Univerzitet u Beogradu

Elektrotehnički fakultet



Primena algoritamama pretrazivanja znakovnih sekvenci u analizi DNK

Seminarski rad

|  |  |
| --- | --- |
| Mentor: | Kandidat: |
| Dr Branko Marović | Nemanja Joksić 3140/2017 |

Beograd, Septembar 2019.

Sadržaj

[Sadržaj i](#_Toc18258775)

[1. Uvod 1](#_Toc18258776)

[2. DNK, gen i genom 2](#_Toc18258777)

[3. Računarska analiza genoma 3](#_Toc18258778)

[3.1. Upotreba računarskih algoritama u pronalasku genetskih oboljenja 3](#_Toc18258779)

[3.2. *Exact string matching* algoritmi 3](#_Toc18258780)

[3.3. Dinamički algoritmi pretraživanja 4](#_Toc18258781)

[3.3.1. Naïve algoritam 4](#_Toc18258782)

[3.3.2. Knuth-Morris-Pratt algoritam 5](#_Toc18258783)

[3.3.3. Boyer-Moore algoritam 6](#_Toc18258784)

[3.3.4. Poređenje Naïve, Knuth-Morris-Pratt i Boyer-Moore algoritma 7](#_Toc18258785)

[3.4. Algoritmi koji vrše pretprocesiranje ulaznog niza 8](#_Toc18258786)

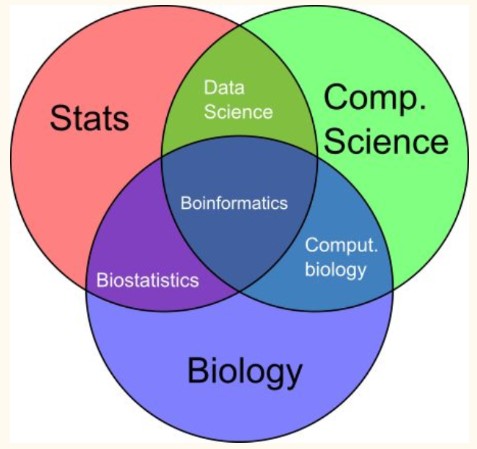
[3.4.1. Sufiks stabla 8](#_Toc18258787)

[3.4.2. Sufiks niz 10](#_Toc18258788)

1. Uvod

Sa otkrićem genoma i razvojem molekularne biologije i genetike pojavila se potreba za razvojem tehnika koje bi mogle da na efikasan način procesiraju i analiziraju te podatke. Istovremeno se razvijalo i računarstvo pa su se računari nametnuli kao prirodan alat za procesirawe velikih količina podataka. Tako je došlo do razvoja jedne nove nauke, bioinformatike.

Bioinformatika je nauka o informacijama i protoku informacija u biološkim sistemima, sa posebnim akcentom na upotrebu računarskih metoda u genetici i genomici (Oxford english dictionary).



Slika 1. Veza statistike, biologije i računarstva

Na slici 1. se može videti veza biologije, statistike i računarstva i to da se bioinformatika nalazi u samom preseku ove tri nauke. Praktično bioinformatika istražuje i razvija metode, softverske algoritme i alate za skladištenje, organizovanje i analiziranje bioloških podataka.

U radu će prvo biti objašnjeno šta su DNK, gen i genom da bi mogli da razumemo njihov značaj. Zatim će biti predstavljen značaj procesiranja i analize genoma za razuvemanje njegovog funkcionisanja i otrivanje nepravilnosti u njegovoj strukturi koje mogu biti uzroci različitih bolesti. Da bi ovakav tip analize radio na računaru, genom se prvo mora predstaviti u nekom obliku pogodnom za računarsku obradu. Biće predstavljen jednostavan način reprezentacije genoma u računarstvu i primene softverskih algoritama za njegovo procesiranje. Naredni deo rada biće posvećen primeni različitih algoritama pretraživanja u analizi DNK, odnosno genoma.

1. DNK, gen i genom

DNK je nukleinska kiselina koja sadrži uputstvo za razvoj i pravilno funkcionisanje svih živih organizama. On predstavlja nasledni materijal u svim živim ćelijama. Sastoji se od para molekula međusobno povezanih vodoničnim vezama. Svaki lanac je izgrađen od nukleotida kojih ima četiri vrste: adenin (A), citozin (C), guanin (G) i timin (T).

