Univerzitet u Beogradu

Elektrotehnički fakultet



Primena algoritamama pretrazivanja znakovnih nizova u analizi genoma

Seminarski rad

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

Beograd, Septembar 2019.

Sadržaj

[Sadržaj i](#_Toc17585674)

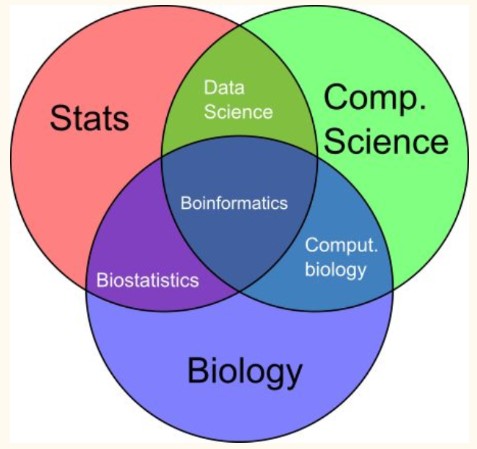
[1. Uvod 1](#_Toc17585675)

[. 2](#_Toc17585676)

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 i najveći deo rada biće posvećen primeni različitih algoritama pretraživanja u analizi 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 razbio bioinformatike, odnosno razvog 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.

Od posebnog značaja je operacija provere da li se neka sekvenca nukleotida nalazi u genomu i na kojim mestima se ona nalazi. Ovaj problem se može preslikati u problem provere da li je jedan znakovni niz podniz drugog niza i ako jeste 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 samom nizu koji se pretražuju, odnosno neke dodatne informacije o postojećim podnizovima koji se mogu pojaviti u njemu i možemo ih nazvati i dinamički algoritmi pretraživanja. Neki od ovih algoritama vrše pretproceisranje niza 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 samog niza koji se pretražuje i zatim se dalja pretraga i analiza vrši nad strukturama podataka kreiranih u ovoj fazi, a ne nad samim originalnim nizom. 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 znakovnih nizova. 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 ova algoritma vrše pretprocesiranje samih nizova koji se traže samo sa različitim ciljem.

U poglevlju nakon ovoga biće predstavljeni neki od algoritama koji pretprocesiraju same nizove koji se pretražuju, a zatim analizu i pretragu rade nad podacima dobijenim u ovoj fazi. Prvo će biti objašnjen algoritam koji koristi najjednostavniju, ujedno i najmanje efikasnu strukturu u koju se mogu smestiti svi mogući podnizovi niza koji se pretražuje, a to je sufiks niz. Zatim će biti predstavljeni algoritmi koji koriste nešto složenije strukture podataka sa istim ciljem, da zapamte sve moguće podnizove koji se pojavljuju u T, a to su sufiks *trie* i sufiks stabla. 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. Dinamički algoritmi pretraživanja

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

T – niz koji se pretražuje

P – niz koji se traži

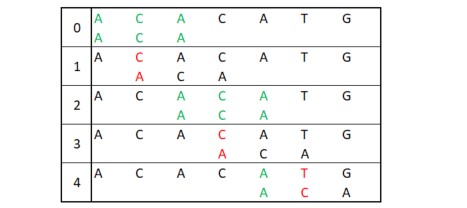
N – dužina niza T

M – dužina niza P

Oba niza se sastoje iz 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.

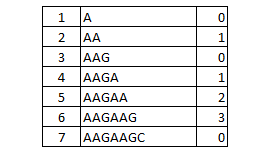
Neefikasnost je zapravo i glavni nedostatak ovog algoritma i to se moglo videti i u ovom jednostavnom primeru gde postoji relativno veliki broj poređenja i za ovako male nizove. Najveći broj poređenja će biti potreban u slučaju da se P sadrži u T na svakom pomeraju i od iznos*(N – M + 1)xM*, a najmanje 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 pozicije od kojih treba nastaviti pretraživanje pošto se desi nepoklapanje odgovarajućih znakova iz T i P. Pretprocesiranje se sastoji iz prolaska kroz sve podnizove kiza 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].*

Neka su *i* i *j* tekuće pozicije u T i P. Algoritam kreće od pomeraja 0. Sastoji se iz nekoliko koraka:

1. Ako se znakovi iz oba niza na odgovarajućim pozicijama podudaraju oba brojača se uvećavaju za 1.
2. U slučaju da se došlo do kraja niza P, brojač j se postavlja na vrednost *lps[j-1].*
3. U suprotnom ako se nije došlo do kraja niza P i nisu se poklopili odgovarajući znakovi iz T i P:

3a. Ako je *j=0*, i se uvećava za 1.

