barem jedan indeks (j^*) gde je izvršeno eksplicitno produženje (videti sliku 2.11). Šta više, faza $i+1$ se završava znajući gde se završava string $S[j^*..i+1]$, tako da za ponovljeno produženje j^* u fazi $i+2$ može da se primeni pravilo za produženje na j^* bez bilo kakvih penjanja, obilazaka sufiksnih linkova ili preskakanja čvorova. To znači da prvo eksplicitno produženje u bilo kojoj fazi troši samo konstantno vreme. Sada je lako potvrditi krajnji rezultat.



Slika 2.11 Crtež mogućih izvršenja Ukonenovog algoritma. Svaka linija predstavlja fazu algoritma a svaki broj predstavlja eksplicitno produženje izvršeno algoritmom. Na ovom crtežu su predstavljene četiri faze i sedamnaest eksplicitnih produženja. U bilo kojoj od ove dve uzastopne faze, postoji najviše jedan indeks u kojem je jedno isto eksplicitno produženje izvšeno u obe faze.

Teorema 2.1.2: Koristeći sufiksne linkove i implementacione trikove 1, 2 i 3, Ukonenov algoritam gradi implicitna sufiksna stabla u celokupnom vremenu $O(m)$.

Dokaz: Vreme koje je potrebno za sva implicitna sufiksna produženja u svakoj fazi je konstantno i iznosi $O(m)$ kroz čitav algoritam. Pošto algoritam izvršava eksplicitna produženja, razmatra se indeks $\bar{j}$ koji odgovara eksplicitnom produženju koji algoritam trenutno izvršava. Kroz čitavo izvršavanje algoritma, $\bar{j}$ nikad ne opada, ali ostaje isti između dve uzastopne faze. S obzirom na to da imamo samo m faza i na to da $\bar{j}$ ima nivo složenosti $O(m)$, algoritam stoga izvršava samo $2m$ eksplicitnih produženja. Kako je ustanovljeno ranije, vreme za eksplicitno produženje je konstantno plus neko vreme proporcionalno broju preskočenih čvorova tokom silaska u tom produženju.
Da bi smo ograničili celokupan broj preskočenih čvorova tokom svih silazaka, razmatramo (slično dokazu teoreme 2.1.1) kako se tekuća dubina čvora menja tokom uzastopnih produženja, pa čak i produženja tokom različtih faza. Ključ je u tome da se prvo eksplicitno produženje u svakoj fazi (nakon faze 1) počinje produženjem j^*, koje je bilo poslednje eksplicitno produženje u prethodnoj fazi. Stoga, tekuća dubina čvora se ne menja između kraja jednog produženja i početka drugog. Ali, (kao što je navedeno u dokazu teoreme 2.1.1), u svakom eksplicitnom produženju tekuće dubine čvora je prvo smanjen za najviše dva (penjanjem za jednu ivicu i obilaskom jednog sufiksnog linka) i stoga silazak u tom produženju uvećava tekuću dubinu čvora za jedan pri svakom preskakanju čvora. S obzirom na to da je maksimum dubine čvora m, biće samo $2m$ eksplicitnih produženja. Sledi (kao u dokazu teoreme 2.1.1) da maksimalni broj preskočenih čvorova napravljen tokom silaska (i to ne samo tokom jedne faze) ima nivo složenosti $O(m)$. Time je teorema dokazana i sav rad je bio vredan toga. □

2.5.6 Konverzija implicitnog u sufiksno stablo

Konačno implicitno sufiksno stablo $\mathbf{I}_m$ može da se trasformiše u pravo sufiksno stablo u vremenu $O(m)$. Prvo se doda znak za kraj stringa $ na kraj S i pusti se da Ukonenov algoritam nastavi sa ovim znakom. Efekat je taj, da ni jedan sufiks sada nije prefiks bilo kog drugog sufiksa, tako da se izvršenje Ukonenovog algoritma završava implicitnim sufiksnim stablom u kome se svaki sufiks završava u listu i time je eksplicitno predstavljen. Jedina druga promena koja je potrebna, je zameniti svaki indeks e u svakoj grani listova brojem m. Ovo je postignuto obilaženjem stabla

u vremenu O(m), posećujući svaku granu listova. Kada se urade ove promene, dobijeno stablo je pravo sufiksno stablo. Sve ovo možemo da zaokružimo narednom teoremom:

Teorema 2.1.3: Ukonenov algoritam gradi pravo sufiksno stablo za S, sa svim njegovim sufiksnim linkovima u vremenu O(m).

2.5.7 Implementacija Ukonenovog algoritma

U dodatku rada nalazi se program SufiksnoStablo koji implementira rad Ukonenovog algoritma za konstrukciju sufiksnog stabla. Programski kod je prevashodno zasnovan na kodu programa SuffixTree čiji su autori Dotan Tsadok i Shlomo Yona preuzetog sa web sajta http://mila.cs.technioin.ac.il/~yona/suffix_tree/ odeljenja za informatiku Univerziteta u Haifi.

Primarni zadatak programa je nalaženje sufiksnog stabla na osnovu zadatog stringa i po potrebi ispisivanja istog na ekran a sekundarni zadatak je nalaženje pozicija zadatog podstringa u okviru istog tog stringa. Njegova primena se najbolje ogleda u pretraživanju DNK živih organizama.

String se zadaje neposredno ili putem pripremljenog fajla dok se podstring zadaje samo neposredno. U narednom segmentu sledi uputstvo za pravilno pokretanje i korišćenje programa.

Uputstvo za pokretanje i korišćenje programa:

UPOTREBA: SufiksnoStablo <cmd> <string | ime fajla> <podstring> [p]

Primeri pravilnog unosa parametara:

Neposredni string: SufiksniNiz s mojstring podstring [p]
String iz fajla: SufiksniNiz f putanja_do_fajla podstring [p]

<cmd> - Naredba. Naredba je prvi argument u komandnoj liniji.
Moguće naredbe su:

s - ako je naredba s, string koji se obrađuje je drugi argument u komandnoj liniji.

f - ako je naredba f, string koji se obrađuje se nalazi u fajlu čija je putanja drugi argument u komandnoj liniji.

ts - isto kao s naredba ali uključuje takodje i samotestiranje

tf - isto kao f naredba, ali uključuje takodje i samotestiranje

<string | ime_fajla> - string koji se obrađuje ili putanja do fajla koji ga sadrži u zavisnosti od prvog argumenta u komandnoj liniji (vidi <cmd>).

<podstring> - podstring je treći argument u komandnoj liniji koji se traži nakon što je konstrukcija sufiksnog stabla završena.

[p] - opciono - ispisuje stablo. Ispisivanje je korisno kada imamo posla sa malim stablima, dok ispisivanje velikog stabla može da potraje dosta vremena.

Funkcije koje se koriste u programu:

Zbog prevelikog broja funkcija (32) opisi funkcija se ovde ne navode nego su opisani u okviru samog koda.

Ulaz i izlaz programa:

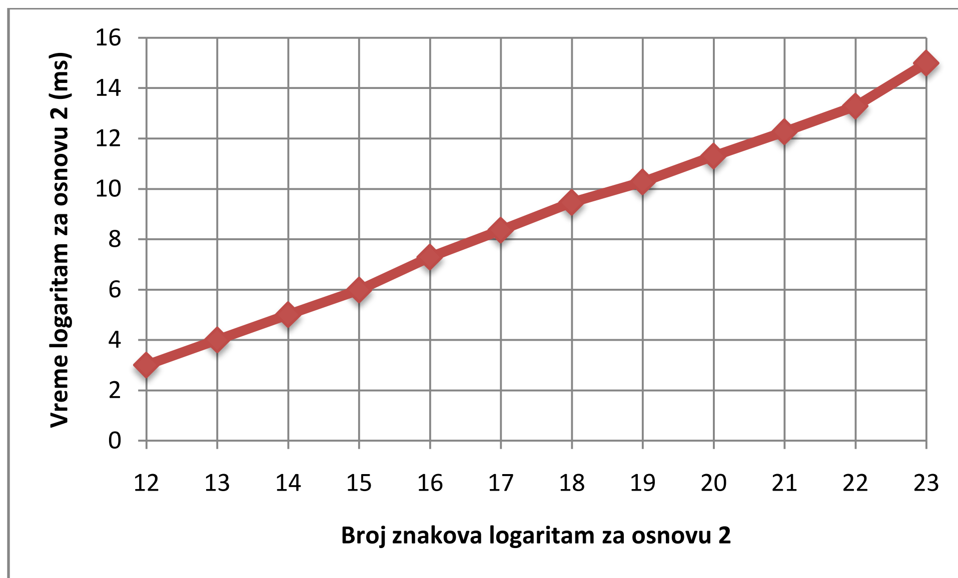
Ulazni parametri programa: izvorni string i podstring kome se traže pozicije pojavljivanja u stringu.
Izlazni parametri programa: sufiksno stablo i pozicije na kojima se nalazi podstring.

Karakteristike računara na kojem je program testiran:

Procesor: 1.61 GHz AMD Sempron(tm) 3000+
RAM memorija: 1.5 GB

Dijagram merenja rezultata:

Neka je x veličina ulaznog niza, a y vreme kostrukcije sufiksnog stabla u milisekundama. Na sledećem dijagramu prikazana je zavisnost $\log_2 x$ (veličine u rasponu od 12 do 23) od $\log_2 y$.



Slika 2.11 Prikaz rezultata merenja potrebnog za konstrukciju sufiksnog stabla u zavisnosti od broja znakova

Ocena koeficjenta pravca prave k:

Sa dijagrama se može oceniti koeficjent pravca prave k kroz prvu i pretposlednju tačku:

$$\log y = \log y_0 + k (\log x - \log x_0)$$
$$\log_2 y = \log_2 y_0 + k (\log_2 x - \log_2 x_0)$$
$$\log_2 y - \log_2 y_0 = k (\log_2 x - \log_2 x_0)$$
$$k = \frac{\log_2 y - \log_2 y_0}{\log_2 x - \log_2 x_0} = \frac{13.28 - 3}{22 - 12} = \frac{10.28}{10} \approx 1.03$$

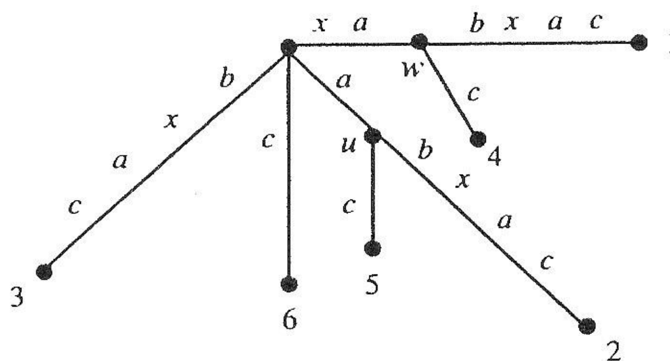
Iz toga sledi procena $y = \frac{y_0}{x_0^{1.03}} x^{1.03}$, što se može smatrati potvrdom da je složenost algoritma linearna.

Sledi primer kojim se testira rad Ukonenovog algoritma za konstrukciju sufiksnog stabla:

Test primer:

Ulaz: String $S = xabxac$
Izlaz: Sufiksno stablo u narednom zapisu: + predstavlja koreni čvor
¦ + predstavlja čvor na postojećoj grani

Slika sufiksnog stabla za dati string:



Prikaz sufiksnog stabla na ekranu za dati string:

```
root
+ xa
¦ + bxac$
¦ + c$
+ a
¦ + bxac$
¦ + c$
+ bxac$
+ c$
+ $
```

2.6 Vajnerov linearni algoritam

Za razliku od Ukonenovog algoritma, Vajnerov algoritam počinje celim stringom S. Međutim, kao i Ukonenov algoritam, unosi se po jedan sufiks u rastuće stablo, iako u bitno različitom rasporedu. Zapravo, najpre se unese string $S(m)\$$ u stablo, zatim string $S[m-1..m]\$,\ldots,$ i na kraju se u stablo unese ceo string $S\$$.

Definicija: $Suff_i$ označava sufiks $S[i..m]$ za S počevši od pozicije i.

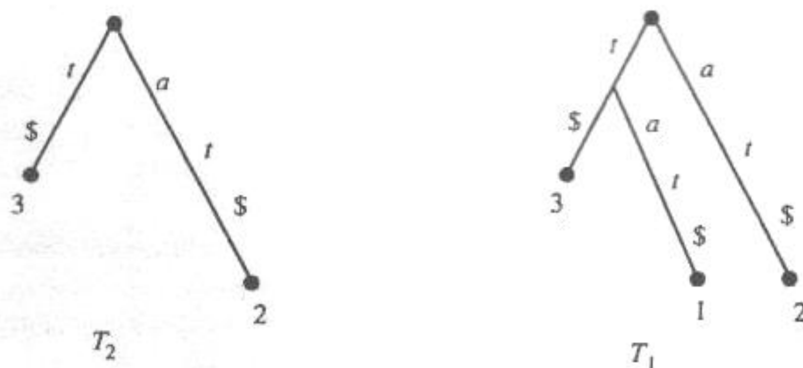
Na primer, $Suff_1$ je čitav string S i $Suff_m$ je pojedinačni znak $S(m)$.

Definicija: Definišimo $\mathbf{T_i}$ tako da bude stablo koje ima $m-i+2$ listova numerisanih brojevima od i do $m+1$, takvih da put od korena do svakog lista j $(i \leq j \leq m+1)$ ima oznaku $Suff_j\$$. Stoga, ovo je sufiksno stablo za string $S[1..m]\$$.

Vajnerov algoritam konstruiše stabla od $\mathbf{T_{m+1}}$ na dole do $\mathbf{T_i}$ (na primer u opadajućem redosledu za i). Prvo će biti implementiran metod na direktan neefikasan način. Ovo će dalje poslužiti kao pomoćno sredstvo, da bi se lakše predstavile i ilustrovale važne definicije i činjenice. Tada će da se ubrza direktna konstrukcija da bi se dobio Vajnerov linearno-vremenski algoritam.