Celovit deo DNK koji prenosi naslednu informaciju sa generacije na generaciju naziva se gen. Građa gena je u stvari građa DNK i ogleda se u tačno određenom redosledu četiri prethodno pomenuta nukleotida. Promena tog redosleda, manjak ili višan nukleotida rezultira u promeni funkcije gena i naziva se genetska mutacija.

Genom je skup svih gena nekog organizma i sadrži sve informacije potrebne za pravilan rad i razvoj organizma. Naučnici decenijama istražuju i analiziraju genom u svrhu nalaženja novih lekova, pronalaska, razumevanja i lečenja genetskih mutacija, kao i u svrhu razumevanja funkcionisanja samih organizama. Zbog ovoga je razvoj bioinformatike, odnosno razvoj metoda i alata koji omogućavaju efikasno i brzo procesiranje genoma od velikog značaja.

1. Računarska analiza genoma

Pošto se DNK sastoji od četiri nukleinske kiseline kao što je objašnjeno u prethodnom poglavlju sam genom se može predstaviti kao niz slova iz skupa {A,C,G,T}. Ovakav format reprezentacije DNK je sada pogodan za računarsku obradu. Samim tim mnogi aspekti analize genoma se mogu svesti na problem pretraživanja i analize znakovnih nizova.

* 1. Upotreba računarskih algoritama u pronalasku genetskih oboljenja

Jedan od najznačajnihih primena bioinformatike u genetici je detekcija pojedinih sekvenci u DNK. Proverava se postojanje različitih genetskih bolesti tako što se odgovarajuće sekvence nukleotima pretražuju u strukturi DNK. Kako su i genomi i same sekvence ogromne potrebno je osmisliti odgovarajuće računarske algoritme koji mogu ovu analizu da rade dovoljno efikasno. U slučajevima kada znamo da je neka sekvenca nukeotida uzrok genetske bolesti, mesta na kojima se ona pojavljuje i broj pojavljivanja mogu dati informacije o inteziteti i težini bolesti.

Zbog toga je od posebnog značaja operacija provere da li se neka sekvenca nukleotida nalazi u DNK i na kojim mestima se ona nalazi. Kako se i genomi i sekvence odgovarajućih bolesti mogu predstaviti sekvencom znakova iz skupa {A,C,G,T} ovaj problem se može preslikati u problem provere da li se jedan sekvenca znakova sadrži u drugoj i ako se sadrži na kojim mestima se on pojavljuje. Računarski algoritami koji se bave ovim problemom nazivaju se *exact string matching* algoritmi.

* 1. *Exact string matching* algoritmi

*Exact string matching* algoritmi se mogu u osnovi podeliti u dve grupe. U prvu grupu spadaju algoritmi koji ne zahtevaju predznanje o samoj sekvenci koji se pretražuju, odnosno neke dodatne informacije o postojećim sekvencama koji se mogu pojaviti u njemu i možemo ih nazvati i dinamički algoritmi pretraživanja. Neki od ovih algoritama vrše pretproceisranje sekvence koji se pokušava naći sa ciljem da se kreira struktura podataka na osnovu koje se onda dinamički u toku same pretrage mogu preskakati nepotrebna poređenja i time skratiti vreme pretrage. Drugu grupu čine algoritmi koji prvo vrše pretprocesiranje same sekvence koji se pretražuje i zatim se dalja pretraga i analiza vrši nad strukturama podataka kreiranih u ovoj fazi, a ne nad samim originalnom sekvencom. Prva grupa algoritama zahteva manje hardverskih resursa za samu pretragu, dok druga grupa algoritama vrši iste pretrage za manje vremena.

U nastavku će biti predstavljeni tri dinamička algoritma pretraživanja. Prvi od njih je *Naïve* algoritam koji predstavlja najjednostavniju implementaciju *еxact string matching* algoritma i koji kao takav nije primenljiv za iole većem setu podataka. Zatim će biti detaljno objašnjena dva algoritma koji da rade i sa većim setovima podataka, a to su *Knuth-Morris-Pratt* i *Boyer-Moore* algoritam. Ova dva algoritma vrše pretprocesiranje sekvenci koji se traže samo sa različitim ciljem.

