SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 4584

Stablo Bloomovih filtara za spremanje sljedova

Luka Škugor

Zagreb, lipanj 2016.

SADRŽAJ

1. Uvod

Usprkos činjenici da su u današnje vrijeme dostupni strojevi koji mogu obraditi veliku količinu podataka u kratkom roku, postoje određeni problemi za koje se još uvijek traže sofisticiraniji algoritmi. Jedan od takvih problema je pronalaženje sličnih nizova. Konkretno, radi se o situaciji u kojoj raspolažemo velikim brojem dugih nizova, a htjeli bi pronaći jednog ili više njih koji je sličan upravo onom nizu kojeg u skupu pokušavamo pronaći. Kada bismo pokušali pretraživati takav skup u primitivnoj maniri, bili bi primorani svaki od nizova iz tog skupa uspoređivati s onim kojeg tražimo - jedan po jedan znak. To za rezultat ima popriličnu vremensku i prostornu složenost. Kako ovakvi nizovi mogu biti vrlo dugi, a skup koji pretražujemo relativno velik, primitivno pretraživanje bi u nekim ekstremnim slučajevima trajalo godinama. Korisnost pronalaženja sličnih nizova leži u činjenici da s jedne strane raspolažemo originalnim nizovima, a s druge pak nizovima koji su donekle izmijenjeni (sadrže određene greške). Takva se primjena primjerice može naći u bioinformatici, gdje se trebaju pretražiti dugi nizovi DNA ili RNA lanaca. Zbog tolike duljine, prilikom čitanja nam se mogu dogoditi određene greške – greške čije rješenje iznalazimo upravo u metodi pronalaženja sličnih nizova.

Prethodno opisan bioinformatički problem bit će obrađivan u nastavku ovog rada. Naime, na raspolaganju imamo bazu proteinskih lanaca Astral[1], čiji su nizovi kraći i sastoje se od abecede veličine 20 znakova. Rješenje ćemo potpuno prilagoditi danom skupu te vidjeti na koji način promjena određenih parametara utječe na rezultate.

Što se tiče sofisticiranijih rješenja, o kojima se govorilo iznad, njih predstavljaju FM index, sufiksno polje i sufiksno stablo. Takve solucije znatno smanjuju ili prostornu ili vremensku složenost, pa su zbog toga korisne u stvarnim aplikacijama. U nastavku ćemo se osvrnuti na jednu od njih, proanalizirati njezine prednosti i mane, te predstaviti strukturu poznatu kao „Bloom filter“ – strukturu namjenjenu smanjivanju prostorne složenosti našeg problema. Bloom filtri se koriste za spremanje nizova te mogu odgovarati na upit nalazi li se neki niz u filtru. Prostorna složenost takve strukture će se zatim dodatno sažeti korištenjem RRR kompresije. Navedeni koraci

rezultiraju stvaranjem nove strukture - Stabla Bloomovih filtara. Ona nam omogućuje prednost spremanja velikog broja dugih nizova u znatno manje memorije – prednost koju ne bismo uživali ukoliko bi ju spremali nemodificiranu. Takvo stablo može biti izgrađeno u veoma kratkom roku, a uz to se može i lako spremiti te proširiti. Isto tako, ne postavlja se zahtjev za pamćenjem nizova koji su indeksirani u stablu.

Nadalje, što se tiče brzine pretraživanja, ona biva povećana, te će sada prosječna složenost upita biti $\mathcal{O}(\log n)$ (gdje je n broj nizova u skupu koji pretražujemo). Rješenje ćemo testirati za različite parametre, i potom usporediti s drugim rješenjem koje je osmišljeno specifično za pronalaženje sličnih proteinskih lanaca.

Prvo ćemo se osvrnuti na već postojeća rješenja te ćemo nakon toga objasniti strukture koje ćemo koristiti za implementaciju rješenja (Bloomov filter i RRR kompresija). Zatim ćemo objasniti strukturu i implementaciju rješenja, popraćenu rezultatom testova provedenih nad već spomenutim bazama proteinskih lanaca.

2. Pretraživanje nizova

S problemom pretraživanja nizova se možemo suočiti koristeći neke već postojeće algoritme. Oni imaju svoje prednosti i nedostatke te ćemo opisati neke od njih kako bi uvidjeli koje su prednosti korištenja Stabla Bloomovih filtara nad već postojećim algoritmima. Generalno, problem pretraživanja sličnih nizova se svodi na ideju da dva niza koja uspoređujemo gledamo kao originalni nepromijenjeni niz i niz na kojem su se dogodile neke greške tokom čitanja (uzrokujući njihovu neidentičnost). Ako uspijemo prepoznati greške na oštećenom nizu (koje uzrokuju neidentičnost), stupanj sličnosti nizova možemo izraziti u terminima njihove razlike, tj. udaljenosti. Najčešći oblik mjerenja te sličnosti je Hammingova udaljenost koja uzima i uspoređuje indekse dvaju nizova te broji na koliko se mjesta nizovi razlikuju. Što se tiče prirode grešaka koje se mogu dogoditi prilikom čitanja nizova, one se uglavnom se svode na brisanje, supstituciju i umetanje.

Većina metoda pri uspoređivanju ne uzima u obzir cijeli niz, već ga cjepka na manje dijelove duljine k. Te dijelove nazivamo k-torka. K-torke su podnizovi fiksne te predefinirane duljine k, danog niza. Možemo ih uzeti na dva načina - s pomičinim prozorom ili bez. Kada uzimamo k-torke s pomičnim prozorom, onda su k-torke nekog skupa svi podnizovi duljine k, za sve indekse od 0 do $n - k$. Ukoliko pak uzimamo k-torke bez pomičnog prozora, tada ćemo indeks povećavati za k (k-torke niza će biti svi podnizevi duljine k, za sve indekse i od 0 do n gdje je i djeljiv s k).

Prije nego što se suočimo s novim načinima indeksiranja i pretraživanja skupa nizova, pogledajmo jedan od dosadašnjih pokušaja efektivnog pretraživanja istih. Jedan od načina pronalaženja sličnih nizova predstavljaju sufiksna polja. To su polja koja u sortiranom redoslijedu pamte sve sufikse nekog niza. Ukoliko primijenimo ideju razbijanja niza u k-torke, tad možemo modificirati našu strukturu kako bi pamtila samo sufikse duljine k-torke k (umjesto da uspoređuje cijelu dužinu sufiksa). Tako dobijemo polje koje u sortiranom redoslijedu pamti sve k-torke nekog niza. Kada bismo nad takvom strukturom postavili upit je li neki niz sličan onomu kojeg smo umetnuli u polje, slijedili bi sljedeće korake. Prvo bismo razbili niz iz upita na njegove k-torke, te bi

zatim za svaki od tih k-torki pokušali pronaći sličnog u našem polju.