2.6.1 Direktna konstrukcija

Prvo stablo $\mathbf{T_{m+1}}$ se sastoji prosto od jedne ivice iz korena označenog znakom za kraj rada \$. Stoga sledi, da za svaki i od m na dole do 1, algoritam konstruiše svako stablo $\mathbf{T_i}$ iz stabla $\mathbf{T_{i+1}}$ i znak $S(i)$. Ideja samog metoda je u osnovi ista kao ideja za konstrukciju stabala sa ključnim rečima, ali za različit skup stringova i bez postavljanja pokazivača unazad. Kako algoritam nastavlja sa radom, svako stablo $\mathbf{T_i}$ će imati svojstvo da za svaki čvor v u $\mathbf{T_i}$, ni jedne dve grane koje izlaze iz v nemaju oznake grana koje počinju istim znakom. S obzirom na to da $\mathbf{T_{m+1}}$ ima samo jednu granu koja izlazi iz korena, ovo je trivijano tačno za $\mathbf{T_{m+1}}$. Induktivno pretpostavlja se da je ovo svojstvo tačno za stablo $\mathbf{T_{i+1}}$ i potvrdiće se da to važi i za stablo $\mathbf{T_i}$.
U opštem slučaju, da bi se kreiralo $\mathbf{T_i}$ iz $\mathbf{T_{i+1}}$ počne se od korena $\mathbf{T_{i+1}}$ i silazi se onoliko dugo koliko je to moguće putem čija se oznaka poklapa sa prefiksom od $Suff_i\$$.. Označimo taj put sa R. Detaljnije, put R je nađen počevši od korena i eksplicitno upoređujući uzastopne znake iz $Suff_i\$$ sa uzastopnim znacima duž jedinstvenog puta u $\mathbf{T_{i+1}}$. Put traženja je jedinstven, s obzirom na to da za svaki čvor iz $\mathbf{T_{i+1}}$ ni jedne dve grane koje izlaze iz čvora v nemaju oznake grana koje počinju istim znakom. Stoga se upoređivanje nastavlja barem na jednoj grani koja izlazi iz v. Ultimativno, s obzirom na to da ni jedan sufiks nije prefiks nekog drugog, dalja upoređivanja nisu moguća. Ako u toj tačci ne postoji ni jedan čvor, tada tu napravimo novi čvor. U svakom slučaju, označimo taj čvor (stari ili novi) sa w. Na kraju, dodamo granu iz w novom listu označenog sa i, i označimo novu granu (w,i) sa preostalim (nepretraženim) delom $Suff_i\$$.
S obzirom na to da dalje traženje nije bilo moguće, prvi znak na oznaci grane (w,i) se ne pojavljuje kao prvi znak na bilo kojoj grani koja izlazi iz w. Stoga je tvrđenje o induktivnom svojstvu održano. Jasno je da put od korena do lista ima oznaku $Suff_i\$$. Odnosno, put tačno ispisuje string $Suff_i\$$, tako da je time stablo $\mathbf{T_i}$ konstruisano.
Na primer, slika 2.12 pokazuje transformaciju iz $\mathbf{T_2}$ u $\mathbf{T_1}$ za string tat.



Slika 2.12 Jedan korak u Vajnerovom algoritmu. Puni string tat je dodat sufiksnom stablu za at. Grana označena pojedinačnim znakom $ je izostavljena, s obzirom na to da je takva grana deo svakog sufiksnog stabla.

Definicija: Za svaku poziciju i, $Glava(i)$ označava najduži prefiks za $S[i..m]$ koji se poklapa sa podstringom za $S[i+1..m]\$$.
Primetimo da $Glava(i)$ može da bude i prazan string. Zapravo, $Glava(i)$ je uvek prazan string jer $S[i+1..m]$ je prazan string kada je $i+1$ veći od m, a znak $S(m) \neq \$$.

Definicija: String $S[i..j]$ je prazan string ako je $i > j$.

S obzirom na to da kopija stringa $Glava(i)$ počinje na nekoj poziciji između $i+1$ i m, $Glava(i)$ je takođe i prefiks za $Suff_k$ za neko $k > i$. Sledi da je $Glava(i)$ najduži prefiks (moguće prazan) koji je zapravo oznaka nekog puta od korena u stablu $\mathbf{T_{i+1}}$. Gore spomenuti direktan algoritam za izgradnju $\mathbf{T_i}$ iz $\mathbf{T_{i+1}}$ može da se opiše na sledeći način:

Uprošćena varijanta Vajnerovog algoritma

Pronađi kraj puta označen sa $Glava(i)$ u stablu $\mathbf{T_{i+1}}$.
Ako nema čvora na kraju $Glava(i)$, onda se napravi jedan, i sa w se označi čvor (napravljen ili ne) na kraju $Glava(i)$. Ako je w napravljen u ovoj tačci, razdvajajući postojeću granu, tada on razdvaja njegovu postojeću oznaku grane $Glava(i)$. Onda napravi novi list numerisan brojem i, i novu granu (w, i) označenu preostalim znacima iz $Suff_i\$$. Odnosno nova oznaka grane bi trebalo da bude poslednji $m - i + 1 - |Glava(i)|$ znak iz $Suff_i\$$, praćen znakom za kraj $.

2.6.2 Popravke efiksanosti

Trebalo bi da bude jasno da je ovim direktnim metodom sufiksno stablo $\boldsymbol{T} = \boldsymbol{T_1}$ konačno konstruisano u vremenu O(m^2). Jasno je takođe da je to teži deo algoritma za nalaženje $Glava(i)$, s obzirom da korak 2 zahteva samo konstantno vreme za svako i. Tako da bi se ubrzao algoritam, biće potreban efikasniji način da se pronađe $Glava(i)$. Ali, u diskusiji za Ukonenov algoritam stoji da nivo složenosti lineranog vremena nije moguć ako su oznake grana eksplicitno ispisane na stablu. Umesto toga, svaka oznaka grane je predstavljena sa dva indikatora koji

pokazuju početak i kraj pozicije označavajućeg podstringa. Radi podsećanja dobro bi bilo ponovo pogledati odeljak 2.5.4 ″Jednostavan detalj za implementaciju″.
Lako je implementirati Vajnerov algoritam koristeći par indeksa da bi se označila grana. Kada se umeće $Suff_i\$$, pretpostavlja se da je algoritam napravio poređenja do k-tog znaka na grani (u,z) označenog intervalom $[s,t]$, ali naredni znak se neće poklapati. Novi čvor w je napravljen deljenjem (u,z) na dve grane (u,w) i (w,z) a nova grana je takođe napravljena od w do lista i. Grana (u,w) biva označena sa $[s,s+k-1]$, grana (w,z) biva označena sa $[s+k\ ,t]$, a grana (w,i) biva označena sa $[i+d(w),m]\$$, gde je $d(w)$ težinska dubina stringa (broj znakova) puta od korena na dole do čvora w. Ove dubine stringova lako mogu da se naprave i koriguju u skladu sa izgradnjom stabla, s obzirom da $d(w)=d(u)+k$. Težinska dubina stringa za list i je $m-i+1$.

Efikasno nalaženje $Glava(i)$

Vratimo se sada centralnoj temi oko toga kako efikasno pronaći $Glava(i)$. Ključ za Vajnerov algoritam su dva vektora koja se čuvaju na svakom čvoru koji nije list (uključujući koren). Prvi vektor se naziva vektor indikator I a drugi se naziva link vektor L. Svaki vektor ima dužinu jednaku veličini alfabeta i svaki je indeksiran znacima iz tog alfabeta. Na primer za engleski alfabet proširen sa $, svaki link i vektor indikator će imati dužinu 27.
Link vektor je u osnovi obrat sufiksnog linka u Ukonenovom algoritmu i dva linka su korištena na sličan način da bi ubrzala obilaske unutar stabla.
Vektor indikator je bit vektor tako da su njegovi ulazi samo 0 ili 1, dok je svaki ulaz u link indikator ili nula vektor ili je pokazivač na stablo čvorova. Neka $I_v(x)$ odredi ulaz vektora indikatora na čvoru v indeksiranog znakom x. Slično neka $L_v(x)$ odredi ulaz link vektora na čvoru v indeksiranog znakom x.
Vektori I i L imaju dva krucijalna svojstva koja će induktivno da se održavaju kroz algoritam:

- Za svaki (pojedinačni) znak x i svaki čvor u, $I_u(x)=1$ u $\mathbf{T_{i+1}}$ ako i samo ako je *put* od korena do $\mathbf{T_{i+1}}$ označen sa $x\alpha$, gde je α oznaka puta čvora u. Put označen sa $x\alpha$ nema potrebe za krajem u čvoru.
- Za svaki znak x, $L_u(x)$ u $\mathbf{T_{i+1}}$ pokazuje na unutrašnji čvor $\bar{u}$ u $\mathbf{T_{i+1}}$, ako i samo ako je oznaka puta tog čvora $x\alpha$, gde $\bar{u}$ ima oznaku puta α. Inače $L_u(x)$ je nula vektor.

Na primer, na slici 2.1 razmatramo dva unutrašnja čvora u i w sa oznakama puteva a i xa respektivno. Tada $I_u(x)=1$ za specifični znak x i $L_u(x)=w$, Takođe, $I_w(b)=1$, ali $L_w(b)$ je nula vektor.
Jasno je da za svaki čvor u i svaki znak x, $L_u(x)$ je različit od nule samo ako je $I_v(x)=1$, ali obrat tog iskaza nije tačan. Takođe je jasno da ako je $I_u(x)=1$, tada je $I_v(x)=1$ za svakog naslednika čvora v za u.
Stablo $\mathbf{T_m}$ ima samo jedan čvor koji nije list odnosno, to je koren r. U ovom stablu $I_r(S(m))$ se postavi na jedan, $I_r(x)$ se postavi na nulu za svaki drugi znak x i sve ulaze za link na koren se postave na nula vektor. Stoga, gornja svojstva važe za $\mathbf{T_m}$. Algoritam će menjati vektore u skladu sa promenama u stablu i induktivno će biti dokazano da gornja svojstva važe za svako stablo.

2.6.3 Osnovna ideja Vajnerovog algoritma

Vajnerov algoritam koristi indikator i link vektore da nađe $Glava(i)$ i da konstruiše $\mathbf{T_i}$ efiksanije. Algoritam mora da vodi računa o dva degenerativna slučaja, ali oni se ne razlikuju mnogo od generalnog "dobrog" slučaja gde se degeneracije ne pojavljuju. Najpre diskutujemo kako da se konstruiše $\mathbf{T_i}$ iz $\mathbf{T_{i+1}}$ u dobrom slučaju a onda možemo da se postaramo za degenerativne slučajeve.

Algoritam u dobrom slučaju

Pretpostavimo da je stablo $\mathbf{T_{i+1}}$ upravo konstruisano i sada sledi izgradnja stabla $\mathbf{T_i}$. Algoritam počinje u listu $i + 1$ za $\mathbf{T_{i+1}}$ (list za $Suff_{i+1}$), i penje se do korena tražeći prvi čvor v ako postoji, takav da $I_v(S(i)) = 1$. Ako je pronađen tada nastavljamo od v penjanjem prema korenu tražeći prvi čvor v' koji sretne (moguće *v)* kada $L_{v'}(S(i))$ nije nula.
Po definiciji, $L_{v'}(S(i))$ nije nula samo ako $I_v(S(i)) = 1$, tako da ako je pronađen v', on će takođe biti prvi čvor na koji se naiđe prilikom hoda od lista $i + 1$ kada je $L_{v'}(S(i))$ različit od nule. U opštem slučaju može da se desi da ni v ni v' ne postoje ili da v postoji a v' ne postoji. Primetimo međutim, da v ili v' može da bude koren. Dobar slučaj je kad i v i v' postoje.
Neka l_i bude broj znakova na putu između v' i v i ako $l_i > 0$, tada označimo sa c prvi od ovih l_i znakova.
Pretpostavljajući da imamo "dobar slučaj" t.j. da oba v i v' postoje, u narednom tekstu biće dokazano da ako čvor v ima oznaku puta α tada $Glava(i)$ je tačno string $S(i)\alpha$. Dalje će biti dokazano da kada $L_{v'}(S(i))$ pokazuje na čvor v'' u $\mathbf{T_{i+1}}$, $Glava(i)$ se završava ili u v'' ako je $l_i = 0$, ili se završava *tačno* l_i znakova ispod v'' na grani koja izlazi iz v''. Tako da u svakom slučaju, $Glava(i)$ može da se nađe u konstatnom vremenu nakon što je v' pronađeno.

Teorema 2.2.1: Pretpostavimo da je čvor v upravo nađen algoritmom i da ima oznaku puta α. Tada je string $Glava(i)$ upravo $S(i)\alpha$.

Dokaz: $Glava(i)$ je najduži prefiks za $Suff_i$, koji je takođe i prefiks za $Suff_k$, za neko $k > i$.
S obzirom da je v pronađeno sa $I_v(S(i)) = 1$ ovde postoji put u $\mathbf{T_{i+1}}$ koji počinje sa $S(i)$, tako da je *Glava(i)* duga barem jedan znak. Stoga, možemo da izrazimo $Glava(i)$ kao $S(i)\beta$, za neki (moguće prazan) string β.
$Suff_i$ i $Suff_k$ oba počinju sa stringom $Glava(i) = S(i)\beta$ a razlikuju se nakon toga. Radi tačnosti, recimo da $Suff_i$ počinje sa $S(i)\beta a$ a $Suff_k$ počinje sa $S(i)\beta b$. Ali tada $Suff_{i+1}$ počinje sa βa a $Suff_{k+1}$ počinje sa βb**.** Obe veličine $i + 1$ i $k + 1$ su veće ili jednake od $i + 1$ i manje ili jednake od m**,** tako da su oba sufiksa predstavljena u stablu $\mathbf{T_{i+1}}$. Stoga u stablu $\mathbf{T_{i+1}}$ mora da postoji put od korena označenog sa β (moguće prazan string) koji se račva na dve grane, jedna nastavlja znakom a, dok druga nastavlja znakom b. Stoga imamo čvor u, u $\mathbf{T_{i+1}}$ sa oznakom puta β, i $I_u(S(i)) = 1$ s obzirom na to da postoji put (naime inicijalni deo puta do lista k) označen sa $S(i)\beta$ *u* $\boldsymbol{T_{i+1}}$. Dalje, čvor u mora da bude na putu do lista $i + 1$ s obzirom na to da je β prefiks od $Suff_{i+1}$.
Sada $I_v(S(i)) = 1$ i v imaju oznaku puta α, tako da $Glava(i)$ mora da počne sa $S(i)\alpha$. To znači da je α prefiks za β, tako da čvor u sa oznakom puta β, mora da bude ili v ili ispod v na putu do lista $i + 1$. Međutim, ako je $u \neq v$ tada bi u bio za čvor ispod v *na putu do čvora* $i + 1$ i

$I_u(S(i)) = 1$. Ovo protivreči izboru čvora v, tako da $v = u$, $\alpha = \beta$, i teorema je dokazana. Odnosno, $Glava(i)$ je upravo string $S(i)\alpha$. □

Primetimo da se u teoremi 2.2.1 i njenom dokazu samo pretpostavlja da čvor v postoji. Nikakva pretpostavka o v' nije data. Ovo će biti korisno u jednom od degenerativnih slučajeva koji će biti proučeni kasnije.