U poglevlju nakon ovoga biće predstavljeni neki od algoritama koji pretprocesiraju sekvence koje se pretražuju, a zatim analizu i pretragu rade nad podacima dobijenim u ovoj fazi. Prvo će biti predstavljeni algoritmi koji koriste strukture podataka zasnovane na n-arnim stablima. Ovi algoritmi su memorijski jako zahtevni pa će u nastavku biti predstavljeni sufiks nizovi kao jedno od potencijalnih rešenja ovog problema. Na kraju će biti detaljno objašnjen *Burrows-Wheeeler* algoritam i kako se on koristi u kombinaciji sa *FM indeksom* za brzo i efikasno pretraživanje ogromnih znakovnih nizova sa dosta manje operativne memorije.

* 1. Dinamički algoritmi pretraživanja

Za početak treba uvesti odgovarajuću terminologiju koja će biti korišćena u ostatku rada.

T – sekvenca koji se pretražuje

P – sekvenca koji se traži

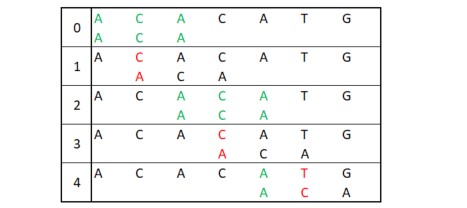
N – broj znakova u T

M – broj znakova u P

Obe sekvence se sastoje od znakova iz skupa {A,C,G,T} jer kao što smo već rekli, genom se može reprezenzovati nizom sastavljenim od ova četiri znaka.

* + 1. Naïve algoritam

Ovo je najjednsotavniji algoritam pretraživanja znakovnih nizova. Ideja ovog algoritma je da se pokuša naći P u T na svakom mogućem pomeraju. Tako se kreće od početnog pomeraja u T i proverava se da li P postoji u T zaključno sa pomerajem N – M. Pretraga za bilo koji pomeraj se prekida ukoliko se odgovarajući znakovi ne poklapaju.



Slika 2. Primer izvršavanja *Naïve* algoritma za T = ACACATG i P = ACA

Na slici 2. je prikazan rad ovog algoritma po iteracijama. Svaka kolona predstavlja pretragu za odgovarajući pomeraj. Slova obojena selenom bojom su situacije kada je došlo do poklapanja odgovarajućih znakova iz T i P, a crvena boja predstavlja situaciju kada se odgovarajući znakovi ne poklapaju i kada se pretraga na tom pomeraju prekida i prelazi se na naredni pomeraj. Lako se može izračunati da je u ovom konkretnom primeru bilo 10 poređenja i da su nađena poklapanja na pomerajima 0 i 2.

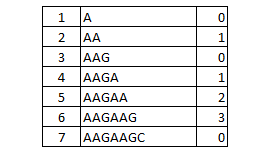
Neefikasnost je zapravo i glavni nedostatak ovog algoritma i to se moglo videti i u ovom jednostavnom primeru gde postoji relativno veliki broj poređenja i za ovako male nizove. Najveći broj poređenja će biti potreban u slučaju da se P sadrži u T na svakom pomeraju i od iznos*(N – M + 1)xM*, a najmanje u slučaju da se P[0] ne nalazi u T i tom slučaju je potrebno *N – M + 1* poređenja. Kako je u slučaju genoma N jako veliko i obično mnogo veće od M, *(N – M + 1) ≈ N*, odnosno *(N – M + 1)xM ≈ NxM*. Na osnovu ovoga možemo reći da je složenost ovog algoritma u najboljem slučaju *O(N),* a u najgorem *O(NxM).*

* + 1. Knuth-Morris-Pratt algoritam

Najveća mana *Naïve* algoritma je u tome što on vrši dosta nepotrebnih poređenja koja mogu biti izbegnuta. Osnovni motiv pri razvoju prvih naprednijih algoritama pretraživanja znakovnih nizova je bilo upravo ovo, da se preskoče sva ili makar neka nepotrebna poređenja i time smanji vreme pretrage. Informacija kada i koja poređenja treba preskočiti može se dobiti iz samog niza koji se traži. Prvi ovakav algoritam osmislili su *Donald Knuth* i *Vaughan Pratt*, a istovremeno samostalno ga je osmislio i *James Morris* 1975. godine. Po ovim naučnicima ovaj algoritam dobija ime *Knuth-Morris-Pratt* algoritam.