Postoje različiti načini na koje možemo određivati jesu li k-torke slične i, ako jesu, u kojoj mjeri. Jedan od njih je primjerice postavljanje bit maske nad k-torkama, te registriranje jeli se dogodilo brisanje, umetanje ili dodavanje elementa u niz za svaki od pojedinih bitova koji su 0. Ovakav način pretraživanja sličnih k-torki bismo mogli optimizirali na način da takve greške ne registriramo tek pri upitu, nego već pri sortiranju strukture. Tad bi zanemarili one indekse niza na kojima je bit 0, te izveli samo jedan upit za sve vrste grešaka. Tako bi upit nad samo jednim nizom imao složenost $\mathcal{O}(\log n)$, gdje je n broj k-torki u nizu iz upita. Kasnije ćemo pokazati na koji način stablo Bloomovih filtara drastično ubrzava vrijeme upita (njegovo najgore vrijeme upita je $\mathcal{O}(sj)$, gdje je s broj nizova u skupu podataka koji pretražujemo, a j prosječan broj k-torki u nizovima).

3. Bloom filtri I RRR struktura

3.1. Bloom filtri

Bloom filter je memorijski efikasna struktura koja omogućava zadavanje upita nad skupom elemenata. Rezultat upita nam može kazati ili da element u strukturi definitivno ne postoji, ili da postoji mogućnost da se ipak nalazi unutar strukture. Jedan bloom filter se sastoji od bit-vektora duljine n i niza funkcija sažimanja. Pri izgradnji jednog bloom filtra, sve su vrijednosti vektora postavljane na nula. Funkcije sažimanja su predefinirane i ne mogu se mjenjati. Kako bi ubacili element u Bloomov filter, trebamo izračunati vrijednosti h_i svake od funkcija sažimanja za dani element. Tada za svaki h_i postavljamo indekse vektora $h_i \bmod n$ u jedan.

Davanje upita u Bloomov filter funkcionira na sličan način kao i umetanje. Prvi dio je isti: prvo izračunamo vrijednosti funkcija sažimanja. Zatim, umjesto da te vrijednosti umećemo u Bloom filter, pribjegavamo ispitivanju koje su vrijednosti vektora na mjestima gdje smo trebali postaviti jedinice, kada bismo isti ubacivali u Bloomov filter. Ukoliko je barem jedna od tih vrijednosti nula, tada možemo biti sigurni da element definitivno nije u filtru. Suprotno, ako su sve vrijednosti na tim mjestima u vektoru jedinice, onda možemo zaključiti da postoji mogućnost da filter sadrži element za kojeg smo postavili upit. Razlog tomu zašto ne možemo sa sigurnošću reći da je element u filtru ukoliko upitom dobijemo pozitivan odgovor, leži u tome što ubacivanjem elemenata u filter može doći do preklapanja bitova. Drugim riječima, neki bit u filtru može biti postavljen na 1 od dvaju ili više različitih elemenata. Primjerice ukoliko imamo elemente A i B koji pri ubacivanju u filtar postavljaju bit vektora na indeksu i. Tada možemo dobiti lažno pozitivan rezultat ako filtar sadrži element A, a mi postavimo upit za element B. Sličan primjer je kad imamo više funkcija sažimanja na Bloomovom filtru te elemente A, B i C takve da elementi A i B pri ubacivanju u filter postavljaju iste ili nadskup onih bitova vektora koji bi element C postavljao. Tada, ukoliko filter sadrži elemente A i B, te ako postavimo upit za element C, opet ćemo dobiti lažno pozitivan rezultat. Ovime vidimo da možemo dobiti lažno pozitivan re-

zultat ako napravimo uniju između dva Bloomova filtra (što će nam biti iznimno bitno kod konstrukcije Stabla Bloomovih filtra).

Sada se pak postavlja pitanje koji bi bili optimalni parametri filtra za određene slučajeve. Oni će nam biti od posebne važnosti pri izgradnji stabla Bloomovih filtra. Najbitniji parametar u Bloomovom filtru je FPR (stopa lažno pozitivnih rezultata). On će odrediti kolika je vjerojatnost da Bloomov filtar odgovori na upit s pozitivnim odgovorom, iako element iz upita nije sadržan u filtru. FPR, duljina filtra te broj funkcija sažimanja su kod optimalnog odabira parametara međuovisni. Stoga željeni FPR obično odaberemo, a optimalne vrijednosti duljine filtra m i broj funkcija sažimanja k koje su potrebne za umetanje elemenata u filtar, izračunamo po formulama.

Kada smo odabrali željeni FPR, sljedeći parametar koji nas zanima je duljina Bloomovog filtra. Pretpostavimo da naša funkcija sažimanja izabire svaki indeks u vektoru filtra s jednakom vjerojatnošću, da je m duljina vektora Bloomovog filtra te da imamo jednu funkciju sažimanja. Tada je vjerojatnost da neki bit vektora filtra, pri ubacivanju elementa u filtar, nije postavljen na 1 sljedeća:

$$1 - \frac{1}{m} \tag{3.1}$$

Ako je, nadalje, k broj funkcija sažimanja, onda je vjerojatnost da neki bit u vektoru filtra nije postavljen na 1:

$$\left(1 - \frac{1}{m}\right)^k \tag{3.2}$$

Konačno, ako u Bloomov filter duljine m, s k funkcija sažimanja, umećemo n elemenata, vjerojatno da neki bit nije postavljen na 1 je:

$$\left(1 - \frac{1}{m}\right)^{kn} \tag{3.3}$$

Ovo će nam biti iznimno važno kako bi mogli odrediti parametre Bloomovog filtra.

3.2. RRR Struktura

Iako su Bloom filtri dosta sažetiji od velikih nizova od kojih smo počeli, upotrebom RRR kompresije možemo dobiti još bolju iskoristivost memorijskog prostora. RRR kompresija predstavlja podatkovnu strukturu koja može odgovarati na upite u složenosti $\mathcal{O}(1)$ i raditi implicitnu kompresiju. Tu je riječ o sažetoj podatkovnoj strukturi

iza koje stoji rezon koji glasi: iako imamo strukturu koja je kompresirana, u svrhu obavljanja operacija nad njom, ne trebamo ju dekompresirati.

4. Stablo Bloomovih filtara

Stablo Bloomovih filtara je nova struktura koja kombinira standardno binarno stablo i Bloomove filtre. Vrijednosti čvorova i listova takvog stabla su Bloomovi filtri. Ideja je takva da svaki roditeljski čvor predstavlja uniju vlastite djece, dok svaki list predstavlja jedan od nizova iz skupa kojeg pretražujemo. Svaki od filtara se pri ubacivanju puni k-torkama onog niza koji se ubacuje.