Teorema 2.2.2: Pretpostavimo da su v i v' pronađeni i $L_{v'}(S(i))$ pokazuje na čvor v''. Ako je $l_i = 0$ tada se $Glava(i)$ završava u v''; inače se završava nakon tačno l_i znakova na jednoj grani koja izlazi iz v''.

Dokaz: S obzirom na to da na putu do lista $i + 1$ i $L_{v'}(S(i))$ pokazuje na čvor v'', put od korena označenog sa $Glava(i)$ mora da uključi v''. Po teoremi 2.2.1 $Glava(i) = S(i)\alpha$ tako da $Glava(i)$ mora da se završi tačno l_i znakova ispod v''. Stoga, kada $l_i = 0$, $Glava(i)$ se završava u v'' . Ali kada je $l_i > 0$, mora da postoji grana $e = (v'', z)$ koja izlazi iz v'' čija oznaka počinje sa znakom c (prvi od l_i znakova na putu od v' do v) u $\mathbf{T_{i+1}}$.
Može li $Glava(i)$ da se produži na dole do čvora z (na primer do čvora ispod v'')? Čvor z mora da bude čvor račvanja. Jer ako bi taj čvor bio list, tada bi neki sufiks $Suff_k$, za $k > i$, bio prefiks za $Suff_i$, što nije moguće. Neka z ima oznaku puta $S(i)\gamma$. Ako se $Glava(i)$ produži na dole do čvora račvanja, tada ćemo tamo imati dva podstringa koji počinju na toj ili narednoj $i + 1$ poziciji u S koja oba počinju stringom γ. Stoga, tamo će da bude čvor z' sa oznakom puta γ u $\mathbf{T_{i+1}}$. Čvor z' će onda da bude ispod v' na putu do lista $i + 1$, što protivreči izboru v'. Tako da $Glava(i)$ ne mora da dosegne z i mora da se završi u unutrašnjosti grane e. Zapravo, završava se tačno l_i znakova od v'' na grani e. □

Stoga kada je $l_i = 0$, znamo da se $Glava(i)$ završava u v'' a kada $l_i > 0$, nalazimo $Glava(i)$ iz v'' ispitivanjem grana koje izlaze iz v'' da bi se identifikovala ta jedinstvena grana e čiji je prvi znak c. Tada se $Glava(i)$ završava upravo na l_i znakova niz e od v''. Stablo $\mathbf{T_i}$ je onda konstruisano podpodelom grane e, praveći čvor w na tom trenutku, i dodajući novu ivicu od w do lista i označenog podsetnikom na $Suff_i$. Traženje prave grane koja izlazi iz v'' troši samo konstantno vreme s obzirom na to da je alfabet fiksiran.
Sve u svemu, kada v i v'' postoje, gornji metod korektno konstruiše $\mathbf{T_i}$ iz $\mathbf{T_{i+1}}$ iako i dalje moramo da se diskutuje kako unaprediti vektore. Takođe, u ovom trenutku još ne mora da bude jasno zašto je ovaj metod za nalaženje $Glava(i)$ efikasniji nego uprošćeni algoritam. To će doći kasnije.
Proučimo sada kako algoritam obrađuje degenerativne slučajeve kada ne postoje ni jedan od v i v''. Dva degenerativna slučaja su ta da čvor v (a stoga i čvor v') ne postoje ili da v postoji a v'' ne postoji. Sada će se u dva slučaja videti kako efikasno naći $Glava(i)$. Prisetimo se da se koreni čvor označava sa r.

Slučaj1: $I_r(S(i)) = 0$.
U ovom slučaju hod se završava u korenu a čvor v nije pronađen. Sledi da se znak $S(i)$ ne pojavljuje ni na jednoj poziciji većoj od i, jer ako bi se pojavio, tada bi neki sufiks u tom opsegu počinjao sa $S(i)$, neki put od korena bi počinjao sa $S(i)$ a $I_r(S(i))$ bi bio 1. Tako da kada je $I_r(S(i)) = 0$, $Glava(i)$ je prazan string i završava se u korenu.

Slučaj2: $I_v(S(i)) = 1$ za neki v (moguće koren), ali v' ne postoji.
U ovom slučaju hod se završava tačno u korenu sa $L_r(S(i))$ kao nula vektorom. Neka t_i bude broj znakova od korena do v. Iz teoreme 2.2.1 $Glava(i)$ se završava t_{i+1} znakova od korena. S obzirom da v postoji, imamo neku granu $e = (r, z)$ čija oznaka grane počinje znakom $S(i)$. Ovo je tačno bez obzira na to da li je $t_i = 0$ ili $t_i > 0$.
Ako je $t_i = 0$, tada se $Glava(i)$ završava nakon ovog prvog znaka $S(i)$ na grani e.
Slično, ako $t_i > 0$., tada se $Glava(i)$ završava t_{i+1} znakova od korena na grani e. Pretpostavimo da se $Glava(i)$ prostire skroz do nekog deteta z (i preko toga). Tada tačno kao u dokazu teoreme 2.2.2, z mora da bude račvajući čvor i tu mora da bude čvor z' ispod korena na putu do lista $i + 1$ kao što je $L_{z'}(S(i))$ vektor različit od nule, što bi bila kontradikcija. Tako da kada $t_i > 0$, $Glava(i)$ se završava tačno t_{i+1} znakova od korena grani e koja izlazi iz korena. Ova grana može da se nađe iz korena u konstantnom vremenu s obzirom da je njegov prvi znak $S(i)$.

U svakom od ovih degenerativnih slučajeva (kao u dobrom slučaju), $Glava(i)$ je nađena u konstatnom vremenu nakon što hod dostigne koren. Nakon što je kraj $Glava(i)$ nađen i w je napravljen ili nađen, algoritam nastavlja isto onako kao i u dobrom slučaju.
Primetimo da je degenerativni slučaj 2 vrlo sličan ″dobrom″ slučaju kada su oba v i v' nađeni ali, se razlikuju u jednom malom detalju jer je $Glava(i)$ nađena t_{i+1} znakova na dole niz e, radije nego t_i znakova na dole (prirodna analogija dobrog slučaja).

2.6.4 Kompletan algoritam za konstrukciju T_i od T_{i+1}

Sastavljanje svih ovih slučajeva zajedno daje naredni algoritam:

Vajnerovo produženje stabla

1) Počevši od lista $i + 1$ za $\mathbf{T_{i+1}}$ (list za $Suff_{i+1}$) i penjući se prema korenu traga se za prvim čvorom v na tom hodu kao što je $I_v(S(i)) = 1$.
2) Ako je koren dostignut i $I_r(S(i)) = 0$, tada se $Glava(i)$ završava u korenu. Idi na korak 4.
3) Neka v bude pronađen čvor *(moguće sam koren)* kao što je $I_r(S(i)) = 1$.
Tada se nastavlja sa penjanjem tražeći prvi čvor v'*(moguće sam v)* takav da $L_{v'}(S(i))$ nije nula.
3a) Ako je koren dostignut i $L_r(S(i))$ je nula, neka t_i bude broj znakova na putu između korena i v. Traži se grana e koja izlazi iz korena koja izlazi iz korena čija oznaka grane počinje znakom $S(i)$. $Glava(i)$ se završava tačno t_{i+1} znakova od korena na grani e.
Inače (kada uslov pod 3a nije održiv).
3b) Ako je v' pronađen tako da $L_{v'}(S(i))$ nije nula, recimo v'', tada se sledi link (za $S(i)$) do v''. Neka je l_i broj znakova na putu od v' do v i neka c bude prvi znak na ovom putu. Ako je $l_i = 0$ tada se $Glava(i)$ završava u v''. Inače, traženje grane e koje izlazi iz v'' čiji prvi znak je c. $Glava(i)$ se završava tačno l_i znakova ispod v'' na grani e.
4) Ako čvor već postoji na kraju $Glava(i)$, tada neka w označava taj čvor; inače, napravimo čvor w na kraju $Glava(i)$. Napravi se novi list numerisan sa i; napravimo novu granu (w, i) označenu preostalim podstringovima za $Suff_i$ (na primer, poslednji od $m - i + 1 - |Glava(i)|$ znakova za $Suff_i$), praćeni znakom za kraj. Time je stablo $\mathbf{T_i}$ konstruisano.

Korektnost

Trebalo bi da bude jasno iz dokaza teoreme 2.2.1 i 2.2.2 i diskusije o degenerativnim slučajevima da kada algoritam korektno konstruiše stablo $\mathbf{T_i}$ iz $\mathbf{T_{i+1}}$, da bi bio u stanju da konstruiše $\mathbf{T_{i-1}}$ on mora prvo da koriguje vektore I i L.

Kako unaprediti vektore

Nakon nalaženja (ili pravljenja) čvora w, moramo da korigujemo vektore I i L, tako da budu korektni za stablo $\mathbf{T_i}$. Ako algoritam pronađe čvor v kao što je $I_v(S(i)) = 1$, tada po teoremi 2.2.1 čvor w ima oznaku puta $S(i)\alpha$ u $\mathbf{T_i}$, gde čvor v ima oznaku puta α. U ovom slučaju, $L_v(S(i))$ bi trebalo da se postavi da pokazuje na čvor w u $\mathbf{T_i}$. Ovo je jedino potrebno unapređenje za link vektore s obzirom da samo jedan čvor može da pokazuje putem link vektora na neki drugi čvor a napravljen je samo jedan novi čvor. Dalje, ako je čvor w novonapravljeni čvor, svi njegovi ulazni linkovi za $\mathbf{T_i}$ bi trebalo da budu nule. Da bi videli ovo, pretpostavi se suprotno - da tamo postoji čvor u u $\mathbf{T_i}$ sa oznakom puta $xGlava(i)$, gde čvor w ima oznaku puta $Glava(i)$. Čvor u ne može da bude list jer $Glava(i)$ ne sadrži znak \$. Ali tada u $\mathbf{T_{i+1}}$ mora da postoji čvor sa oznakom puta $Glava(i)$, što protivreči činjenici da je čvor w umetnut u $\mathbf{T_{i+1}}$ da bi se kreirao $\mathbf{T_i}$. Kao posledica toga proizilazi, da ne postoji čvor u $\mathbf{T_i}$ sa oznakom puta $xGlava(i)$ za svaki znak x i sve vrednosti vektora L za w treba da su nula.
Razmotrimo sada potrebne korekcije za vektore indikatore za stablo $\mathbf{T_i}$. Za svaki čvor u na putu od korena do čvora $i+1$, $I_u(S(i))$ mora da bude postavljen na 1 u $\mathbf{T_i}$ s obzirom na to, da je tamo sada put za string $Suff_i$ u $\mathbf{T_i}$. Lako je induktivno ustanoviti da ako je čvor v sa $I_u(S(i)) = 1$ pronađen za vreme hoda od lista $i+1$, tada svaki čvor u iznad v na putu do korena već ima $I_u(S(i)) = 1$. Stoga, potrebno je postaviti samo vektor indikatore za čvorove ispod v na putu do lista $i+1$. Ako nikakav čvor v nije pronađen, to znači da su obiđeni svi čvorovi na putu od lista $i+1$ do korena i da svi ovi čvorovi moraju da imaju unapređene njihove vektor indikatore. Potrebna unapređenja za čvorove ispod v mogu da se naprave tokom traženja v. Za vreme hoda od lista $i+1$, $I_u(S(i))$ je postavljen na 1 za svaki čvor u koji se sretne pri hodu. Vreme koje je potrebno da bi se postavili ovi vektori indikatori je proporcionalno vremenu hoda.
Jedino što je preostalo da se unapredi je da se postavi I vektor za novonapravljeni čvor w, napravljen u unutrašnjosti grane $e = (v'', z)$.

Teorema 2.2.3: Kada je napravljen novi čvor w u unutrašnjosti ivice (v'', z) vektor indikator za w bi trebalo da se kopira iz vektora indikatora za z.

Dokaz: Neposredno sledi da ako $I_v(x) = 1$ tada $I_w(x)$ takođe mora da bude 1 u $\mathbf{T_i}$. Ali, da li može da se desi da $I_w(x)$ treba da bude 1 a opet $I_z(x)$ je postavljen na 0 u momentu kada je w napravljen? Videćemo da ne može.
Neka čvor z ima oznaku puta γ, i naravno čvor w ima oznaku puta $Glava(i)$ prefiksa za γ. Činjenica da u $\mathbf{T_i}$ nema čvorova između u i z, znači da svaki sufiks iz $Suff_m$ na dole do $Suff_{i+1}$ koji počinje stringom $Glava(i)$ mora zapravo početi dužim stringom γ. Stoga u $\mathbf{T_{i+1}}$ može da bude put označen sa $xGlava(i)$ samo ako je tamo i put označen sa $x\gamma$, a ovo važi za svaki znak x. Stoga ako imamo put u $\mathbf{T_i}$ označen sa $xGlava(i)$ (potreban uslov da bi $I_w(x)$ bio 1) ali ne i put $x\gamma$, tada string koji razmatramo $xGlava(i)$ mora početi znakom na poziciji i u S. To znači da $Suff_{i+1}$ mora početi stringom $Glava(i)$. Ali s obzirom na to da w ima oznaku puta $Glava(i)$, list

$i+1$ mora da bude ispod w u $\mathbf{T_i}$ a takođe mora da bude ispod z u $\mathbf{T_{i+1}}$. Odnosno, z je na putu od korena do lista $i+1$. Međutim algoritam za konstrukciju $\mathbf{T_i}$ iz $\mathbf{T_{i+1}}$ počinje u listu $i+1$, i silazi do korena a kada nađe čvor v ili dostigne koren, indikator ulaska za x je postavljen na 1 na svakom čvoru puta od lista $i+1$. Hod se završava pre nego što je napravljen čvor w, tako da ne može da bude $I_z(x)=0$ u vreme kada je napravljen w. Tako da ako put $xGlava(i)$ posotoji u $\mathbf{T_i}$, tada $I_z(x)=1$ kada je w napravljen i teorema je dokazana. □

2.6.5 Analiza složenosti Vajnerovog algoritma

Vreme potrebno za konstrukciju $\mathbf{T_i}$ od $\mathbf{T_{i+1}}$ i korekcije vektora indikatora je proporcionalno vremenu za prelazak puta od lista $i+1$ do v' ili do korena. Prelazi se iz čvora do njegovog roditelja, a pošto u čvorovima postoje pokazivači na roditelja, za taj prelazak troši se konstantno vreme. Konstantno vreme je potrebno i za praćenje pokazivača L na sufiks (u daljem tekstu sufiksni pokazivač) i nakon opet konstatno vreme za dodavanje čvora w i grane (w,i). Stoga je vreme za konstrukciju $\mathbf{T_i}$ proporcionalno broju čvorova na koje se naiđe prilikom prelaska od lista $i+1$.
Prisetimo se da dubina čvora, čvora v je broj čvorova na putu u stablu od korena do v.
Za analizu vremena zamislimo da kako se algoritam izvršava pratimo koji smo čvor poslednji sreli i koja je njegova dubina čvora. Nazovimo dubinu čvora poslednjeg čvora kojeg smo sreli *trenutna dubina čvora*. Na primer, kada algoritam počne sa radom, trenutna dubina čvora je 1 i odmah nakon što je $\mathbf{T_m}$ kreirano, dubina čvora je 2. Jasno, kada algoritam krene da se penje po putu od lista na gore, trenutna dubina čvora opadne za jedan na svakom koraku. Takođe kada je algoritam u čvoru v'' (ili u korenu) i napravi novi čvor w ispod v'' (ili ispod korena), trenutna dubina čvora poraste za jedan. Jedino pitanje koje je preostalo da se reši jeste kako se trenutna dubina čvora menja kada je sufiksni pokazivač obiđen od čvora v' do v''.