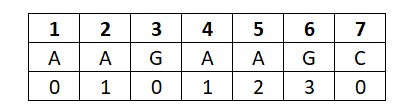
Osnovna ideja ovog algoritma je da iskoristi činjenicu da su neki znakovi već poznati iz prethodnih iteracija algoritma i da je na osnovu tog znanja moguće preskočiti neka poređenja za koja već znamo da će biti uspešna. Stoga se ovaj algoritam sastoji iz dve faze. Prva je pretprocesiranje niza P kako bi se kreirao jedan pomoćni niz u kom se nalaze pomeraji koje treba preskočiti u toku same pretrage, a druga faza je sama pretraga.

Cilj faze pretprocesiranja niza P je da se kreira pomoćni niz koji sadrži poziciju na koju se treba vratiti u P pošto se desi nepoklapanje odgovarajućih znakova iz T i P. Pretprocesiranje se sastoji iz prolaska kroz sve podnizove niza P koji počinju od pozicije 0 u P. To su dakle nizovi *Р[0:0], P[0:1], P[0:2],..., P[0:M-1]*. U svakom podnizu se traži najduži prefiks koji je ujedno i sufiks u tom podnizu, stim da je ceo podniz ujedno i prefiks i sufiks samog sebe pa se taj slučaj preskače. Zatim se dužina odgovarajućeg najdužeg prefiksa koji je ujedno i sufiks upisuje na poziciju u rezultujućem nizu na kom se odgovarajući podniz završava u P.



Slika 3. Proces pretprocesiranja niza AAGAAGC

Na slici 3. je prikazano pretprocesiranje niza AAGAAGC. Prva kolona prestavlja redni broj iteracije, naredna odgovarajući podniz koji se obrađuje u toj iteraciji, a poslednja kolona je dužina prefiksa koji je ujedno i sufiks, vodeći se pravilom opisanim u prethodnom pasusu. Iz ovoga se lako može dobiti odgovarajući podniz koji će se koristiti u daljem procesu pretraživanja, a to je *lps = [0, 1, 0, 1, 2, 3, 0].*



Slika 4. Struktura *lps* niza za primer sa slike 3.

Na slici 4. prikaza je tabela kojom se treba voditi u toku pretraživanja. Prvi red predstavlja tekuću poziciju u P, a poslednja kolona redni broj znaka u P na koji se treba vratiti ukoliko se desi nepoklapanje odgovarajućih znakova iz T i P. Pravilo je dosta jednostavno i kaze:

1. Ako se nepoklapanje desilo u slučaju poređenja prvog znaka iz P, pomeriti se u T za jedno mesto.
2. U suprotnom pomeriti se na poziciju *lps[j-1],* gde je *j* tekuća pozicija u P.

U odnosu na *Naïve* algoritam *Knuth-Morris-Pratt* algoritam postiže poboljšanje u slučaju najgoreg scenarija, a to je da se P u T sadrži na svakom pomeraju i složenost u ovom slučaju je *O(N+M).* Što dosta bolje od *O(NxM)* kod *Naïve* algoritma.

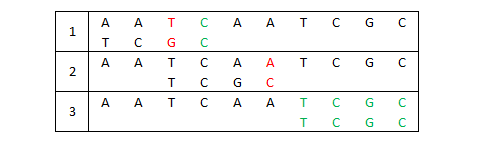
* + 1. Boyer-Moore algoritam

1977. godine *Robert Boyer* i *J Strother Moore* su osmislili jedan efikasniji algoritam pretraživanja koji je po njima i dobio ime *Boyer-Moore* algoritam. Kao i *Knuth-Morris-Pratt* algoritam i ovaj algoritam vrši pretprocesiranje sekvence P sa ciljen da u fazi pretraživanja može da u odgovarajućim trenucima preskoči neka poređenja. Osnovna karakteristika ovog algoritma je da poređenje vrši u obrnutom poretku, odnosno poredi odgovarajuće znakove počevši od poslednjeg znaka iz P, za razliku od *Knuth-Morris-Pratt* algoritma koji poređenje vrši počevši od prvog znaka u P. Upravo ovo svojstvo u opštem slučaju zahteva daleko manji broj poređenja nego kod prethodna dva algoritma i omogućava preskakanje nekih nepotrebnih poređenja koja bi kod *Knuth-Morris-Pratt* algoritma bila izvršena.

I ovaj algoritam se izvršava u dve faze. Prvi deo je faza pretprocesiranja sekvence P kako bi se u ovom slučaju izgenerisala odgovarajuća tabela (koja zavisi od pravila koje se koristi, a o kojima će uskoro biti više reči) i na osnovu koje se računaju pomeraji koje treba preskočiti u toku same pretrage. Druga faza algoritma je sama pretraga.