3b. U suprotnom *j=lps[j-1].*

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 niza 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.

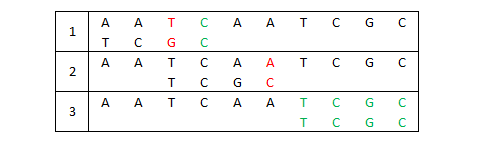
I ovaj algoritam se izvršava u dve faze. Prvi deo je faza pretprocesiranja niza P kako bi se u ovom slučaju izgenerisala odgovarajuća tabela (koja zavisi od pravila koje se koristi, a o kojima će uskoro biti više reči) i na osnovu koje se računaju pomeraji koje treba preskočiti u toku same pretrage.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

Cilja faze pretprocesiranja je kreirati niz R koji sadrži pozicije na kojima se odgovarajući znak pojavljuje u P. Svaki element niza predstavlja jedan od znakova koji se pojavljuje u T i P i potrebno je uspostaviti logčku vezu između indeksa u nizu R i stvarne vrednsti znaka koju taj element reprezentuje. Ako bi kao referentni skup uzeli skup znakova iz *ASCII* tabele onda bi ovaj niz imao 128 elemenata. Ukoliko posmatramo genom i skup od samo četiri znaka kojima se on predstavlja onda je moguće svesti ovaj niz na četiri elementa. Izgled ovog niza u situaciji kada je P = TCGC bio bi R = [-1, 1, 2, 0]. Odnosno R(A) = -1 jer se A ne nalazi u P, R(C) = 1, R(G) = 2 i R(T) = 0.

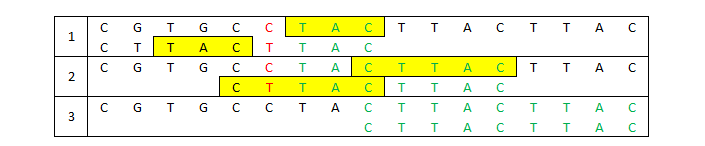
Neka je *i* pozicija znaka u P u trenutku kada se desi nepoklapanje sa odgovarajućim znakom *k* iz niza T. Pravilo lošeg znaka kaže da P treba pomeriti u desno za *Max[1, i – R(T(k))].* Poslednja formula obezbeđuje da se P uvek pomera u desno maker za jedno mesto, ali daje mogućnost i da se neka poređenja preskoče tako što se P pomeri za više mesta ukoliko se *t* nalazi u P na nekoj poziciji iz opsega (0,i).



Slika 4. Primena *Boyer-Moore* algoritma po iteracijama korišćenjem pravila lošeg znaka

Na slici 4. 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 5. Primer *Boyer-Moore* algoritma po iteracijama korišćenjem pravila dobrog sufiksa

Na slici 5. 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. Algoritmi 2

Često je slučaj da je isti niz potrebno pretraživati mnogo puta. 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, a zatim pretraži taj skup. Međutim ovakav pristup donosi i neke nove izazove koje je potrebno rešiti. Prvi je izbor strukture podataka u koju se mogu smestiti svi podnizovi, a drugi je kako takvu strukturu podataka pretraživati efikasno.

Prvi i najjednostavniji algoritam iz ove kategorije koristi sortirani sufiks niz i on će prvi biti predstavljen. Nešto napredniji algoritam koristi sufiks *trie* kao primarnu strukturu podataka, a još bolji i efikasniji algoritam koristi sufiks stabla. Svaki od ovih algoritama kao osnovni nedostatak ima to da je sama struktura podataka koju koristi prevelika u slučaju jako velikih nizova kao što je to najčešće slučaj sa genomima. Jedna od metoda koja pokušava da prevaziđe ovaj problem je kombinuje *Burrow-Wheeler* transformaciju i FM indeks. Svaki od navedenih algoritama će u nastavku biti detaljno objašnjen.

* + 1. 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 6. dat je izgled sufiks niza za tekst ACGTCA pre i posle sortiranja.



Slika 6. Sufiks niz za T = ACGTCA

Sufiks niz za primer teksta ACGTCA je [5, 0, 4, 1, 2, 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 .