Najveća prepreka kod izgradnje Stabla Bloomovih fitlra s velikom količinom informacija je prezasićenost filtara prvi vrhu stabla. Kako smo spomenuli, utoliko što je svaki čvor unija vlastite djece, čvorovi pri vrhu stabla će sadržavati više jedinica u vektoru filtra nego oni pri dnu, te će takvi filtri imati znatno veći FPR. Valja napomenuti kako će ovo samo rezultirati sporijim upitima, a ne preciznošću samog rezultata.

Problem prezasićenosti možemo djelomično riješiti raznim metodama poput pravilnog biranja parametara, grupiranja nizova sa sličnim filtrima te uzimanja samo onih filtara koji imaju dovoljnu zastupljenost u nizu. S tako izgrađenim stablom moći ćemo lakše odrediti te odbaciti one čvorove, tj. podstabla, koja sigurno ne sadrže niz našeg upita.

4.1. Konstrukcija stabla Bloomovih filtara

Stablo Bloomovih filtera se gradi tako da u njega ubacujemo jedan po jedan niz. Za svaki niz s se prvo mora izgraditi Bloomov filter $b(s)$ od svih njegovih podnizova jednake duljine. Tek ga nakon toga možemo ubaciti na dno stabla, prolazeći po stablu sa sljedećim pravilima: kada smo u čvoru u, ako u ima samo jedno dijete, ubacujemo novi čvor kao drugo dijete od u; taj čvor sadrži $b(s)$; ukoliko u ima dvoje djece onda uspoređujemo $b(s)$ sa filtrima koje sadržavaju lijevo i desno dijete od u; tada ćemo ubacivati $b(s)$ u ono podstablo koje je bliže po Hammingovoj udaljenosti do $b(s)$.

Konačno, ako u nema djece, onda se stvara novi čvor koji se umeće kao roditelj od u. Taj čvor ima dvoje djece: u i novi čvor koji sadrži $b(s)$. Filter tog čvora je unija od filtra unutar u čvora i $b(s)$. Kako se $b(s)$ spušta po stablu, tako se za svaki čvor

po kojem prođe uređuje njegov filter tako da je nova vrijednost njegova filtra unija dotadašnjeg i $b(s)$. Operacija unije se može izvesti brzo jer je to zapravo izvršavanje OR operacije nad danim vektorima. Na ovaj način sabit ćemo nizove s sličnim Bloom filterima u isto podstablo. To je važno iz dva razloga. Prvi je taj što ovakav pristup rješava problem prezasićenosti filtra. Kad bi imali previše različitih Bloomovih filtra u jednom podstablu, Bloomov filtar korijenskog čvora tog podstabla bi bio popunjen jedinicama. Uz to ćemo dobiti brže upite jer će se cijela podstabla koja nisu nisu slična danom upitu preskočiti i izbaciti iz reda za pretraživanje ranije.

Nakon što smo izgradili kompletno Stablo Bloomovih filtera, svaki od njih po čvorovima možemo kompresirati RRR metodom. Kako RRR kompresiranjem dobijemo sažetu strukturu, moći ćemo postavljati upite bez da dekompresiramo cijelu strukturu.

4.2. Postavljanje upita

Pronalaženje nizova koji su slični nekom nizu s u Stablu Bloomovih filtra se svodi na pronalaženje listova koji sadrže slični Bloomov filter. Upit provodimo tako da prvo razbijemo niz s na k-torke te ih provodimo po stablu od korjena stabla. Na svakom čvoru u, za svaku od k-torki ispitivamo sadrži li Bloom filter tog čvora danu k-torku. Ako filter sadrži više od $\theta_u|K_q|$ k-torki onda kažemo da $b(u)$ sadrži upitani niz, gdje je θ prag okidanja između nula i jedan za čvor u. Ako $b(u)$ sadrži s, onda se pretraga nastavlja na djecu od čvora u. Inače, ako je nedovoljan broj k-torki sadržan u filtru, onda se podstabla čvora u više ne pretražuje (preskače se)[Algoritam 1].

Pretraživanje stabla pak možemo izvesti na način da čvorove stavljamo u red za procesiranje. To neće zahtijevati puno memorije jer trebamo pamtiti samo reference na čvorove te generirani Bloom filter od niza kojeg pretražujemo po stablu. Pretraživanje bi mogli paralelizirati tako da iskoristimo višejezgrenost računala na kojem radimo. Na taj način u teoriji ne bismo povećali memorijsko zauzeće našeg programa, dok bi pak višestruko ubrzali proces pretraživanja. Ipak, u ovom radu rečeni predlošci nisu implementirani. Svi prezentirani testovi i performanse biti će izvedene na jednoj dretvi.

Što se tiče pretraživanja kao procesa, njega bismo mogli dodatno poboljšati kada bi istovremeno tražili više različitih nizova u stablu. Tako ćemo pak poboljšati memorijsku lokalizaciju programa (na način da grupiramo upite po onom čvoru nad kojim se izvršava upit). Primijenivši prethodne korake, čvor se ne treba dohvaćati više puta, što je iznimno korisno ukoliko želimo čvorove čitati direktno s diska koji je znatno sporiji od radne memorije. Ovakav način pretraživanja također može biti paraleliziran na isti način kao i prije.

Algorithm 1 Procesiranje upita

```
1:  function TRAZI(niz)
2:      rezultati ← ∅
3:      redUpita ← korijenStabla
4:      while !prazan(redUpita) do
5:          cvor ← pop(redUpita)
6:          if !imaDjece(cvor) then
7:              rezultati ← rezultati ∪ cvor.niz
8:          else
9:              dodajURed(redUpita, cvor.lijevoDijete)
10:             dodajURed(redUpita, cvor.desnoDijete)
11: function DODAJURED(red, cvor)
12:     if poklapanje(niz, cvor.filter) > θ then
13:         red ← red ∪ cvor
```

4.3. Odabir parametara

Kada sastavljamo Bloomove filtere za Stablo Bloomovih filtra, postoje dva bitna parametra koja moramo uzeti u obzir. To su: duljina Bloomovog filtra m i broj funkcija sažimanja k koje će se koristiti pri umetanju elementa u filter. Također, valja nam odabrati prag okidanja θ za svaki čvor - prag koji primjerice možemo postaviti da ovisi o dubini stabla. Kako će u čvorovima Bloomovi filtri koji su bliži vrhu stabla biti zasićeniji, uputno bi bilo i njihov prag okidanja postaviti većim nego na onim čvorovima koji su dalje od korjena. Tako bismo trebali dobiti bolje rezultate. Radi pojednostavljena implementacije, u ovom rješenju su svi pragovi okidanja θ_u za svaki čvor u postavljeni u fiksni prag okidanja θ. Nadalje, temeljem onoga što znamo o Bloomovim filtrima, odrediti ćemo uvjete za određivanje parametara.