Algoritam se vodi sa dva osnovna pravila pri pretraživanju. Prvo pravilo se naziva pravilo lošeg znaka, a drugo pravilo dobrog sufiksa. Drugo pravilo se može pojaviti sa još jednom dodatnom optimizacijom o kojoj će kasnije biti reči. Oba pravila se mogu primenjivati nezavisno za pretraživanje i prvo će tako biti i predstavljena. Međutim oba pravila se mogu i kombinovati kako bi se ovaj algoritam učinio još efikasnijim.

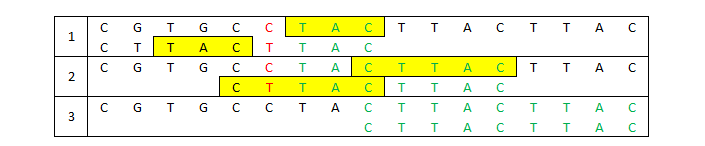
Pravilo lošeg znaka je jako jednostavno i primenjuje se u situaciji kada se desi nepoklapanje znakova iz T i P. Neka je *t* odgovarajući znak iz T i neka je *p* odgovarajući znak iz P i neku su oni različiti. Tada treba P pomeriti u desno za onoliko mesta koliko ima do najbližeg znaka u P koji je jedank *t* ako takav postoji, a ako ne postoji, onda P pomeriti do prvog znaka iz *t*.



Slika 5. Primena *Boyer-Moore* algoritma po iteracijama korišćenjem pravila lošeg znaka

Na slici 5. demonstriran je *Boyer-Moore* algoritam primenom pravila lošeg znaka. U prvoj iteraciji dolazi do nepoklapanja znakova T i G, pa se iz tabele dobija da P treba pomeriti za dva mesta u desno. U drugoj iteraciji dolazi do nepoklapanja znakova A i C, a kako P ne sadrži znak A, P se pomera za celu svoju dužinu u desno. Konačno u trećoj iteraciji dolazi do poklapanja svih znakova.

Drugo nešto složenije pravilo je pravilo dobrog sufiksa. Neka je t podniz niza T koji se poklapa sa sufiksom niza P. Pomeriti P u desno za onoliko mesta koliko ima u prvom narednom podnizu t u P. Ako takvog nema pomeriti P u desno za onoliko mesta koliko ima do najdužeg prefiksa iz P koji je isti kao odgovarajući sifiks iz t. Ako ni to ne postoji pomeriti P u desno za onoliko mesta kolika mu je dužina.



Slika 6. Primer *Boyer-Moore* algoritma po iteracijama korišćenjem pravila dobrog sufiksa

Na slici 6. demonstriran je *Boyer-Moore* algoritam primenom pravila dobrog sufiksa. U prvoj iteraciji dolazi do nepoklapanja znakova C i T, pa se P pomera za četiri mesta u desno koliko je rastojanje do narednog podniza TAC u P. U drugoj iteraciji dolazi do nepoklapanja znakova C i T koliko je rastojanje do prefiksa u P koji je ujedno i sufiks u T, CTTAC. Konačno u trećoj iteraciji dolazi do poklapanja svih znakova.

Ova dva pravila se mogu koristiti i zajedno na jako jednostavan način. Svaki put treba proveriti koja od dva pravila će dati veći pomeraj i primeniti ono koje daje veći.

Složenost algoritma u slučaju kada je potrebno izvršiti najviše poređenja je ista kao i kod *Naïve* algoritma i iznosi *O(NxM).* Međutim u najboljem slučaju, a to je situacija kada se poslednji znak iz P ne sadrži u T, složenost je *O(N/M).*

* + 1. Poređenje Naïve, Knuth-Morris-Pratt i Boyer-Moore algoritma

Zajednička osobina sva tri algoritma kao što je već rešeno je da ne vrše nikakvo procesiranje ulaznog niza. Zbog toga ovi algoritmi ne zahtevaju značajne dodatne memorijske resurse koji u slučaju analize DNK mogu biti jako velike. Razmatranje najpovoljnijeg slučaja kod svih algoritama a to je situacija da se i T i P sastoje od samo jednog znaka nije od interesa jer DNK nije takva struktura. Takođe i razmatranje najnepovoljnije situacija, a to je da se prvi znak iz P ne sadrži u T, takođe nije od interesa jer se DNK sastoji od samo četiri nukleinske kiseline i one se uvek sve pojavljuju u njemu.