Lema 2.2.1 Kada algoritam obiđe sufiksni pokazivač od čvora v' do čvora v'' u $\mathbf{T_{i+1}}$, trenutna dubina čvora poraste najviše za jedan.

Dokaz: Neka je u nekoreni čvor u $\mathbf{T_{i+1}}$ na putu od korena do v'' i pretpostavlja se da u ima oznaku puta $S(i)\alpha$ za neke neprazne stringove α. Svi čvorovi na putu od korena do v'' su ovog tipa, izuzev za jedan čvor (ako postoji) sa oznakom puta $S(i)$. Sada $S(i)\alpha$ je prefiks za $Suff_i$ i za $Suff_k$ za neko $k>i$, i ovaj string se različito produžava u dva slučaja.
S obzirom na to da je v' na putu od korena do lista $i+1$, α je prefiks za $Suff_{i+1}$ i tamo mora da bude čvor (moguće i koren) sa oznakom puta α na putu do v' u $\mathbf{T_{i+1}}$. Stoga put do v' ima odgovarajući čvor za svaki čvor na putu do v'', osim čvora (ako postoji) sa oznakom puta $S(i)$. Stoga dubina za v'' je najviše za jedan više nego dubina za v', iako bi mogla da bude i manja. □

Sledeća teorema daje ocenu složenosti Vajnerovog algoritma.

Teorema 2.2.4: Pretpostavljajući da postoji fiksan alfabet, Vajnerov algoritam konstruiše sufiksno stablo za string dužine m u vremenu $O(m)$.

Dokaz: Trenutna dubina čvora može da poraste za jedan svaki put kada je napravljen novi čvor i svaki put kada je obiđen sufiksni pokazivač. Stoga ukupan broj uvećanja u trenutnoj dubini čvora

je najviše $2m$. Sledi da trenutna dubina čvora može takođe da opadne najviše $2m$ puta s obzirom na to da trenutna dubina čvora počinje od nule i nikada nije negativna. Trenutna dubina čvora opada prilikom svakog penjanja, tako da je ukupan broj čvorova posećenih tokom svih penjanja najviše $2m$. Vreme za algoritam je proporcionalno ukupnom broju čvorova posećenih prilikom penjanja, tako da je time teorema dokazana. □

Poslednji komentar o Vajnerovom algoritmu

Ova diskusija o Vajnerovom algoritmu uspostavlja važnu činjenicu o sufiksnim stablima, u zavisnosti od toga kako su konstruisana:

Teorema 2.2.5: Ako je čvor v u sufiksnom stablu označen stringom $x\alpha$, gde je x pojedinačni znak, tada u stablu postoji čvor označen stringom α.

Ova činjenica je utvrđena kao Tvrđenje 2.1.2 tokom diskusije o Ukonenovom algoritmu.

3. Primene sufiksnih stabala

U tekstu koji sledi biće razmotreno nekoliko primena sufiksnih stabala. Većina ovih primena dozvoljava iznenađujuće efikasna rešenja linearne složenosti za kompleksne probleme sa stringovima.
Verovatno najbolji način za procenu moći sufiksnih stabala bi bio pokušaj da se niže navedeni problemi reše bez korišćenja sufiksnih stabala. Bez ovakvog napora ili bez istorijske perspektive, raspoloživost sufiksnih stabala bi mogla da stvori utisak trivijalnosti nekih od tih problema, iako linearni algoritmi za ove probleme nisu bili poznati pre pojave sufiksnih stabala. Interesantno je (videti [5]) da je Knut smatrao da za *problem nalaženja najdužeg zajedničkog podstringa* (odeljak 3.4) ne postoji linerni algoritam; međutim, ako se upotrebe sufiksna stabla, problem postaje skoro trivijalan.

3.1 Tačno traženje stringova

Postoje tri varijante ovog problema u zavisnosti od toga koji string (P ili T) je prvi poznat i fiksiran. Već smo videli upotrebu sufiksnih stabala na problem tačnog traženja. U tom slučaju upotrebom sufiksnih stabala dostiže se isti nivo složenosti u najgorem slučaju $\mathrm{O}(n+m)$, kao i kod KMP i Bojer-Murovog algoritma.
Međutim problem tačnog traženja se često javlja u situacijama kada je tekst T poznat i fiksiran izvesno vreme. Nakon što je tekst prošao kroz predobradu, na ulazu imamo dugi niz uzoraka i za svaki uzorak P u nizu traženje svih pojavljivanja P u T mora biti urađeno što je brže moguće. Označimo sa n dužinu P, a sa k broj pojavljivanja P u T. Koristeći sufiksno stablo za T, sva pojavljivanja mogu biti pronađena za vreme $\mathrm{O}(n+k)$, nezavisno od veličine T. Činjenica, da se bilo koji uzorak (nepoznat u fazi predobrade) može pronaći u vremenu proporcionalnom samo njegovoj dužini i to nakon utrošenog linearog vremena za predobradu T, bila je primarni motiv za razvijanje sufiksnih stabala. Nasuprot tome, algoritmi koji predobrađuju uzorak trošili bi vreme $\mathrm{O}(n+m)$ za traženje svakog pojedinačnog uzorka P.
Obrnuta situacija – kada je uzorak fiksiran i može da se predobrađuje pre nego što je tekst poznat, je klasična situacija koju rešavaju KMP i Bojer-Murov algoritam, radije nego sufiksna stabla. Ovi algoritmi troše vreme $\mathrm{O}(n)$ za predobradu, tako da traženje može da se završi u vremenu $\mathrm{O}(m)$ kad god je tekst T naveden. Mogu li sufiksna stabla da se koriste u ovom scenariju da bi se postigao isti novo složenosti? Iako to nije očigledno, odgovor je da.
Za problem tačnog traženja (pojedinačnog uzorka), sufiksna stabla mogu da se koriste da bi se dostigao isti nivo prostorne i vremenske složenosti kao kod KMP i Bojer-Murovog algoritma kada je uzorak prvi poznat ili kada su poznati i uzorak i tekst zajedno, ali sufiksna stabla su superiorna u najvažnijem slučaju kada je tekst prvi poznat i fiksiran, dok uzorci variraju.

3.2 Sufiksna stabla i problem tačnog traženja skupa uzoraka

Sada će biti razmotren problem tačnog traženja svih pojavljivanja stringova iz skupa P u tekstu T, gde je ulaz celokupan skup. Neka je n ukupna dužina stringova iz skupa P i neka je tekst T dužine m. Aho-Korasikov [1] metod nalazi sva pojavljivaja bilo kog uzorka iz P u T u vremenu $\mathrm{O}(n+m+k)$, gde je k broj pojavljivanja. Ova vremenska složenost se lako dostiže koristeći sufiksno stablo $\mathbf{T}$ za tekst T. U stvari, u prethodnom odeljku smo videli da kada je T prvi poznat

i fiksiran, a uzorak P varira, sva pojavljivanja bilo kog konkretnog P (dužine n) u T mogu biti pronađena za vreme $O(n + k_p)$, gde je k_p broj pojavljivanja u P. Stoga problem tačnog traženja skupa uzoraka može lako da se reši pomoću sufiksnog stabla, jer se skup P zadaje kada je tekst već poznat. Da bi se problem rešio, izgradimo sufiksno stablo **T** za string T u vremenu $O(m)$ i tada koristimo ovo stablo da bismo pronašli sva pojavljivanja svakog uzorka iz P. Celokupno vreme potrebno za ovaj pristup je $O(n + m + k)$.

Upoređivanje sufiksnih stabala i stabala ključnih reči za tačno traženje skupa uzoraka

Ovde upoređujemo relativne prednosti *stabala ključnih reči* [1] u odnosu na sufiksna stabla za tačno traženje skupa uzoraka. Iako je tačno da su vremenska i prostorna složenost ova dva metoda iste kada su skup P i string T zadati istovremeno, jedan od ovih metoda može da bude pogodniji od drugog u zavisnosti od odnosa veličina P i T, i od toga da li se predobrada može primeniti na P ili T. Aho-Korasikov metod koristi stabla sa ključnim rečima veličine $O(n)$, koja su izgrađena u vremenu $O(n)$ i pretražuju se u vremenu $O(m)$. Nasuprot tome, sufiksno stablo **T** je veličine $O(m)$, za izgradnju uzima vreme $O(m)$ a za traženje koristi vreme $O(n)$. Konstantni faktori za prostornu složenost i vreme traženja zavise od načina na koji su stabla predstavljena, ali oni zbog veličine utiču na praktične performanse.
U slučaju kad je skup uzorka veći nego tekst, pristup sa sufiksnim stablima koristi manji prostor, ali zahteva više vremena za traženje. U primenama u molekularnoj biologiji gde je uzorak mnogo veći od tipičnog teksta, Aho-Korasikov metod koristi manje prostora nego sufiksno stablo, ali sufiksno stablo troši manje vremena za traženje. Stoga ovde imamo razmenu vreme-prostor, i ni jedan metod nije uniformno bolji u odnosu na drugi u pogledu vremena i prostora.

3.3 Problem podstringa za bazu podataka uzoraka

Problem podstringa je predstavljen ranije u ovom radu. U najinteresantnijoj verziji ovog problema, zadat je skup stringova (baza podataka), posle čega za svaki zadati string S algoritam mora naći sve stringove u bazi podataka koji sadrže S kao podstring. Ovo je obratni problem u odnosu na traženje skupa uzoraka, gde treba pronaći koji od fiksiranih uzoraka je podstring zadatog stringa.
U kontekstu baza podataka sa genomskom DNK [4] problem traženja podstringa je problem koji ne može biti rešen tačnim upoređivanjem sa skupom uzoraka. Baza podataka DNK sadrži kolekciju prethodno sekvencionisanih DNK stringova. Kada se sekvencioniše novi DNK string, on može da bude sadržan u nekom već sekvencionisanom stringu, i potreban nam je neki efikasan metod da ovo proveri. (Naravno suprotan slučaj je takođe moguć, da novi string sadrži jedan od stringova iz baze podataka, ali to je slučaj tačnog traženja skupa uzoraka.)
Jedna malo morbidna primena problema podstringa je pojednostavljena verzija procedure koja je u aktuelnoj upotrebi pri identifikaciji posmrtnih ostataka pripadnika vojske SAD-a. Mitohondrijalna DNK se prikuplja od živih pripadnika vojske, i mali interval DNK svake osobe se sekvencioniše. Sekvencionisani interval ima dva ključna svojstva: može pouzdano da se izoluje tzv. *polimerizujućom lančanom reakcijom*, i DNK string u njemu je veoma promenljiv (t.j. skoro sigurno se razlikuje kod različitih ljudi). Taj interval se zbog toga koristi kao “skoro

jedinstven“ identifikator. Kasnije, ako je potrebno, mitohondrijalna DNK se uzima iz ostataka poginulih pripadnika vojske. Izolovanjem i sekvencionisanjem spomenutog intervala, string iz posmrtnih ostataka može da se uporedi sa bazom podataka stringova prikupljenih ranije (ili upoređen sa manjom bazom podataka stringova sastavljene od nestalih pripadnika vojske). Nakon toga pojavljuje se *podstring varijanta* ovog problema, jer stanje posmrtnih ostataka može da ne dopusti kompletno uzimanje ili sekvencionisanje određenog DNK intervala. U tom slučaju, kada ne postoji kompletan DNK interval, trebalo bi da se pogleda da li je izolovan i sekvencionisan string, zapravo podstring nekog od stringova iz baze podataka. Zbog grešaka je realnija varijanta problema kad hoćemo da izračunamo dužinu najdužeg zajedničkog podstringa za novoizvađenu DNK i neki od stringova u bazi podataka. Taj najduži zajednički podstring onda sužava izbor pri identifikaciji određene osobe.
Pretpostavlja se da je zbir m dužina svih stringova u bazi podataka veliki. Postavlja se pitanje izbora dobre strukture podataka i algoritma pretraživanja za problem podstringa. Postoje dva ograničenja: da ta baza podatka treba da bude malih dimenzija a pri tome svaka pretraga treba da bude relativno brza.
Sufiksna stabla za ovaj problem sa bazom podatka nude vrlo privlačno rešenje. Uopšteno sufiksno stablo **T** za stringove u bazi podataka se formira za vreme $\mathrm{O}(m)$, i što je još važnije, zahteva prostora veličine samo $\mathrm{O}(m)$. Pored toga, provera koji se string S dužine n nalazi u bazi podataka, ili ustanovljavanje da ga tamo nema, traje $\mathrm{O}(n)$ vremena. Kao i obično ovo se postiže upoređivanjem stringova sa putem S u stablu polazeći od korena. Kompletan string S se nalazi u bazi podatka ako i samo ako se putem traženja dostiže list stabla **T** u tački gde je pročitan poslednji znak iz S. Šta više, ako je S podstring stringa u bazi podataka, tada algoritam može da pronađe sve stringove u bazi podatka koja sadrži S kao podstring. Ovo zahteva vreme $\mathrm{O}(n+k)$, gde je k broj pojavljivanja podstringa. Postiže se obilaskom podstabla ispod kraja puta pređenog pri traženju S. Ako celi string S ne može da se poklopi sa putem u **T**, tada S nije u bazi podataka niti je tamo sadržan u nekom stringu. Međutim, pretražen put određuje najduži prefiks S koji je sadržan kao podstring u bazi podataka.
Problem podstringa je jedna od klasičnih primena sufiksnih stabala. Rezultati dobijeni korišćenjem sufiksnog stabla su jako dobri i ne bi mogli da se dostignu korišćenjem KMP, Bojer-Murovim niti Aho-Korasikovim algoritmom.