Kada prilikom umetanja novog niza u strukturu provodimo niz kroz stablo, radimo uniju između vektora filtra na čvoru kojeg obilazimo i filtra konstruiranog od tog niza. Ako pretpostavimo da je preklapanje k-torki u filtru uniformno, te da je ta vjerojatnost p, a imamo r nizova u skupu koji pretražujemo onda je očekivani broj elemenata u uniji:

$$n(1 - (1 - p)^r)/p \tag{4.1}$$

Temeljem prethodne formule možemo odrediti optimalan broj funkcija sažimanja h^* koje će minimizirati FPR filtra nad kojem je napravljena unija:

$$h^* = (m(\ln 2)/(n(1-(1-p)^r)/p)) = (p \ln 2)/(1-(1-p)^r)n/m \tag{4.2}$$

Tada će FPR iznositi:

$$\left(\frac{1}{2}\right)^{h^*} \tag{4.3}$$

U našem slučaju, mi smo ipak nismo zainteresirani za upit jedne k-torke nad Bloomovim filtrom već za upit skupa k-torki koji su dio većeg niza. Tako smo ponajvećma zainteresirani za FPR na upitima cijelih skupina k-torki, a ne jednom k-torkom. Uzmimo sada da je niz s kojim postavljamo upit q i on sadrži l k-torki. Ako pretpostavimo da su k-torke neovisne, vjerojatnost da se više od $\lfloor \theta l \rfloor$ lažno pozitivnih k-torki nalazi u filtru s lažno pozitivnom stopom ξ je:

$$1 - \sum_{i=0}^{\lfloor \theta l \rfloor} \binom{l}{i} \xi^i (1-\xi)^{\lfloor \theta l \rfloor - i} \tag{4.4}$$

Prilikom pretraživanja sasvim nam je prirodno da postavimo zahtjev da se barem pola k-torki od upita nalaze u filtru. Ako se više od pola k-torki ne nalazi u nekom filtru ili listu koji sadržava samo jedan niz, možemo reći da takva dva niza nisu dovoljno slična da ih prezentiramo kao jedan od rezultata pretraživanja. U ovom slučaju, naš prag okidanja filtra θ će biti 0.5, te ako izaberemo FPR Bloomovog filtra 0.5, po prethodnoj formuli, malo je vjerojatno da ćemo vidjeti više od $\lfloor \theta l \rfloor$ lažno pozitivnih k-torki u filtru.

FPR Bloomovog filtra vrijednosti 0.5 je mnogo veći nego što se inače koristi kod korištenja Bloomovih filtra, gdje se tipično traže puno manje vrijednosti FPR-a. Ovakva analiza odabira vrijednosti FPR-a uzima za pretpostavku da su k-torke neovisne i nepovezane, što naravno nije slučaj. Međutim, ova analiza na neki način postavlja formalni zaključak da odabirom visokog FPR-a smanjujemo broj grešaka. Također, ako postavimo visoki FPR, možemo koristiti puno manje Bloomove filtre u našem stablu, što će smanjiti memorijsko zauzeće Stabla Bloomovih filtra. Također, kako smo postavili vrijednost FPR-a na 0.5, imat ćemo samo jednu funkciju sažimanja u svakom filtru.

5. Implementacija

U ovom radu su funkcioniranje programa i njegovi parametri točno prilagođeni jednom skupu podataka, odnosno točno specifičnom skupu nizova. Nizovi o kojima je riječ su proteinski lanci. Oni su sastavljeni od abecede duljine 20 znakova te su prosječne duljine oko 500 znakova. Pri ubacivanju k-torki u filtre korištena je metoda cjepkanja niza na k-torke bez klizajućeg prozora. Takav način ubacivanja k-torki jednostavno je odabran po uzoru na drugi algoritam pretrage sličnih nizova - Tachyon. Taj je algoritam napravljen specifično za pretragu sličnih nizova proteinskih lanaca te ćemo upravo s njim uspoređivati rezultate pretrage naše aplikacije. Također, kako bi mogli što više optimizirati programsko rješenje, implementacija je pisana u C++ programskom jeziku kako bi dobili kontrolu nad nižim programskim slojevima.

5.1. Funkcija sažimanja

Vrlo bitna stvar pri funkcioniranju Stabla Bloomovih filtra je jedina funkcija sažimanja koja raspoređuje bitove po filtru. Kako bi dobili što raznovrsnije filtre te podjednaku raspršenost k-torki u filtru, iznimno je važno da funkcija sažimanja bude dobro balansirana. Našu smo funkciju prilagodili tako da uzima u obzir ovaj izniman slučaj gdje su nam na raspologanju nizovi čija se abeceda sastoji od dvadeset znakova, te ih pri izračunu Bloomovih filtera cjepkamo na k-torke duljine 5 znakova. Uzeli smo da svaki od znakova predstavlja jedan od brojeva od 0 do 19, koji će svaki zauzimati po 3 bita. Sada, kada želimo izračunati vrijednost sažimanja, možemo za svaki od znakova poslagati njihove vrijednosti jedan do drugoga. Kako je bitno da funkcija sažimanja bude i brza, to ćemo programski izvesti posmakom svakog od znakova prema lijevo za njegov indeks u nizu pomnožen s 3. Dobivena vrijednost stane u 32 bita podataka odnosno u cjelobrojni podatkovni tip. Tim smo dobili jedinstvenu vrijednost funkcije sažimanja za svaku moguću petorku. Međutim, kako možemo vidjeti po grafu (Slika 5.1), to nam neće dati baš najbolje balansiranu raspršenost po našem filtru koji ima predefiniranu duljinu 144. Da to poboljšamo, dodali smo još male izmjene kojima postižemo bolji

balans raspršenosti unutar naših filtara (Slika 5.2 i Slika 5.3). Izmjene su dodane na način promašaja i pogreške, ali glavni cilj im je bio da malo pomaknu vrijednosti zadnjih bitova, kako se pri dodavanju u Bloomove filtre gleda vrijednost funkcije sažimanja modulom 144. Kako broj 144 zahtjeva 7 bitova za zapisivanje, najveći je fokus na promjenu vrijednosti 7 bitova s najmanjom važnošću. Kako želimo koristiti što manje operacija, a da ipak zadržimo dobru raspršenost odabrana je funkcija sažimanja (Slika 5.2)