*Naïve* algoritam ne vrši procesiranje sekvence koja se traži, pa je sa te strane on memorijski optimalan. Ipak to ga čini praktično neupotrebljivim u analizi velikih nizova kao što je DNK jer u opštem slučaju mnogo puta poredi iste znakove i stalno se vraća u nazad kroz nizove posle svake greške u poređenju.

*Knuth-Morris-Pratt* algoritam pretprocesira P i tada koristi dodatni niz iste dužine kao i P, sa ciljem da na osnovu podataka u tom nizu, tekuću poziciju u sekvenci koju traži ne vraća svaki put na početak sekvence, i da se kroz ulazni niz ne vraća nikada. Kao takav, ovaj algoritam je u opštem slučaju dosta efikasniji od prethodnog jer izbegava veliki broj nepotrebnih poređenja. Može biti efikasan u slučaju da u sekvenci koja se traži postoje duge podsekvence koje se ponavljaju, što može biti slučaj sa DNK sekvencama.

Još efikasniju implementaciju nudi *Boyer-Moore* algoritam koji pretragu na svakom pomeraju radi od kraja niza P. Istina pravilo lošeg znaka u slučaju analize genoma ili samog DNK nije od velike koristi jer su sekvence duge, a sastoje se od četiri znaka. Međutim pravilo dobrog sufiksa može omogučiti preskakanje mnogo većeg broja nepotrebnih poređenja. Ovo mu omogućava da potencijalno preskoči dosta poređenja u slučaju da se sekvence sa kraja P ne javljaju često u T. U slučaju da u uzorku DNK koji se analizira postoji dosta seknvenci koje imaju isti dugačak prefiks kao i sekvenca koja se traži, ali nisu iste, *Boyer-Moore* algoritam je daleko efikasniji od *Knuth-Morris-Pratt* algoritma, a ovo može često da bude slučaj.

Obzirom da svi algoritmi iz ove grupe ne vrše procesiranje ulazne sekvence, nije moguće dobiti bolje performanse u situacijama kada se vrši analiza stalno istog uzorka DNK na postojanje različitih sekvenci nukleotida u okviru njega, u odnosu na analize velikog broja različitih uzoraka. U gentici se često može pojaviti potreba za ovakvim tipom analize. Stoga će u nastavku biti izloženi algoritmi koji vrše protprocesiranje ulazne sekvence pre same pretrage i koji kao takvi pružaju mogućnost za optimizaciju ovakvog tipa analize.

* 1. Algoritmi koji vrše pretprocesiranje ulaznog niza

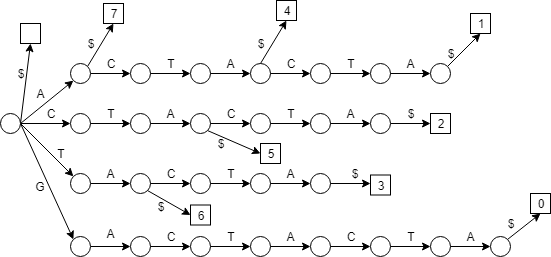
Postavilo se pitanje da li postoje efikasnije metode pretraživanja koje ne zahtevaju da se svaki put iznova prolazi kroz cele nizove. Prva ideja koja se nameće kao rešenje ovog problema je da se ne pretražuju sami nizovi, već da se prvo izvdoje svi mogući podnizovi originalnog niza koji , a zatim pretraži taj skup. Međutim ovakav pristup donosi i neke nove izazove koje je potrebno rešiti, kao što je izbor strukture podataka u koju se mogu smestiti svi podnizovi, a drugi je kako takvu strukturu podataka pretraživati efikasno.

Prvo će biti predstavljeni algoritmi koji koriste strukture podataka zasnovane za *n*-arnim stablima sa ciljem da zapamte sve moguće sekvence koje se pojavljuju u T. Ipak iako efikasne ove strukture su memorijski jako zahtevne i u slučaju analize DNK potrebni su gigabajti operativne memorije za smestanje ove strukture za samo jedan uzorak DNK. Stim u vezi biće predstavljeni sufiks nizovi koji su nastali sa ciljem da ostvare iste ili slične performanse kada je u pitanju samo pretraživanje, ali sa daleko manjih memorijskim zauzećem. Na kraju će biti detaljno objašnjen Burrows-Wheeeler algoritam i kako se on koristi u kombinaciji sa FM indeksom za brzo i efikasno pretraživanje ogromnih znakovnih nizova.