3.4 Najduži zajednički podstring dva stringa

Klasičan problem u analizi stringova je pronalaženje najdužeg zajedničkog podstringa za dva data stringa S_1 i S_2 (*problem najdužeg zajedničkog podstringa).* Na primer, ako je
$S_1 = superiorcalifornialives$ i $S_2 = sealiver$, tada najduži zajednički podstring za S_1 i S_2 glasi *alive*. Efiksan i konceptualno jednostavan način za pronalaženje najdužeg zajedničkog podstringa započinje formiranjem uopštenog sufiksnog stabla za S_1 i S_2. Svaki list stabla predstavlja ili sufiks jednog od dva stringa ili sufiks koji se pojavljuje u oba stringa. Svaki unutrašnji čvor v se označi sa 1(2) ako se u podstablu sa korenom u v nalazi list koji predstavlja sufiks iz S_1 (S_2). Oznaka puta svakog unutrašnjeg čvora označenog i sa 1 i sa 2 je zajednički podstring za oba stringa S_1 i S_2 i najduži takav string je najduži zajednički podstring. Stoga algoritam treba samo da pronađe čvor sa najvećom S-dubinom (sa najvećim brojem znakova na putu do njega) koji je označen i sa 1 i sa 2. Konstrukcija sufiksnog stabla obavlja se za linearno vreme (proporcionalno zbiru dužina S_1 i S_2), a označavanja i izračunavanja S-dubina čvorova

mogu da se obave standardnim obilaskom stabla linearne složenosti. Prema tome, dokazana je naredna teorema:

Teorema: Najduži zajednički podstring za dva stringa može da se pronađe algoritmom linearne složenosti koristeći uopšteno sufiksno stablo.

Iako problem nalaženja najdužeg zajedničkog podstringa sada izgleda trivijalno, s obzirom na naše sadašnje znanje o sufiksnim stablima, interesantno je zapaziti da je Knut još 1970 godine pretpostavio da linearni algoritam za ovaj problem nije moguć [5].
Prisetimo se sada problema identifikacije ljudskih posmrtnih ostataka spomenutih u prethodnom odeljku. Taj problem se može svesti na traženje najdužeg podstringa za zadati string i za drugi string, koji se na odgovarajući način formira od svih stringova iz baze podataka stringova.

4. Linearni algoritmi za konstrukciju sufiksnih nizova

4.1 Uvod

Sufiksna stabla i sufiksni nizovi su široko korištene i praktično ekvivalentne indeksne strukture za rad sa stringovima i nizovima. Ipak, do 2003. godine nije bio poznat algoritam za konstrukciju sufiksnog niza u linearnom vremenu bez korišćenja sufiksnog stabla. Algoritmi za direktnu konstrukciju sufiksnog niza objavljeni te godine omogućili su da se pomoću sufiksnih nizova rešavaju isti problemi za čije su se rešavanje do tada koristila sufiksna stabla. Zbog eksplicitnije strukture i postojanja linearnog algoritma za konstrukciju, teoretičari su uvek davali prednost sufiksnim stablima u odnosu na sufiksne nizove. S druge strane, praktičari često koriste sufiksne nizove, jer su efikasniji u pogledu prostora i jednostavniji za implementaciju. Pojavom prvog direktnog linearnog algoritma [6] za konstrukciju sufiksnog niza, sufiksni nizovi se podižu u istu ravan sa sufiksnim stablima. Nezavisno i simultano sa ovim rezultatom, pojavila su se još dva različita linearna algoritma [14, 17].

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

Primer: 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
```

Ako se zna originalni string, svaki sufiks je određen indeksom svog prvog znaka. Sufiksni niz je niz indeksa ovih sufiksa uređenih leksikografski. Za string "abrakadabra", sufiksni niz je {10,7,0,5,3,8,1,6,4,9,2}, jer sufiks "a" počinje na 11-tom znaku, "abra" počinje na 8-om znaku, itd.

U nastavku će biti prikazan jednostavan algoritam DC3 [13] za konstrukciju sufiksnog niza.
Ideja na kojoj se zasniva algoritam DC3 slična je ideji Farahovog algoritma [7] za konstrukciju sufiksnog stabla, pri čemu se koristi 2/3 - rekurzija umesto 1/2 - rekurzije:

1. Konstruiše se sufiksni niz za sufikse koji počinju na pozicijama i, takvim da je $i \bmod 3 \neq 0$. 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.

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 S_i i S_j, takvih da je $i \bmod 3 = 0$ i $j \bmod 3 = 1$, najpre se upoređuju prvi znaci, pa ako su isti, upoređuju se sufiksi S_{i+1} i S_{j+1}, čiji je relativni poredak već poznat iz prvog koraka.

Ovaj deo rada organizovan je na sledeći način. U odeljku 4.2 su navedene oznake koje će se koristiti u daljem tekstu. Odeljak 4.3 objašnjava osnovni algoritam linearne složenosti DC3. Na kraju odeljak 4.4 prikazuje implementaciju algoritma DC3 u okviru programa SufiksniNiz i test primer za isti program.

4.2 Oznake

Koristimo skraćenice $[i,j] = \{i,..,j\}$ i $[i,j) = [i,j-1]$ za raspone celih brojeva i prostire se do podstringa kao što se vidi niže u tekstu.
Ulaz algoritma za konstrukciju sufiksnog niza je string $T = T[0,n) = t_0t_1t_2...t_{n-1}$ nad alfabetom $[1,n]$, koji je niz od n celih brojeva iz raspona $[1,n]$. Zarad pogodnosti, pretpostavlja se da je $t_j = 0$ za $j \geq n$. Ponekad se takođe pretpostavlja da je n+1 umnožak neke male konstante v ili je potpuni kvadrat da bi smo izbegli razmatranje velikog broja slučajeva, odnosno operaciju $\lceil \cdot \rceil$. U implementacijama se ili ispituju ti različiti slučajevi ili se string dopunjuje odgovarajućim brojem nula znakova. Restrikcija alfabeta $[1,n]$ nije ozbiljno ograničenje. Za string T nad svakim alfabetom, najpre mogu da se sortiraju znaci iz T, uklone se duplikati, dodeli se rang svakom znaku i napravi se novi string T' nad alfabetom $[1,n]$ preimenovanjem znakova iz T njihovim rangovima. S obzirom na to da preimenovanje čuva poredak, poredak stringova se ne menja.
Za $i \in [0,n]$ neka S_i označava sufiks $T[1,\ n) = t_0t_1t_2...t_{n-1}$. Takođe se proširuje označavanje na skupove: za $C \subseteq [0,n]$, $S_c = \{S_i | \ i \in C\}$. Cilj je sortirati skup sufiksa $S_{[0,n]}$ iz T gde poređenje podstringova ili torki pretpostavlja leksikografski poredak.
Izlaz je sufiksni niz $SA_{[0,n]}$ iz T permutacija za $[0,n]$ koja zadovoljava relaciju
$S_{\mathrm{SA[0]}} < S_{\mathrm{SA[1]}} < \cdots < S_{\mathrm{SA[n]}}$.

4.3 Algoritam DC3 linearne vremenske složenosti

Počinjemo sa detaljnim opisom jednostavnog linearno-vremenskog algoritma, koji zovemo DC3 (za pokrivanje razlike po modulu 3). Izvršenje algoritma je ilustrovano narednim primerom

		0	1	2	3	4	5	6	7	8	9	10	11
$T[1,n]$	=	y	a	b	b	a	d	a	b	b	a	d	o

gde tražimo sufiksni niz $SA = (1,6,4,9,3,8,2,7,5,10,11,0)$.

Korak 0: Formiranje uzorka. Za $k = 0,1,2$ neka je

$$B_k = \{i \in [0,n] \mid i \bmod 3 = k\}.$$

Neka je $C = B_1 \cup B_2$ *skup izabranih pozicija* i S_c, *skup izabranih sufiksa*, odnosno skup sufiksa sa indeksima iz C.

Primer 1: Iz $B_1 = \{1, 4, 7, 10\}$, $B_2 = \{2, 5, 8, 11\}$, dobija se, $C = \{1,4,7,10,2,5,8,11\}$.

Korak 1: Sortiranje uzoraka sufiksa. Za $k = 1,2$ konstruišu se stringovi

$$R_k = [t_k t_{k+1} t_{k+2}][t_{k+3} t_{k+4} t_{k+5}] \dots [t_{maxB_k} t_{maxB_k+1} t_{maxB_k+2}]$$

čiji znaci su trojke $[t_i t_{i+1} t_{i+2}]$. Primetimo da je poslednji znak u R_k uvek jedinstven, jer je $t_{maxB_k+2} = 0$. Neka je $R = R_1 \odot R_2$ rezultat spajanja (konkatenacije) R_1 i R_2. Tada neprazni sufiksi iz R odgovaraju skupu S_c iz uzorka sufiksa: $[t_i t_{i+1} t_{i+2}]\ [t_{i+3} t_{i+4} t_{i+5}]$... koji odgovaraju S_i. Pridruživanje čuva poredak, što znači da sortiranjem sufiksa R dobijamo poredak uzorka sufiksa S_c.

Primer 2: $R = [abb]\ [ada][bba]\ [do0]\ [bba]\ [dab]\ [bad]\ [o00]$.

Da bi se sortirali sufiksi iz R, prvo se radiks sortiranjem sortiraju znaci iz R i preimenuju se njihovim rangovima da bi se dobio string R'. Ako su svi znaci različiti, poredak znakova daje direktno poredak sufiksa. Inače, sufiksi iz R' se sortiraju koristeći algoritam DC3.

Primer 3: $R' = (1,2,4,6,4,5,3,7)$ i $SA_{R'} = (8,0,1,6,4,2,5,3,7)$.

Pošto je uzorak sufiksa sortiran, pridružimo rang svakom sufiksu. Za $i \in C$, neka $rang(S_i)$ označava rang od S_i u uzorku skupa S_c. Dodatno, definišimo $rang(S_{n+1}) = rang(S_{n+2}) = 0$. Za $i \in B_0$, $rang(S_i)$ nije definisan.

Primer 4:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$rang(S_i)$	⊥	1	4	⊥	2	6	⊥	5	3	⊥	7	8	⊥	0	0

Korak 2: Sortiranje neuzoračkih sufiksa. Svaki neuzorački sufiks $S_i \in S_{B_0}$ se predstavlja parom $(t_i, rang(S_{i+1}))$. Primetimo da je $rang(S_{i+1})$ uvek definisan za $i \in B_0$. Jasno je da je ovde za sve $i, j \in B_0, S_i \leq S_j$ <=> $(t_i, rang(S_{i+1})) \leq (t_j, rang(S_{j+1}))$.

Parovi $(t_i, rang(S_{i+1}))$ su tada radiks sortirani.

Primer 5: $S_{12} < S_6 < S_9 < S_3 < S_0$ jer $(0, 0) < (a, 5) < (a, 7) < (b, 2) < (y, 1)$.

Korak 3: Objedinjavanje. Dva sortirana skupa sufiksa su objedinjena koristeći standardno spajanje zasnovano na upoređivanju. Da bi smo uporedili sufiks $S_i \in S_c$ sa $S_j \in S_{B_0}$, razlikujemo dva slučaja:
$i \in B_1$: $S_i \leq S_j$ <=> $(t_i, rang(S_{i+1})) \leq (t_j, rang(S_{j+1}))$
$i \in B_2$: $S_i \leq S_j$ <=> $(t_i, t_{i+1}, rang(S_{i+2})) \leq (t_j, t_{j+1}, rang(S_{j+2}))$

Primetimo da su rangovi definisani u svim slučajevima.

Primer 6: $S_1 < S_6$ jer $(a, 4) < (a, 5)$ i $S_3 < S_8$ jer $(b, a, 6) < (b, a, 7)$.

Vremenski nivo složenosti je ustanovljen narednom teoremom [8].

Teorema 1: Vremenska složenost algoritma DC3 je $O(n)$.

Dokaz: Izuzev rekurzivnog poziva, sve može da se uradi u linernom vremenu. Rekurzija je na stringu dužine $[2n/3] + O(n)$. Stoga je složenost data rekurentonom relacijom $T(n) = T(2n + 3) + O(n)$, čije rešenje je $T(n) = O(n)$. □

4.4 Implementacija algoritma DC3

U dodatku rada nalazi se program SufiksniNiz koji implementira rad DC3 algoritma. Jedan deo programa - modul DC3.c, preuzet je iz rada [2].

Program služi za nalaženje sufiksnog niza (SA) zadatog stringa i nalaženje pozicija zadatog podstringa (koristeći sufiksni niz SA).
String se zadaje neposredno ili putem pripremljenog fajla dok se podstring zadaje samo neposredno.

Korisnik mora da ima na umu da je shodno uštedi vremena i prostora prilikom ispisivanja rezultata, program napravljen tako, da ispisuje rezultate samo za string manji od 1000 znakova. U protivnom rezultati neće biti prikazani na ekranu – samo pozicije podstringa koji se traži i vreme potrebno za konstrukciju sufiksnog niza.

U narednom segmentu sledi uputstvo za pravilno pokretanje i korišćenje programa.

Uputstvo za pokretanje i korišćenje programa:

Upotreba: SufiksniNiz <cmd> <string | ime fajla> <podstring>

Primeri pravilnog unosa parametara:

Neposredni string: SufiksniNiz s mojstring podstring
String iz fajla: SufiksniNiz f putanja_do_fajla podstring

<cmd> - Naredba. Naredba je prvi argument u komandnoj liniji.
Moguće naredbe su:

s - ako je naredba s, string koji se obrađuje je drugi argument u komandnoj liniji.

f - ako je naredba f, string koji se obrađuje se nalazi u fajlu čija je putanja drugi argument u komandnoj liniji.

<string | ime_fajla> - string koji se obrađuje ili putanja do fajla koji ga sadrži u zavisnosti od prvog argumenta u komandnoj liniji (vidi <cmd>).

<podstring> - podstring je treći argument u komandnoj liniji koji se traži nakon što je konstrukcija SA završena.

Funkcije koje se koriste u programu:

Main – glavna funkcija koja upravlja programom.

PrintUsage – ispisuje moguće načine pravilnog unosa prametara programa

PrintV – ispisuje na ekran kolone Redni broj: s: SA:

isPermutation – proverava da li je dobijeni niz već postojeća permutacija

leq – upoređuje nizove leksikografski

isSorted – proverava da li je dobijeni niz sortiran a ukoliko nije dodatno ga sortira

Find_Substring – nalazi na kojim pozicijama se nalazi podstring unutar zadatog stringa

Suffix_Array – na osnovu zadatog stringa nalazi sufiksni niz SA

Radix_Pass – funkcija unutar funkcije Suffix_Array koja radi radiks sortiranje sufiksnog niza

Ulaz i izlaz programa:

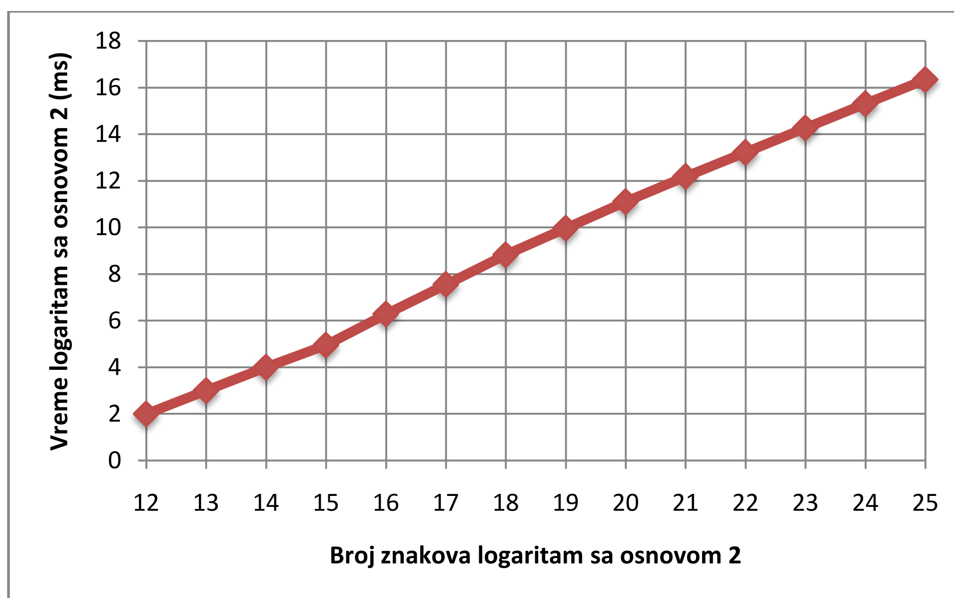
Ulazni parametri programa: izvorni string i podstring kome se traže pozicije pojavljivanja u stringu.
Izlazni parametri programa: sufiksni niz i pozicije na kojima se nalazi podstring (koristeći sufiksni niz).

Karakteristike računara na kojem je program testiran:

Procesor: 1.61 GHz AMD Sempron(tm) 3000+
RAM memorija: 1.5 GB

Dijagram merenja rezultata:

Neka je x veličina ulaznog niza, a y vreme kostrukcije sufiksnog niza u milisekundama. Na sledećem dijagramu prikazana je zavisnost $\log_2 x$ (veličine u rasponu od 12 do 25) od $\log_2 y$.



Slika 4.1 Prikaz rezultata merenja potrebnog za konstrukciju sufiksnog niza SA u zavisnosti od dužine ulaznog niza

Ocena koeficjenta pravca prave k:

Sa dijagrama se može oceniti koeficjent pravca prave k kroz pravu i pretposlednju tačku:

$$\log y = \log y_0 + \ k\ (\log x - \log x_0)$$
$$\log_2 y = \log_2 y_0 + \ k\ (\log_2 x - \log_2 x_0)$$
$$\log_2 y - \log_2 y_0 = \ k\ (\log_2 x - \log_2 x_0)$$
$$k = \frac{\log_2 y - \log_2 y_0}{\log_2 x - \log_2 x_0} = \frac{14.61 - 2}{24 - 12} = \frac{12.61}{12} = 1.05$$

Iz toga sledi procena $y = \frac{y_0}{x_0^{1.05}} x^{1.05}$, što se može smatrati potvrdom da je složenost algoritma linearna.
Sledi primer na kojem je testiran rad programa:

Test primer:

Ulaz: String $S = yabbadabbado$
Izlaz: Sufiksni niz $SA = (1, 6, 4, 9, 3, 8, 2, 7, 5, 10, 11, 0)$

Prikaz rezultata na ekranu za dati string:

```
Red. broj:          s:        SA:

         0           y          1 abbadabbado
         1           a          6 abbado
         2           b          4 adabbado
         3           b          9 ado
         4           a          3 badabbado
         5           d          8 bado
         6           a          2 bbadabbado
         7           b          7 bbado
         8           b          5 dabbado
         9           a         10 do
        10           d         11 o
        11           o          0 yabbadabbado
```

5. Primene sufiksnih nizova

Prilikom obrade dugačkih stringova, kao što su na primer genomi, memorijski prostor koji zahtevaju sufiksna stabla može da predstavlja nedostatak. Ispostavlja se da je svaki algoritam sa stringovima koji se zasniva na obilasku sufiksnog stabla, moguće zameniti odgovarajućim algoritmom koji koristi prošireni sufiksni niz. Pored toga, algoritmi sa proširenim sufiksnim nizovima su lakši za implementaciju od algoritama sa sufiksnim stablima. Pokazaćemo kako se efikasno mogu pronalaziti *maksimalna*, *supermaksimalna* i *tandem ponavljanja*, kao i *maksimalna jedinstvena poklapanja*. Činjenica da prošireni sufiksni nizovi zahtevaju mnogo manje prostora nego sufiksna stabla, omogućuje da se indeksiraju i analiziraju veoma veliki genomi, što nije bilo izvodljivo pomoću sufiksnih stabala. Eksperimenti pokazuju da program koji koristi sufiksni niz ne samo da zahteva manje prostora, nego i manje vremena u odnosu na programe koji obavljaju iste zadatake korišćenjem sufiksnog stabla.

5.1 Motivacija – primene u analizi genoma

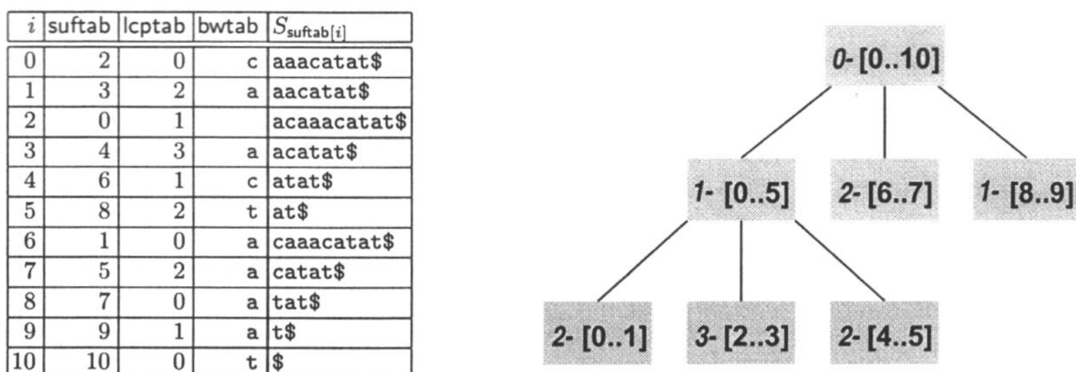
Analiza ponavljanja je značajan deo proučavanja, analize i upoređivanja kompletnih genoma. U analizi pojedinačnog genoma osnovni zadatak je da se okarakterišu i pronađu njegovi ponovljeni elementi (ponavljanja).
Ponavljanja mogu da se podele u dve velike grupe: razmaknuta i spojena ponavljanja DNK i mogu da čine veliki deo genoma. Na primer, 50% od 3 milijarde *baznih parova* (bp) ljudskog genoma su ponavljanja. Ponavljanja takođe sadrže 11% genoma gorčice, 7% genoma gliste i 3% genoma muve. Za sistematsko proučavanje ponavljanja u genomima potrebna je ozbiljna softverska podrška, kao što su programi REPuter [11] i MUMmer [12]. REPuter-a nalazi maksimalna ponavljanja primenom algoritama linearne prostorne i vremenske složenosti. Iako je zasnovan na efikasnoj i kompaktoj implementaciji sufiksnih stabala, potreba za memorijskim prostorom je ipak prilično velika, prosečno 12.5 bajtova po ulaznom znaku. Šta više, u širokoj paleti primena, sufiksno stablo je manje efikasno zbog slabe lokalizacije memorijskih referenci.
Isti problem se pojavljuje u kontekstu upoređivanja genoma. U novije vreme, DNK sekvence kompletnih genoma određuju se velikom brzinom. Kada genomske DNK sekvence bliže srodnih organizama postanu dostupne, jedno od prvih pitanja koje istraživači postave jeste koliko su im genomi slični. Ova sličnost može da pomogne na primer u razumevanju zašto je jedna vrsta bakterija patogena ili otporna na antibiotike dok druga to nije. Softverski alat MUMer je razvijen da bi se efikasno uporedila dva slična genomska DNK niza. U prvoj fazi MUMmer izračunava takozvanu MUM dekompoziciju dva genoma S_1 i S_2. MUM (maksimalno jedinstveno poklapanje, engl. maximal unique match, MUM) je niz koji se javlja tačno po jednom u genomima S_1 i S_2, i nije sadržan ni u jednom dužem nizu sa istom osobinom. Koristeći sufiksna stabla za $S_1\#S_2$, izračunavanje MUMova ima prostornu i vremensku složenost $O(n)$, gde je $n = |S1\#S2|$, a # je simbol koji se ne pojavljuje ni u S_1 ni u S_2. Međutim, zahtev sufiksnih stabala za memorijskim prostorom je glavni problem prilikom upoređivanja velikih genoma.
Dakle, iako su sufiksna stabla nesumnjivo jedna od najvažnijih struktura podataka u analizi nizova znakova, njihovi memorijski zahtevi su usko grlo u zadacima analize velikih genoma.
Postoje strukture podataka koje su efikasije u pogledu korišćenja memorijskog prostora nego sufiksna stabla. Najvažniji je sufiksni niz, koji u svom osnovnom obliku zahteva samo 4n bajtova. U većini praktičnih primena, sufiksni nizovi mogu da se smeste u spoljašnju memoriju

jer im se pristupa sekvencijalno. Konstrukcija sufiksnog niza ima složenost O(n) (utvrđeno u prethodnom poglavlju). Dalje, traženje uzoraka zasnovano na sufiksnim nizovima je uporedivo sa traženjem uzoraka zasnovanom na sufiksnim stablima.
Ovde se najpre predstavlja struktura podataka dobijena dodavanjem *lcp*-intervalnog stabla sufiksnom nizu. Po uspostavljanju ove strukture podataka, fokus se premešta na algoritme koji efikasno izračunavaju različite vrste ponavljanja. Preciznije, biće pokazano kako efikasnije izračunati maksimalno, supermaksimalno i uzastopno ponavljanje stringa S koristeći odozdo naviše formiranje *lcp*-intervalnog stabla **S**. Ovo stablo je samo konceptualno, u smislu da eksplicitno ne postoji za vreme formiranja odozdo naviše (na primer u bilo kojoj fazi reprezentacija samo jednog malog dela lcp-intervalnog stabla se nalazi u glavnoj memoriji).
Gore spomenuti programi zahtevaju manje vremena i prostora nego drugi programi za isti zadatak. Međutim ova struktura podataka nije ograničena na gore spomenute primene. Na primer, izračunavanje Lempel-Ziv (Lempel-Ziv) dekompozicije stringa (koja je važna u kompresiji podataka) može takođe da bude implementirano koristeći ovu strukturu podataka.

5.2 Osnovne napomene

Neka je S string dužine $|S| = n$ nad alfabetom Σ. Da bi se pojednostavila analiza, pretpostavlja se da je veličina alfabeta konstantna i da je $n < 2^{32}$. To znači da svaki ceo broj u rasponu $[0, n]$ može da se uskladišti u 4 bajta. Pretpostavlja se da je specijalni simbol $ element iz Σ (koji je veći nego svi drugi elementi Σ), ali se ne pojavljuje u S. $S[i]$ označava znak na poziciji i u S, za $0 \leq i < n$. Za $i \leq j$, $S[i..j]$ označava podstring od S koji počinje na poziciji i a završava se znakom na poziciji j. Podstring $S[i..j]$ je takođe označen parom pozicija (i, j).
Par podstringova $R = ((i_1, j_1), (i_2, j_2))$ je ponavljanje ako i samo ako $((i_1, j_1) \neq (i_2, j_2))$ i $S[i_1..j_1] = S[i_2..j_2]$. Dužina R je $j_1 - i_1 + 1$. Ponavljanje $((i_1, j_1), (i_2, j_2))$ se zove *levo maksimalno* ako $S[i_1 - 1] \neq S[i_2 - 1]$, [2] odnosno *desno maksimalno* ako $S[j_1 + 1] \neq S[j_2 + 1]$. Ponavljanje se zove *maksimalno* ako je i levo i desno maksimalno. Podstring ω iz S je (maksimalno) ponovljeni podstring ako tamo postoji (maksimalno) ponavljanje $((i_1, j_1), (i_2, j_2))$ takvo da $\omega = S[i_1..j_1]$. Super maksimalno ponavljanje je maksimalno ponavljanje koje se nikad ne pojavljuje kao podstring u bilo kom drugom maksimalnom ponavljanju. String je tandemsko ponavljanje ako može da se napiše kao ωω za neki neprazan string ω. Pojavljivanje tandemskog ponavljanja $\omega\omega = S[p..p + |2\omega| - 1]$ koje je predstavljeno parom $(|\omega|, p)$ je *grananje* ako $S\,[p + |\omega|] \neq S\,[p + 2|\omega|]$.

[2] Ova definicija bi trebalo da se proširi i na slučajeve $i_1 = 0$ ili $i_2 = 0$, ali ovaj rad se ne bavi tim graničnim slučajevima.



i	suftab	lcptab	bwtab	$S_{suftab[i]}$
0	2	0	c	aaacatat$
1	3	2	a	aacatat$
2	0	1		acaaacatat$
3	4	3	a	acatat$
4	6	1	c	atat$
5	8	2	t	at$
6	1	0	a	caaacatat$
7	5	2	a	catat$
8	7	0	a	tat$
9	9	1	a	t$
10	10	0	t	$

Slika 5.1 Prošireni sufiksni niz stringa S = acaaacatat$ i njegovo lcp-intervalno stablo

Sufiksni niz $\boldsymbol{suftab}$ (videti poglavlje 4) je celobrojni niz sa indeksima od 0 do n, koji određuje leksikografski poredak *n+1* sufiksa stringa S\$. Drugim rečima $S_{suftab[0]}, S_{suftab[1]}, \dots, S_{suftab[n]}$ je niz sufiksa S\$ u rastućem leksikografskom poretku, gde $S_i = S[i..n-1]\$$ označava i-ti neprazni sufiks stringa S\$, $0 \leq i \leq n$. Sufiksni niz zahteva 4n bajtova za smeštanje u memoriji.

Tabela $\boldsymbol{bwtab}$ je tabela veličine n+1 takva da je za svako $\boldsymbol{i}$, $0 \leq i \leq n$, $bwtab[i] = S[suftab[i]-1]$ ako $suftab[i] \neq 0$; $bwtab[i]$ je nedefinisano ako je $suftab[i] = 0$. $Bwtab$ je Barous-Vilerova (Burrows–Wheeler) transformacija [10]. Ona je uskladištena u n bajtova i konstruisana u jednom prolasku kroz $suftab$ u vremenu O(n).

Lcp tabela $\boldsymbol{lcptab}$ je celobrojni niz dužine $n+1$. Po definiciji je $lcptab[0] = 0$ a $lcptab[i]$ se definiše dužina najdužeg zajedničkog prefiksa za $S_{suftab[i-1]}$ i $S_{suftab[i]}$, za $1 \leq i \leq n$. S obzirom na to da je $S_{suftab[n]} = \$$, uvek je $lcptab[n] = 0$ (videti sliku 5.1). Lcp tabela zahteva 4n bajtova u najgorem slučaju. Međutim u praksi može da se implementira korišćenjem nešto više od n bajtova. Preciznije, uskladišti se većina vrednosti tabele $\boldsymbol{lcptab}$ u tabelu $\boldsymbol{lcptab_1}$ koristeći n bajtova, tako da je za svako $i \in [1, n]$, $lcptab_1[i] = max\{255, lcptab[i]\}$. Obično postoji samo nekoliko elemenata $lcptab$ koji su veći ili jednaki od 255. Da bi im se efikasnije pristupalo, ovi elementi se smeštaju u posebnu tabelu $llvtab$. Ona sadrži sve parove $(i, lcptab[i])$ takve da $lcptab[i] \geq 255$; spisak tih parova zadat je prvom tabelom $\boldsymbol{lcptab_1}$. Na mestu sa indeksom i u tabeli $lcptab_1$ zapisuje se 255 uvek kada je $lcptab[i] \geq 255$. Tačna vrednost iz *lcptab* pronalazi se tada u $llvtab$. Ako se tabela $lcptab_1$ pregleda sekvencijalno, kada pronađemo vrednost 255, tada se tačna vrednost iz *lcptab* nalazi u narednom slogu tabele $llvtab$. Ako se vrednostima u $lcptab_1$ pristupa proizvoljnim redosledom, kada se pronađe vrednost 255 na poziciji i, tada se vrednost pronalazi binarnom pretragom tabele $llvtab$, koristeći i kao ključ. Tako se $lcptab[i]$ dobija za vreme $O(\log_2 |llvtab|)$.

5.3 Lcp intervalno stablo sufiksnog niza

Definicija 1: Interval $[i..j]$, $0 \leq i < j \leq n$, je lcp-interval lcp-vrednosti l ako

1. $lcptab[i] < l$
2. $lcptab[k] \geq l$ *za sve k sa* $i + 1 \leq k \leq j$,
3. $lcptab[k] = l$ *za barem jedno k sa* $i + 1 \leq k \leq j$,
4. $lcptab[j + 1] < l$.

Takođe se koristi skraćenica *l-interval* (ili l-$[i..j]$ za lcp-interval $[i..j]$ *lcp*-vredosti l. Svaki indeks k, $i + 1 \leq k \leq j$, sa $lcptab[k] = l$ se naziva *l-indikator*. Skup svih *l-indikatora l-intervala* $[i..j]$ biće označen sa $lIndikatori(i,j)$.
Ako je $[i..j]$ *l-interval* takav da je $\omega = S\,[suftab[i]..suftab[i] + l - 1]$ najduži zajednički prefiks sufiksa $S_{suftab[i]}, S_{suftab[i+1]}, \ldots S_{suftab[j]}$, tada se $[i..j]$ takođe zove *ω-interval*.

Primer 1: Razmotrimo tabelu na slici 5.1 $[0..5]$ je *1-interval* jer $lcptab[0] = 0 < 1$, $lcptab[5 + 1] = 0 < 1$, $lcptab[k] \geq 1$ za sve k iz $1 \leq k \leq 5$, recimo $lcptab[2] = 1$. Dalje, $lIndikator(0,5) = \{2,4\}$.

Kasai (Kasai) [9] je prikazao algoritam linearne složenosti za fiktivni obilazak sufiksnog stabla odozdo naviše, koristeći sufiksni niz i *lcp* tabelu. Naredni algoritam predstavlja neznatnu modifikaciju tog algoritma prikazanog u [9]. Algoritam određuje sve lcp intervale predstavljene trojkama <*lcp, lg, dg*>, gde je *lcp* - lcp vrednost intervala, *lg* je njena leva granica, a *dg* je njena desna granica. U algoritmu 2, *push* (stavlja element na stek) a *pop* (skida element sa vrha steka i vraća taj element) su uobičajene operacije sa stekom; *vrh* je pokazivač na vrh steka.

Algoritam 2 Određivanje *lcp*-intervala