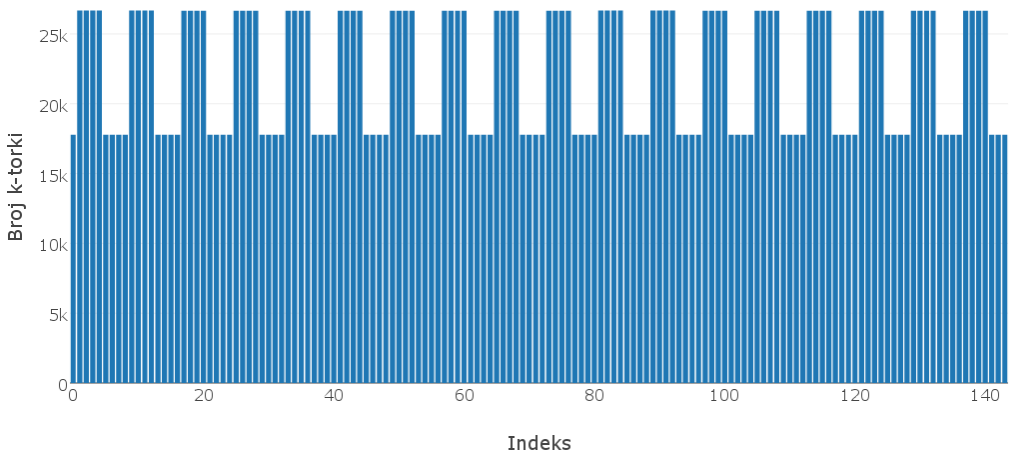
$$(i<<12) + (j<<9) + (k<<6) + (l<<3) + m + l>>1 + k>>2$$

gdje i, j, k, l, m predstavljaju vrijednosti znakova petorke.

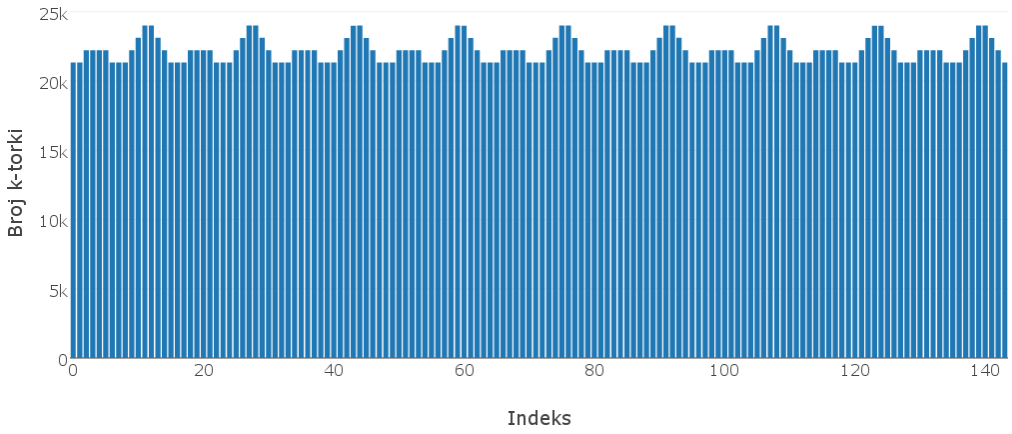
Pretraživanje stabla je napravljeno na najjednostavniji način koji je već prethodno opisan. Imamo red čvorova koji čekaju procesiranje, odnosno uspoređivanje njihovih Bloomovih filtara s onim filtrom koji predstavlja niz koji pretražujemo. Može se napraviti višedretveno rješenje u kojem bi se uspoređivalo više čvorova odjednom. Takvo rješenje je trivijalno za ostvariti, upravo iz razloga što nam redoslijed ispitivanja čvorova nije bitan. Jedino na što bi trebalo paziti je da ne unosimo više čvorova u red istovremeno. Ipak, pri implementaciji ovog rada nije ostvareno nikakvo višedretveno rješenje pa će se svi upiti izvoditi isključivo na jednoj dretvi, odnosno jednoj jezgri.

5.2. Podrezivanje stabla na određenoj dubini

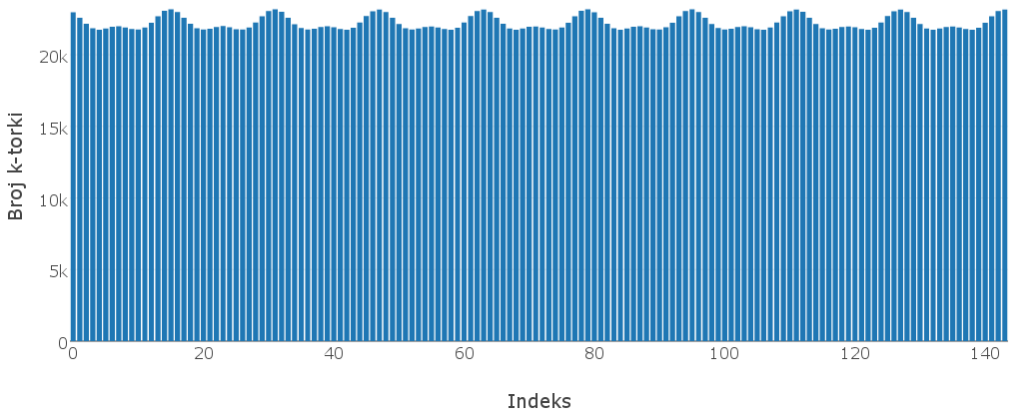
Kako bi dodatno smanjili vremena pretraživanja, dodana je mogućnost podrezivanja stabla na određenoj dubini. Na ovaj način nećemo trebati uspoređivati sve čvorove nekog podstabla već ćemo odmah dobiti neki skup rezultata, skup čija je unija slična našem upitanom nizu. Valja napomenuti da na ovaj način možemo dobiti listu nizova od kojih možda niti jedan uopće nije dovoljno sličan nizu iz našeg upita. Ipak ,ovakav postupak može biti koristan ako želimo ovakvu strukturu koristiti zajedno s nekim drugim algoritmima. Tim više zato što korištenjem takve strukture ne možemo biti stopostotno sigurni da za pozitivan upit stvarno dobivamo sličan niz. Dakle, ovom metodom možemo npr. dobiti listu nekih nizova te potom preciznijim algoritmima provjeravati sličnost između pronađenih nizova i niza iz našeg upita. U našim rezultatima od takvog okidanja nećemo nužno imati neku veliku korist zbog toga što se radi o relativno malom skupu podataka i što su nizovi u skupu dosta kratki.