* + 1. Sufiks stabla

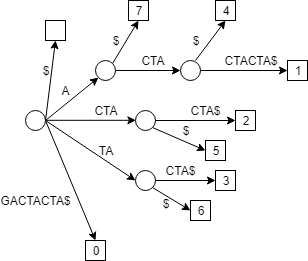
Algoritmi iz ove grupe u fazi pretprocesiranja ulaznu sekvencu transformišu u teki oblik *n*-arnog stabla, a zatim pretrage vrše nad tim stablom, a ne nad originalnom sekvencom. Ovaj koncept je prvi uveo *Peter Weiner* 1973. godine. Ideja je da se kreira stablo od svih mogućih sufiksa sekvence T. Prva i najjednostavnija implementacija ovog stabla se naziva sufiks *trie*. U ovom stablu svaka grana je obeležena jednim znakom iz skupa {A, C, G, T}. U listovima se nalaze pomeraji na kojima odgovarajući sufiks počinje u originalnoj sekvenci. Svaka putanja od korena stabla do lista predstavlja jedan sufiks sekvence T. Jako je bitno reći da se sekvenca T prvo treba proširiti za jedan specijalan znak koji će imati vrednost manju od bilo kog drugo znaka iz skupa. U ovom radu će biti u tu svrhu korišćen nak $.

Izgled ovakvog stabla za T = GACTACTA dat je na slici 7. Može se lako prebrojati da na slici ima 9 listova, što odgovara broju sufiksa u T plus 1. Jedan dodatni sufiks postoji jer je T proširen znakom $.



Slika 7. Sufiks trie za T = GACTACTA

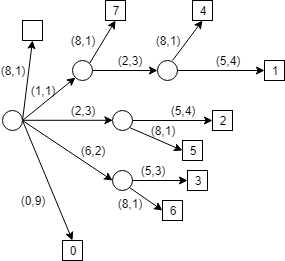
Pretraga pojavljivanja sekvence P u većoj sekvenci T se vršti tako što se koristeći jedan po jedan znak iz P prolazi kroz stablo vodeći se granama koje odgovaraju tekućem znaku. Ako se desi da odgovarajuća grana ne postoji, to znači da se P ne sadrži u T, u suprotnom ako se uspešno iskoriste svi znakovi iz P to znači da se P sadrži u T. Broj pojavljivanja i mesta na kojima se P pojavljuje u T dobija se analizom podstabla iza čvora kome je pretraga zaustavljena. Broj listova jednak je broju pojavljivanja, a iz samih listova mogu se dobiti odgovarajući pomeraji na kojima se tražena sekvenca pojavljuje. Složenost procesa pretraživanja je *O(M),* stim da tu treba i dodatno uračunati vreme potrebno za analizu podstabla ukoliko ju je potrebno izvršiti.



Slika 8. Kompaktni sufiks trie za T = GACTACTA

Memorijski ovaj algoritam je jako zahtevan. U stablu može postojati dosta čvorova koji imaju samo jednu izlaznu granu. Ti čvorovi i grane su nepotrebni. Daleko kompaktnija struktura se može kreirati tako što se sve sukcesivne grane koje imaju samo jednu izlaznu granu spoje u jednu, a odgovarajućoj labeli dodeli nova vrednost koja je jednaka skupu vrednosti labela koje su izblačene. Takođe sve čvorove na toj putanji treba izbaciti iz stabla. Na slici 8. prikazano je stablo sa slike 7. u komprimovanom obliku.

Dodatno povoljšanje je moguće postići tako što se labele na granama zamene parovima (pomeraj, dužina), gde pomeraj predstavlja početnu poziciju u T na koju ta grana ukazuje, a dužina broj znakova koji su obuhvaćeni.