```
push (<0, 0, ⊥>)
for i : = 1 to n do
    lg : = i – 1
    while lcptab[i] < vrh.lcp
        vrh.dg : = i – 1
        interval : = pop
        stampaj(interval)
        lg : =  interval.lg
    if lcptab[i] > vrh.lcp then
        push (<lcptab[i], lg, ⊥>)
```

Primer 1: Ilustracija algoritma 2 pomoću vrednosti sa slike 5.1. Posmatrajmo proces dobijanja svih intervala (osim korenog) kojih na slici 5.1 ima šest. Stek je u formi trojki $< lcp, lg, dg >$ a ⊥ je nedefinisan simbol koji nije deo alfabeta.

i	*Stanje steka*	*lg*	*lcptab[i]*	*lcptab[i] < vrh.lcp*	*Interval za štampu*
1	<0, 0, ⊥>	0	2	2 < 0 ne	
	<0, 0, ⊥>, <2, 0, ⊥>	0	2	2 > 0 da	
2	<0, 0, ⊥>, <2, 0, ⊥>	1	1	1 < 2 da	
	<0, 0, ⊥>, <2, 0, 1>	0			<2, 0, 1>
	<0, 0, ⊥>	0		1 < 0 ne	
3	<0, 0, ⊥>	2	3	3 < 0 ne	
	<0, 0, ⊥>, <3,2, ⊥>	2	3	3 > 0 da	
4	<0, 0, ⊥>, <3, 2, ⊥>	3	1	1 < 3 da	
	<0, 0, ⊥>, <3, 2, 3>	2			<3, 2, 3>
5	<0, 0, ⊥>, <1, 0, ⊥>	4	2	2 < 1 ne	
	<0, 0, ⊥>, <1, 0, ⊥>, <2, 4, ⊥>	4	2	2 > 1 da	
6	<0, 0, ⊥>, <1, 0, ⊥>, <2, 4, ⊥>	5	0	0 < 2 da	
	<0, 0, ⊥>, <1, 0, ⊥>, <2, 4, 5>	0			<2, 4, 5>
	<0, 0, ⊥>, <1, 0, ⊥>	0		0 < 1 da	
	<0, 0, ⊥>, <1, 0, 5>	0			<1, 0, 5>
	<0, 0, ⊥>	5	1	0 > 0 ne	
7	<0, 0, ⊥>	6	2	2 < 0 ne	
	<0, 0, ⊥>, <2, 6, ⊥>	6	2	2 > 0 da	
8	<0, 0, ⊥>, <2, 6, ⊥>	7	0	0 < 2 da	
	<0, 0, ⊥>, <2, 6, 7>	6			<2, 6, 7>
9	<0, 0, ⊥>	8	1	1 < 0 ne	
	<0, 0, ⊥>, <1, 8, ⊥>	8	1	1 > 0 da	
10	<0, 0, ⊥>, <1, 8, ⊥>	9	0	0 < 1 da	
	<0, 0, ⊥>, <1, 8, 9>	8			<1, 8, 9>

Uvedimo sada koncept *lcp*-intervalnog stabla:

Definicija 2: Za *m-interval* $[l..r]$ kaže se da je zatvoren u *l-intervalu* $[i..j]$ ako je podinterval intervala $[i..j]$ (tj. $i \leq l < r \leq j$) $i\ m > l$.[3] Tada kažemo da *l-interval* $[i..j]$ obuhvata interval $[l..r]$. Ako $[i..j]$ obuhvata interval $[l..r]$ i ne postoji interval obuhvaćen intervalom $[i..j]$ koji obuhvata interval $[l..r]$, tada je $[l..r]$ sin interval od $[i..j]$.

Ovaj odnos otac-sin uspostavlja virtuelno stablo, *lcp*-intervalno stablo sufiksnog niza. Koren ovog stabla je *0-interval* $[0..n]$, (videti primer 1). Lcp-intervalno stablo je u osnovi sufiksno stablo bez listova. Listovi su implicitno sadržani u ovoj strukturi podataka: svaki list u sufiksnom stablu koji odgovara sufiksu $S_{suftab[j]}$, može da se predstavi jednočlanim intervalom $[l..l]$. Otac takvog intervala je najmanji lcp-interval $[i..j]$ koji sadrži l. Na primer, nastavljajući primer sa slike 5.1, sinovi intervala $[0..5]$ su $[0..1]$ $[2..3]$ i $[4..5]$. Naredna teorema pokazuje kako odnosi otac-sin između lcp-intervala mogu da se odrede operacijama nad stekom u Algoritmu 2.

[3] Primetimo da ne možemo da imamo oba $i = l$ i $r = j$, jer je $m > l$.

Teorema 1: Razmotrimo **for**-petlju Algoritma 2 za neke indekse i. Neka je *vrh* najviši mogući interval na steku i neka je $(vrh-1)$ interval ispod njega (primetimo da $(vrh-1).lcp < lcptab[i]$).
1. Ako $lcptab[i] \leq (vrh-1).lcp$, tada je *vrh* sin intervala od $(vrh\ -1)$.
2. Ako $(vrh-1).lcp < lcptab[i] < vrh.lcp$, tada je *vrh* sin onog $lcptab[i]$-intervala koji sadrži i.

Primer 2: Ilustracija prethodne teoreme pomoću vrednosti sa slike 5.1. Ako se pogleda tabela iz prethodnog primera vidi se da za $i=4$, $lcptab[4]=1$, $vrh.lcp=3$ dok je $(vrh-1).lcp=0$ dok interval koji se štampa za ovu vrednost je <3, 2, 3>. Pošto je $0 < 1\ < 3$ time je zadovoljen drugi uslov iz teoreme. Na osnovu toga se može se reći da će interval <3, 2, 3> biti sin intervala koji sadrži ovu vrednost i, a jedini interval koji ovde zadovoljava taj uslov je <1, 0, 5>.

Važna posledica teoreme 1 je tačnost algoritma 3. U tom algoritmu su elementi na hipu lcp-intervali, predstavljeni četvorkama $< lcp, lg, dg, sinList >$, gde je *lcp* lcp-vrednost intervala, *lg* je njegova leva granica, *dg* je njegova desna granica a *sinList* je lista intervala - njegovih sinova. Dalje, $dodaj([c_1,\ldots,c_k],c)$ dodaje element c u listu $[c_1,\ldots,c_k]$.

Algoritam 3: Formiranje i obrada lcp-intervalnog stabla