Slika 5.1: Funkcija sažimanja
$(i << 12) + (j << 9) + (k << 6) + (l << 3) + m$



Slika 5.2: Funkcija sažimanja
$(i << 12) + (j << 9) + (k << 6) + (l << 3) + m + l >> 1 + k >> 2$



Slika 5.3: Funkcija sažimanja

$$(i << 12) + (j << 9) + (k << 6) + (l << 3) + m + l >> 1 + k >> 2 + j >> 3 + i >> 4$$

6. Rezultati

Prije no što se iznesu rezultati, valjalo bi napomenuti činjenicu da su sva mjerenja u ovim eksperimentima izvedena na osobnom računalu sa Intel procesorom koji sadrži 4 jezgre radnog takta 2.4GHz. Unatoč tome što računalo ima 4 jezgre, naš program koristi samo jednu dretvu te efektivno koristimo samo jednu jezgru procesora. Sama implementacija zadatka testirana je na dvjema bazama podataka veličine 3.7MB i 7.4MB. Nad svakom od baza provedeno je 100 upita koje te baze sadrže. Za svaku od baza su izmjerena vremena konstrukcije stabla i pretraživanja po čvorovima. Također, izmjereno je i memorijsko zauzeće za oba eksperimenta. U našem postupku, dodatno ćemo usporediti dobivene rezultate s jednom drugom metodom pretraživanja - Tachyonom koja je dizajnirana specifično za ovaj problem, tj. pretraživanja lanaca proteina u velikoj bazi lanaca.

Što se tiče izgradnje Stabla Bloomovih filtra, ona se sastoji od 3 glavna koraka: stvaranja Bloomovih filtera za svaki od nizova iz skupa koji želimo pretraživati, konstruiranja stabla (i njegovih čvorova nad kojima se radi unija svaki put kad novi niz prođe nad njima) te RRR kompresije svakog od filtra u stablu, bilo na čvorovima ili listovima nakon što je stablo konstruirano. U ovom radu konkretno, sagradili smo Stablo Bloomovih filtra za dvije različite baze proteina. Jedna baza je velika 3.7MB i sadržava nizove s njihovim opisima i detaljima (u njoj se nalazi 13760 proteinskih nizova). Veća baza je dvostruko veća, tj. Zauzima 7.4MB u tekstualnoj datoteci te sadrži 28010 nizova proteinskih lanaca. Nad obe baze smo provodili eksperiment tako da prvo odaberemo slučajnih 100 nizova iz baze te nad izgrađenim stablom provodimo upite koji sadržavaju takve slučajne nizove.

Prvi rezultati su pokrenuti s parametrom duljine filtra od samo 144 bita po vektoru filtra. Duljina od 144 je izabrana upravo zato što bi to bila optimalna duljina filtra ukoliko bismo van našeg stabla imali nezavisan Bloomov filter, te ukoliko bi on sadržavao samo jedan niz. Rezultati takvog prvog pokušaja su možda na prvi pogled i zadovoljavajući. Potrebno vrijeme za konstrukciju stabla za veću bazu iznosi 1.539 sekundi, dok izvršavanje svih 100 upita traje 1.307 sekundi. Međutim, pri ovakvim postavkama se

pojavljuje jedna ozbiljna mana: broj posječenih čvorova je jako velik. Prosječan broj posječenih čvorova s ovakvim postavkama je 14505, što je više od pola broja nizova u bazi. Stablastom strukturom smo pokušali izbjeći upravo to - da ne uspoređujemo veliku količinu filtra, već samo one najsličnije. Kada obilazimo tako veliku količinu čvorova, stablasta struktura nam ne dolazi toliko do izražaja. Štoviše, tako smo možda mogli koristiti i nekakvu listu za spremanje listova, umjesto stabla. Izvor tog problema proilazi upravo iz spomenute duljine filtra. Kada imamo tako malu duljinu filtra, filtri bliži korijenu će brzo postati zasićeni te ćemo filtrirati upite tek na čvorovima koji su dosta dalje od korijena stabla. Kako bi rješili taj problem, odnosno smanjili vrijeme upita i broj posječenih čvorova u stablu, povećat ćemo duljinu Bloomovih filtera u stablu (Slika 6.1). Na taj način ćemo također povećati osjetljivost Bloomovih filtra u čvorovima, pa će se upiti u nekim podstablima prekinuti puno ranije u odnosu na postavku kada smo imali manje filtre, rezultirajući u manjem broju posjećenih čvorova (Slika 6.2).

S druge strane, ovo rješenje ipak ima dvije mane: kada povećamo duljinu filtra, svaki će čvor zauzimati puno više memorije. Dodatno, izgradnja stabla će trajati puno duže (Slika 6.3). Razlog tome leži u činjenici da pri ubacivanju stabla treba iterirati po cijelom filtru i raditi OR operaciju nad filterom u čvoru koji obilazimo. Ovaj dio se može ubrzati na dva načina. Prvi u teoriji dopušta ubrzanje do 64 puta (ili s koliko već bita naš procesor može obraditi odjednom) kako trenutna implementacija iterira po bitovima, a moglo bi se napraviti i tako da se radi OR operacija nad više bitova odjednom. Drugi način je zanimljiviji, i za ovaj će konkretan slučaj (gdje imamo kratke nizove proteina) znatno ubrzati implementaciju.

Što se tiče težine konstrukcije stabla, ona pak leži u dvije funkcije: unija filtra pri obilasku te računanje Hammingove udaljenosti. Kako su u prvoj implementaciji te dvije funkcije napravljene da iteriraju po cijelom filtru, konstrukcija stabla će nam najviše ovisiti upravo o duljini filtra. Ipak, te dvije funkcije možemo napraviti tako da ne ovise o duljinu filtra, već o broju k-torki u nizovima koje ubacujemo. Za funkciju unije, to možemo postići tako da sve k-torke niza kojeg ubacujemo u stablo umetnemo izravno u Bloomov filter s kojim želimo napraviti uniju. Tako bismo, umjesto iteriranja po svim bitovima filtra dvaju vektora, iterirali samo po k-torkama te filtru s kojim želimo napraviti uniju. Jednostavno bismo postavljali bitove na 1 tamo gdje je potrebno. Funkciju računanja Hammingove udaljenosti možemo ubrzati tako da pamtimo koliko je bitova u svakom filtru u stablu postavljeno na 1. Nadalje, pri računanju Hammingove udaljenosti ćemo iterirati po k-torkama te gledati na kojim bi se indeksima u vektoru filtra oni nalazili. Recimo da je s broj unikatnih indeksa na koje k-torke umetajućeg

niza padaju, a da su vrijednosti bitova u drugom filtru na tim indeksima postavljeni na 1. S druge strane, recimo da je a broj unikatnih indeksa na koje k-torke umetajućeg niza padaju, a da su vrijednosti bitova u drugom filtru na tim indeksima postavljeni na 0. Također, neka je b broj bitova u filtru koji su postavljeni na 1. Sada možemo reći da je Hammingova udaljenost između umetajućeg niza i Bloomovog filtra kojeg trenutno obilazimo:

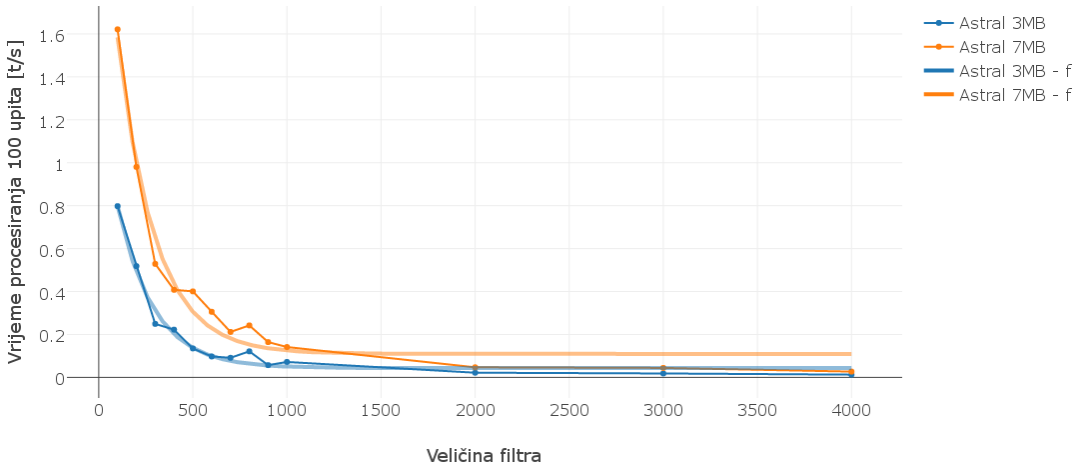
$$b + a - s \tag{6.1}$$

Kako možemo vidjeti po grafu (Slika 6.4), takva nam implementacija znatno smanjuje porast vremena izgradnje stabla s porastom duljine filtra, dok s druge strane povećava zauzeće memorijskog prostora (Slika 6.5).

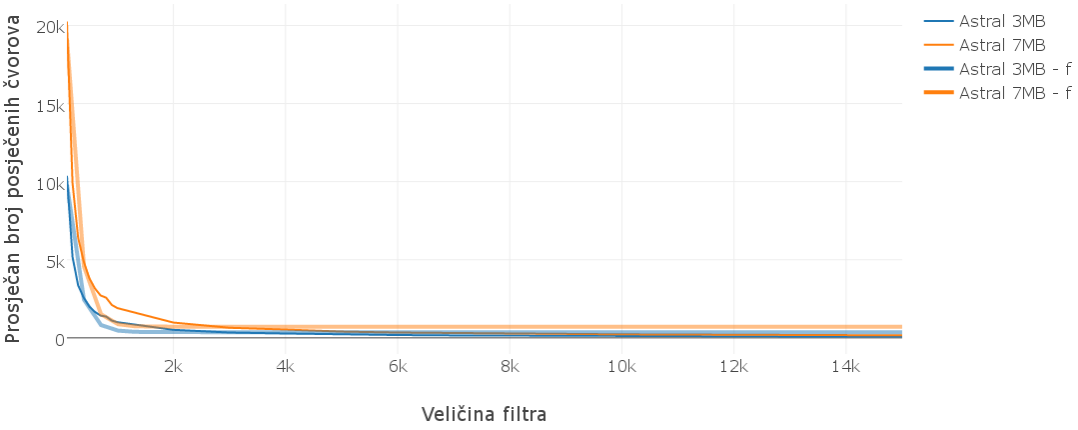
Sumirano, promjenom duljine filtra smo dobili smo nešto sporiju konstrukciju stabla. Pošto bi u teoriji trebali dobiti još manja vremena konstrukcije stabla, možemo zaključiti da implementacija ipak nije optimalna. To možemo potkrijepiti i činjenicom da je graf ovisnosti konstrukcije stabla o duljini filtra (Slika 6.4) dosta deformiran i nelinearan. S druge strane, znatno smo ubrzali procesiranje upita nad stablom.

Dodatno, možemo pogledati kako duljina filtra ovisi o rezultatima. Kao što možemo vidjeti iz dosadašnjeg saznanja o Bloomovim filtrima, broj lažno pozitivnih rezultata je tim veći što je duljina filtra manja. To možemo vidjeti i na našoj implementaciji gdje je za premalene duljine filtra broj lažno pozitivnih rezultata relativno velik (Slika 6.6 i Slika 6.7). Ipak, prijašnjim povećanjem filtra lako rješavamo taj problem te je na duljinama filtra preko 500, taj problem gotovo uklonjen.

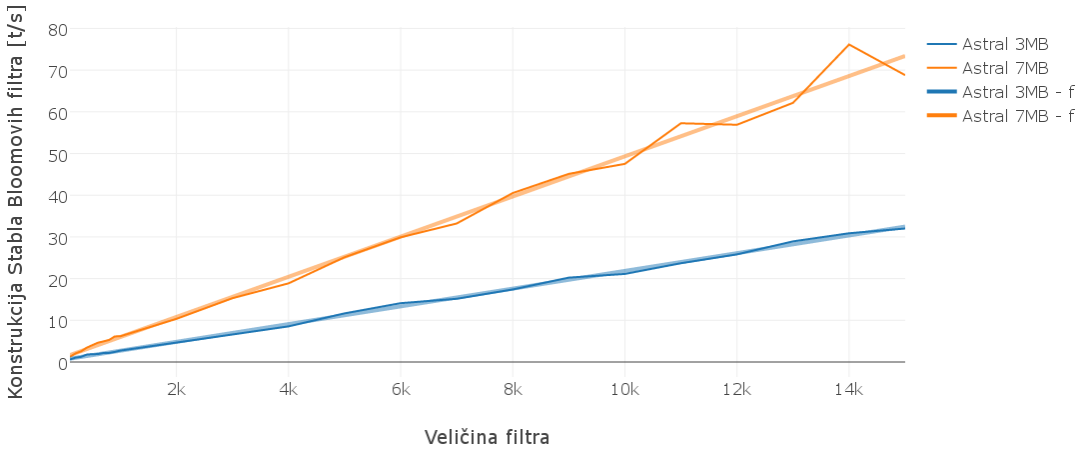
Za kraj, možemo usporediti vrijeme upita za Stablo Bloomovih fiiltra i Tachyon nad Astral bazama. Za usporedbu smo koristili iste hardverske postavke. Za Stablo Bloomovih filtra su korišteni jednaki parametri korišteni i u prvom testu s iznimkom duljine filtra. Duljinu smo proizvoljno postavili na 800 iz razloga što je to otprilike optimalna točka. S takvom postavkom nećemo zauzimati previše memorije, a povećanjem duljine filtra ionako ne možemo previše skratiti vrijeme upita. Kako možemo vidjeti sa slike (Slika 6.8), Stablo Bloomovih filtra je gotovo dvostruko brže od Tachyon pretraživanja.