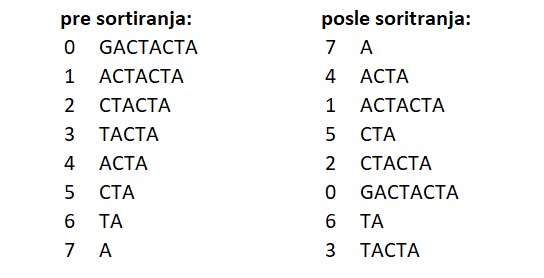


Slika 9. Sufiks stablo za T = GACTACTA

Na slici 9. prikazano je sufiks stablo sa poslednjom opisanom optimizacijom. Složenost same pretrage je u svim slučajevima *O(M)*, na šta treba dodati i složenost analize odgovarajućeg podstabla ukoliko je ta analiza potrebna, odnosno ako se traži broj ili pozicije na kojima se P javlja u T. Međutim u prvom slučaju je potrebno obići daleko više čvorova, a kako i sam obilazak čvora i provera kuda dalje treba nastaviti pretragu traje neko vreme, jasno je da i ovako jednostavna optimizacija može značajno ubrzati pretragu. Da su performanse ovog algoritma značajno bolje nego performanse bilo kog iz grupe dinamičkih može se lako potvrditi uvidom u njihove složenosti. Složenost algoritama zasnovanih na sufiks stablima zavisi isklučivo od dužine sekvence koja se traži, dok složenost dinamičkih algoritama prvenstveno zavise od dužine sekvence koja se pretražuje. Kada je u pitanju DNK razlika u performansama ovih algoritama je jako velika jer je veličina DNK često daleko veća od veličine sekvenci koje se traže. Problem međutim leži u tome što faza pretprocesiranja može da dugo traje, a samo stablo i pored različitih optimizacija zahteva mnogo memorije što predstavlja ključni nedostatka ovih algoritama. U nastavku će biti predstavljena dve strukture podataka koje pokušavaju da ograniče memorijske zahteve, a da zadrže brzinu same pretrage.

* + 1. Sufiks niz

Kao što pokazano, strukture zasnovane na stablima su efikasne, ali zahtevaju puno memorije. Sa ciljem da prevaziđu ovaj problem, a zadrže efikasnost same pretrage, *Udi Manber* i *Gene Myers* su 1990. godine osmislili novu strukturu podataka koju su nazvali sufiks niz. Sufiks niz je niz celih brojeva gde svaki broj predstavlja poziciju na kojoj odgovarajući sufiks počinje u originalnom nizu. Takođe sufiksi u nizu su soritrani. Na slici 10. dat je izgled sufiks niza za tekst GACTACTA pre i posle sortiranja.



Slika 10. Sufiks niz za T = GACTACTA

Sufiks niz za primer teksta GACTACTA je [7, 4, 1, 5, 2, 0, 6, 3]. Sam niz se može konstruisati na više načina, sa različitom efikasnošću i složenošću. Što je pristup jednostavniji performance su lošije. Najjednostavniji pristup je da se prvo kreira nesortiran niz svih mogućih sufiksa, a zatim koristeći neki od efikasnijih algoritama soritranja kao što su *quicksort* ili *mergesort* niz sortira. Složenost samog algoritma soritranja je , a poređenja sufiksa pa je složenost ovakvog načina kreiranja sufiks niza , gde je *n* dužina niza za koji se kreira sufiks niz. Sufiks niz može biti kreiran i sa složenošću *O(N)*, koristeći se sufiks stablom i obilaskom stabla po dubini, međutim različiti algoritmi konstrukcije sufiks nizova kao ni sufiks stabala nisu od interesa u ovom radu i neće dalje biti razmatrani. Od interesa je primena ovih struktura podataka u analizi DNK sekvenci.

Provera da li se P pojavljuje u T i na kojim mestima, može se jednostavno uraditi sa dve binarne pretrage zahvaljujući činjenici da su sufiksi soritrani po leksikografskom poretku. U prvoj pretraži se traži najlevlji sufiks čiji je prefiks tražena sekvenca, a zatim se u drugoj pretrazi traži sufiks čiji je prefiks tražena sekvenca. Sve vrednosti u nizu između njih (ukljucujući i njih) su mesta na kojima se tražena sekvenca pojavljuje u T. Složenost same binarne pretrage je *O(*log *N)*, a porovere na jednakost sufiksa i sekvence *O(M),* pa to ukupno daje *O(M* log *N).* U primeru sa slike 10. ukoliko bi se tražila sekvenca ACT, rezultat prve pretrage bi bili indeks 1, druge indeks 2, pa bi se tražene pozicije nalazile u opsegu *sufiks\_niz[1:2]*. To su pozicije 4 i 1.