```
poslednjiInterval : = ⊥
push(<0, 0, ⊥, [ ]>)
for i : =  to n do
        lg : = i -1
        while  lcptab[i] < vrh.lcp
                vrh.dg : = i-1
                poslednjiInterval : = pop
                obradi (poslednjiInterval)
                lg : = poslednjiInterval.lg
                if lcptab[i] ≤ vrh.lcp then
                        vrh.sinList : = dodaj(vrh.sinList, poslednjiInterval)
                        poslednjiInterval : = ⊥
        if lcptab[i] ≤ vrh.lcp then
                if poslednjiInterval ≠ ⊥ then
                        push (<lcptab[i], lg, ⊥, [poslednjiInterval]>)
                        poslednjiInterval : =  ⊥
                else push (<lcptab[i], lg, ⊥, [poslednjiInterval]>)
```

Primer 3: Ilustracija algoritma 2 pomoću vrednosti sa slike 5.1. Posmatrajmo proces dobijanja svih intervala (osim korenog) kojih na slici 5.1 ima šest. Stek je u formi četvorki $< lcp, lg, dg, [\,] >$ a ⊥ je nedefinisan simbol koji nije deo alfabeta, dok [ ] je prazna lista.

Početna vrednost za poslednjiInterval je ⊥. Početna vrednost steka je $< 0, 0, \perp, [\,] >$.

i	*Stanje vrha steka*	*lg*	*lcptab[i]*	*lcptab[i] < vrh.lcp*	*lcptab[i] ≤ vrh.lcp*	*Interval za obradu*
1	<0, 0, ⊥, [ ]>	0	2	2 < 0 ne	2 ≤ 0 ne	
2	<2, 0, 1, [ ]>	1	1	1 < 2 da	1 ≤ 2 da	
	<2, 0, 1, [<2, 0, 1, [ ]>]>	0				<2, 0, 1, [ ]>
3	<0, 0, ⊥, [ ]>	2	3	3 < 0 ne	3 ≤ 0 ne	
4	<3, 2, ⊥, [ ]>	3	1	1 < 3 da	1 ≤ 3 da	
	<3, 2, 3, [<3, 2, 3, [ ]>]>	2				<3, 2, 3, [ ]>
5	<0, 0, ⊥, [ ]>	4	2	2 < 0 ne	2 ≤ 0 ne	
6	<2, 4, ⊥, [ ]>	5	0	0 < 2 da	0 ≤ 2 da	
	<2, 4, 5, [<2, 4, 5, [ ]>]>	4	1			<2, 4, 5, [ ]>
	<1, 0, ⊥, [ ]>	5	1	1 < 2 da	1 ≤ 2 da	
	<1, 0, 5, [<1, 0, 5, [ ]>]>	0				<1, 0, 5, [ ]>
7	<0, 0, ⊥, [ ]>	6	2	2 < 0 ne	2 ≤ 0 ne	
8	<2, 6, ⊥, [ ]>	7	0	0 < 2 da	0 ≤ 2 da	
	<2, 6, 7, [<2, 6, 7, [ ]>]>	6				<2, 6, 7, [ ]>
9	<0, 0, ⊥, [ ]>	8	1	1 < 0 ne	1 ≤ 0 ne	
10	<1, 8, ⊥, [ ]>	9	0	0 < 1 da	0 ≤ 1 da	
	<1, 8, 9, [<1, 8, 9, [ ]>]>	8				<1, 8, 9, [ ]>

U algoritmu 3, lcp-intervalno stablo se prolazi odozdo naviše linearnim prolaskom *lcptab*, a potrebne informacije se usput smeštaju na stek. Treba napomenuti da se lcp-intervalno stablo ne formira eksplicitno: uvek kada je *l-interval* obrađen generičkom funkcijom *obradi*, poznati treba da budu samo njegovi sinovi intervali. Oni su određeni samo iz lcp-informacija, tako da eksplicitnih pokazivača otac-sin nema u ovoj strukuri podataka. Nasuprot algoritmu 2, algoritam 3 izračunava sve lcp-intervale lcp-tabele sa informacijama o sinovima. U nastavku se pokazuje rešavanje nekoliko problema preciziranjem funkcije *obradi* pozvane u liniji 8 algoritma 3.

5.4 Efikasna implementacija algoritma za nalaženje maksimalnih ponavljanja

Algoritam za nalaženje svih maksimalnih parova izračunava maksimalno ponavljanje u nizu S ([1], odeljak 7.12.3, strana 147). Izračunavanje se izvršava u vremenu $O(kn+z)$ gde je $k=|\Sigma|$ a z je broj maksimalnih ponavljanja. Ovo vreme izvršavanja je optimalno. Ovaj algoritam je implementiran u programu REPuter koji je zasnovan na efikasnim sufiksnim stablima [11]. U ovom odeljku se pokazuje kako implementirati algoritam koristeći proširene sufiksne nizove. Ovo značajno smanjuje potrebu za memorijom, pa dakle uklanja usko grlo u algoritmu. Zahvaljujući tome, mogu se tražiti ponavljanja u mnogo većim genomima. Implementacija zahteva tabele *suftab*, *lcptab*, *bwtab*, ali ne pristupa ulaznom stringu. Ovim trima tabelama se pristupa sekvencijalno, što dovodi do poboljšane koherencije keš memorije, odnosno značajno se smanjuje vreme izvršavanja.

Uvešćemo najpre neke oznake. Neka je ⊥ simbol za nedefinisani znak. Pretpostavlja se da se ⊥ razlikuje od svih znakova u Σ. Neka je $[i..j]$ *l-interval* u $S[suftab[i] \dots suftab[i]+l-1]$. i neka je u zajednički prefiks sufiksa $S[suftab[k], k=i,i+1,\dots,j$. Definišimo $P_{[i..j]}$ kao skup pozicija p takvih da je u prefiks S_P, tj.

$P_{[i..j]} = \{suftab[r] \mid i \le r \le j\}$. Podelimo $P_{[i..j]}$ na disjunktne i eventualno prazne skupove shodno znacima levo od svake pozicije: za svako $a \in \Sigma \cup \{\bot\}$ definišimo:

$$P_{[i..j]}(a) = \begin{cases} \{0 \mid 0 \in P_{[i..j]}\}, \text{ ako } a = \bot \\ \{p \in P_{[i..j]} \mid p > 0 \text{ i } S[p-1] = a\}, \text{ inače.} \end{cases}$$

Primer 4: Uzmimo za primer jedan od l-intervala sa slike 5.1 recimo $3[2,3]$. Za njega je $u = [0..2] = aca$. U tom slučaju je $P_{[2..3]}(aca) = \{0,4\}$.

pozicija	0	1	2	3	4	5	6	7	8	9	10
string $P_{[2..3]}$	*a*	*c*	*a*	*a*	*a*	*c*	*a*	*t*	*a*	*t*	$

Algoritam izračunava skupove pozicija redosledom odozdo naviše. To znači da se svaki lcp-interval obrađuje tek nakon što su obrađeni svi njegovi sinovi.
Pretpostavimo da je $[i..j]$ jednočlani interval, tj. $i = j$. Neka je $p = suftab[i]$.
Tada $P_{[i..j]} = \{p\}$ i

$$P_{[i..j]}(a) = \begin{cases} \{p\} \text{ ako } p > 0 \text{ i } S[p-1] = a \text{ ili } p = 0 \text{ i } a = \bot. \\ \varnothing, \text{ inače.} \end{cases}$$

Primer 5: Uzmimo za primer jedan od l-intervala sa slike 5.1: $P_{[0..5]}(\bot) = \{0\}$, $P_{[0..5]}(\mathrm{c}) = \{2,6\}$, $P_{[0..5]}(\mathrm{a}) = \{3,4\}$, $P_{[0..5]}(\mathrm{t}) = \{8\}$.
Navedene vrednosti se dobijaju za string *acaaacatat*$ i prikazane su u tabeli niže:

pozicija	0	2	3	4	6	8
znak	⊥	*c*	*a*	*a*	*c*	*t*

Neka je sada $i < j$. Za svako $a \in \Sigma \cup \{\bot\}$, $P_{[i..j]}(a)$ se izračunava korak po korak dok se obrađuju sinovi intervala $[i..j]$. Oni se obrađuju sleva na desno. Pretpostavlja se da su oni numerisani i da je već obrađeno q sinova intervala $[i..j]$. Neka $P^q_{[i..j]}(a)$ označava podskup $P_{[i..j]}(a)$ dobijen nakon obrade q-tog sina intervala $[i..j]$. Neka je $[i'..j']$ *(q+1)*-vi sin intervala $[i..j]$. Zbog prolaska odozdo naviše, $[i'..j']$ je obrađen i zato su pozicije skupova $P_{[i'..j']}(b)$ raspoložive za svako $b \in \Sigma \cup \{\bot\}$.
$[i'..j']$ je obrađen na sledeći način: Prvo, maksimalno ponavljanje je postavljano na izlaz kombinacijom skupa pozicija $P^q_{[i..j]}(a)$, $a \in \Sigma \cup \{\bot\}$ sa skupovima pozicija $P_{[i'..j']}(b), b \in \Sigma \cup \{\bot\}$. Pogotovu je, $((p,\ p+l-1),(p',\ p'+l-1))$ maksimalno ponavljanje. Po konstrukciji, različite su samo one pozicije p i p' koje su kombinovane za koje su znaci neposredno levo na primer na pozicijama p i $p'-1$ (ako one postoje). Ovo garantuje levu-maksimalnost izlaznih ponavljajućih parova.
Pozicije skupova $P^q_{[i..j]}(a)$ su bile nasleđene od sinova intervala od $[i..j]$ koji su različiti od $[i'..j']$. Shodno tome znaci neposredno na desno od u na pozicijama $p+l$ i $p'+l'$ (ako postoje) su različite. Kao posledica izlazno ponavljanje je maksimalno.

Jednom kada su maksimalna ponavljanja za trenutni sin interval $[i'..j']$ postavljena na izlaz, izračuna se unija $P_{[i..j]}^{q+1}(e) := P_{[i..j]}^{q}(e) \cup P_{[i'..j']}^{q}(e)$ za sve $e \in \Sigma U\{\perp\}$. To znači da je pozicija skupova nasleđena od $[i'..j']$ do $[i..j]$.
U algoritmu 3 ako se funkcija *obradi* primeni na lcp-interval, tada ona pronalazi svi intervale sinove. Stoga algoritam za maksimalno ponavljanje može da se implementira obilaskom odozdo naviše lcp-intervalnog stabla. Tako funkcija *obradi* u algoritmu 3 daje na izlazu maksimalna ponavljanja, i dalje održava pozicije skupova na steku (koje su dodate četvorkama kao peta komponenta). Obilazak odozdo naviše zahteva vreme $O(n)$.
Algoritam 3 pristupa *lcp*-tabeli sekvencijalno. Pored toga, algoritam za maksimalno ponavljanje to radi pomoću tabele $suftab$. Razmotrimo sada pristup ulaznom stringu S: kadgod je $suftab[i]$ obrađen, ulaznom znaku $S[suftab[i]-1]$ je pristupljeno. S obzirom na to da $bwtab[i] = S[suftab[i]-1]$, kadgod je $suftab[i] > 0$, pristup S-u može da se zameni pristupom $bwtab$-u. Drugim rečima, ulazni string nije potreban kada se izračuna maksimalno ponavljanje. Umesto toga, koristi se tabela $bwtab$ bez uvećavanja zahteva za ukupnim prostorom. Ista tehnika se primenjuje kada se izračunavaju supermaksimalna ponavljanja.
Za vreme obrade *lcp*-intervala $[i..j]$ izvode se dve operacije. Kombinovanje pozicija skupova, čime se dobijaju maksimalna ponavljanja i unija pozicija skupova. Svako kombinovanje pozicija skupova znači izračunavanje njihovog Dekartovog proizvoda. Ovo daje listu parova pozicija tj. maksimalna ponavljanja. Svako ponavljanje se izračunava za konstantno vreme polazeći od liste pozicija. Sve zajedno, kombinacije se izračunavaju za vreme $O(z)$, gde je z broj ponavljanja. Operacija nalaženja unije za pozicije skupova može da se implementira za konstantno vreme, korišćenjem povezanih lista. Za svaki *lcp*-interval, imamo $O(k)$ operacija, gde $k = |\Sigma|$. S obzirom na to da treba da se obradi $O(n)$ lcp-intervala, unija i dodatne operacije zahtevaju vreme $O(kn)$. Ukupno vreme izvršavanja algoritma je $O(kn + z)$.
Prelazimo na analizu prostorne složenosti algoritma. Skup pozicija $P_{[i..j]}(a)$ je unija skupova pozicija sinova intervala $[i..j]$. Kad se obradi sin intervala $[i..j]$, odgovarajući skup pozicija postaje nepotreban. Zbog toga nije potrebno da se kopiraju skupovi pozicija. Šta više, potrebno je samo uskladištiti skupove pozicija na steku, koje se koriste za obilazak odozdo naviše lcp-intervalnog stabla. Zato je prirodno uskladištiti reference na pozicije skupova na steku zajedno sa drugim informacijama o *lcp*-intervalu. Stoga je veličina prostora potrebnog za skupove pozicija ograničena maksimalnom veličinom steka. S obzirom na to da je maksimalna veličina steka $O(n)$, potreban je prostor $O(|\Sigma|n)$. U praksi je, međutim, veličina steka mnogo manja. Dakle, algoritam ima linearnu prostornu i vremensku složenost.

6. Zaključak i pravci daljeg rada

U ovom radu su izloženi neki od algoritama za konstrukciju sufiksnih stabala i sufiksnih nizova. Prvi linearni algoritmi za konstrukciju sufiksnog niza (direktno, a ne polazeći od sufiksnog stabla) pojavili su se tek 2003. godine. Pojava ovih algoritama je podstakla upotrebu sufiksnih nizova pre svega kao jednostavnije strukture, ali isto tako i zbog manjih zahteva za memorijom. Zahvaljujući efikasnosti ovih algoritama u današnje vreme za pretraživanje relativno velikih genoma dovoljan je personalni računar.

Priloženi programi SufiksnoStablo i SufiksniNiz mogu se iskoristiti za analizu genoma čemu su prevashodno i namenjeni. Međutim pored toga mogu da imaju i mnoge druge namene na svim poljima gde postoji neka vrsta pretraživanja stringova. Mogući pravac daljeg rada je usavršavanje ovih programa i traženje i realizacija njihovih novih primena.

S druge strane s obzirom na to da se ovaj rad bavio analizom algoritama za konstrukciju sufiksnih stabala i nizova isključivo sa stanovišta korišćenja personalnih računara, njihov dalji logičan pravac razvoja bio bi istraživanje naprednih modela izračunavanja i korišćenje paralelnih računara za njihovu realizaciju.

7. Literatura

[1] D. Gusfield, *Algorithms on Strings, Trees and Sequences,* Computer science and Computational Biology, Cambridge University press, New York, 1997.

[2] J. Kärkkäinen, P. Sanders, S. Burkhardt, *Linear work suffix array construction,* JACM, 53, 918-936, Springer, Berlin, 2003.

[3] M. I. Abouelhoda, S. Kurtz, E. Ohlebusch, *The enhanced suffix array and its applications to genome analysis,* In Proc. 2nd Workshop on Algorithms rn Bioinformatics, vol. 2452 serije *LNCS,* 449-463, Springer, Berlin, 2002.

[4] P. Bieganski. *Genetic sequence data retrieval and manipulation based on generalized suffix trees,* Ph.D thesis, University of Minnesota, Dept. Computer Science, Minneapolis, 1995.

[5] A. Apostolico, *The myriad virtues of subword trees,* U knjizi A. Apostolico, Z. Galil, ed., Combinatorics of words, Nato ASI series vol. 112, 85-96, Springer, Montreal, 1985.

[6] J. Kärkkäinen, P. Sanders, *Simple linear work suffix array construction,* In Proc. 30th International Conference on Automata, Languages and Programming, vol. 2719 serije LNCS, 943-955, Springer, Berlin, 2003.

[7] M. Farach, *Optimal suffix tree construction with large alphabets*, In Proc. 38th Annual Symposium on Foundations of Computer Science, 137-143, IEEE, 1997.

[8] C. J. Colbourn, A. C. H. Ling, *Quorums from difference covers*, Information Processing Letters, 75(1-2), 9-12, Amsterdam, 2000.

[9] T. Kasai, G. Lee, H. Arimura, S. Arikawa, K. Park, *Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and its Applications*, In Proceedings of the 12th annual Symposium on Combinatorial Pattern Matching, 596-604, IEEE Computer Society, New York, 1999.

[10] M. Burrows i D.J. Wheeler, *A Block-Sorting Lossless Data Compresion Algorithm. Research Report 124*, Digital Systems Research Center, 1994.

[11] S. Kurtz, J.V. Choudhuri, E. Ohlebush, C. Shleiermacher, J. Stoye, R. Giegerich. *REPuter: The Manifold Applications of Repeat Analysis on a Genomic scale.* Nucleic Acids Res., 29(22), 4633-4642, Oxford University Press, 2001.

[12] A.L. Delcher, S. Kasif, R.D. Fleischmann, J Peterson, O. White, S.L. Salzberg, *Alignment of Whole Genomes*, Nucleic Acids Res., 27, 2369-2376, Oxford University Press, 1999.