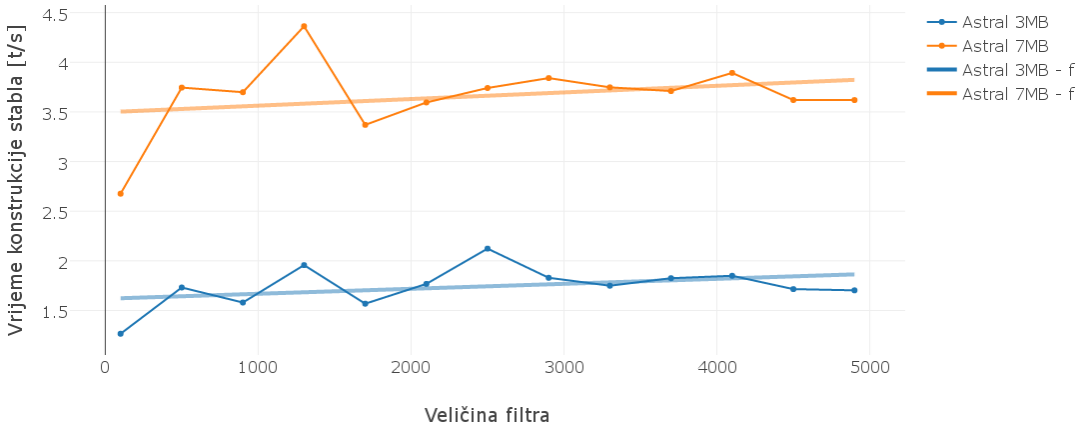
Slika 6.1: Brzina upita



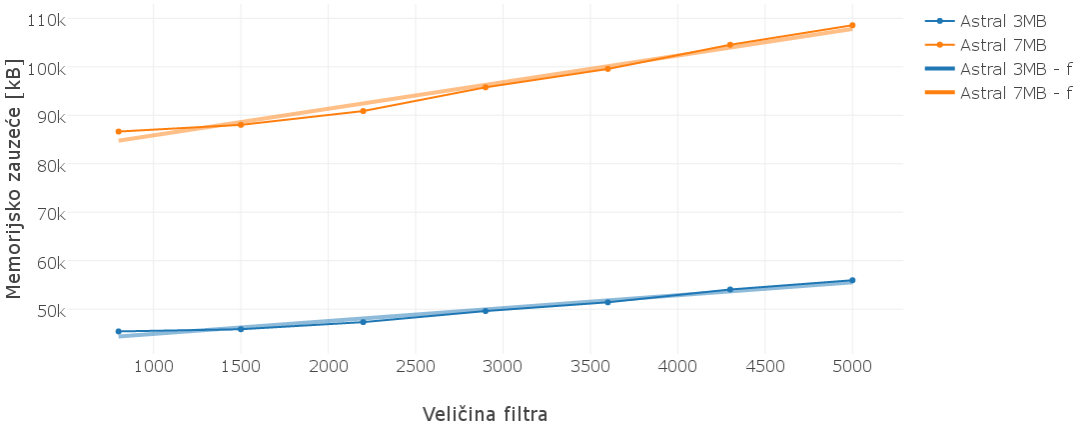
Slika 6.2: Prosječan broj posjećenih čvorova



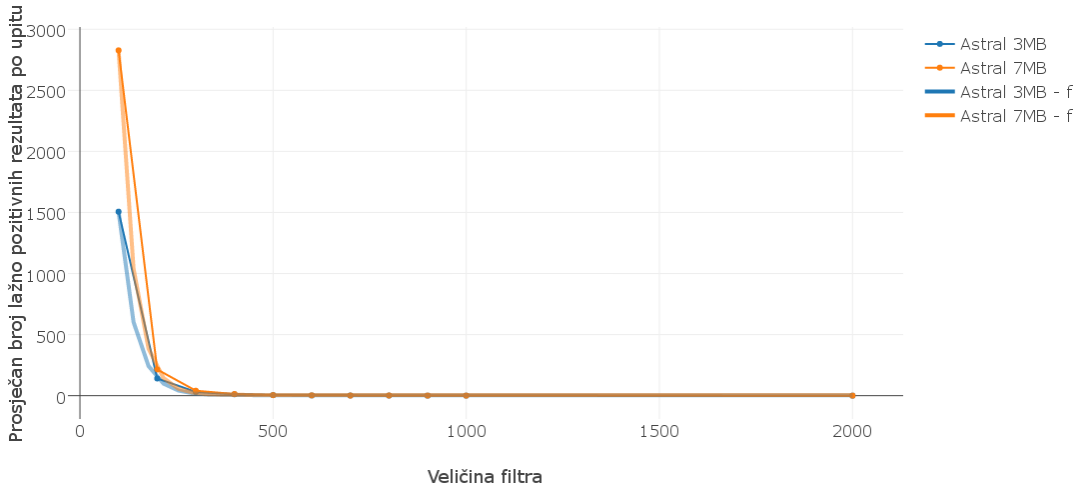
Slika 6.3: Prvotna implementacija konstrukcije stabla



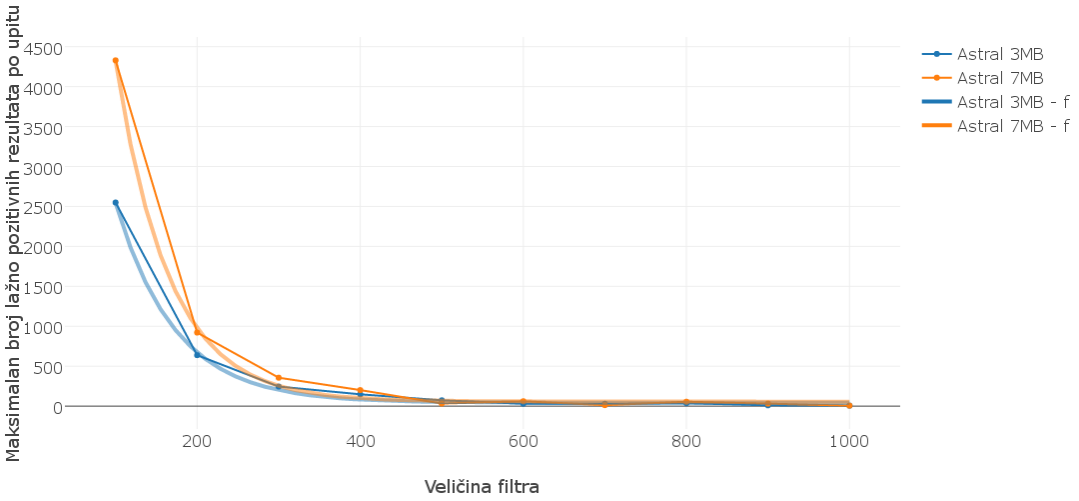
Slika 6.4: Ubrzana implementacija konstrukcije stabla



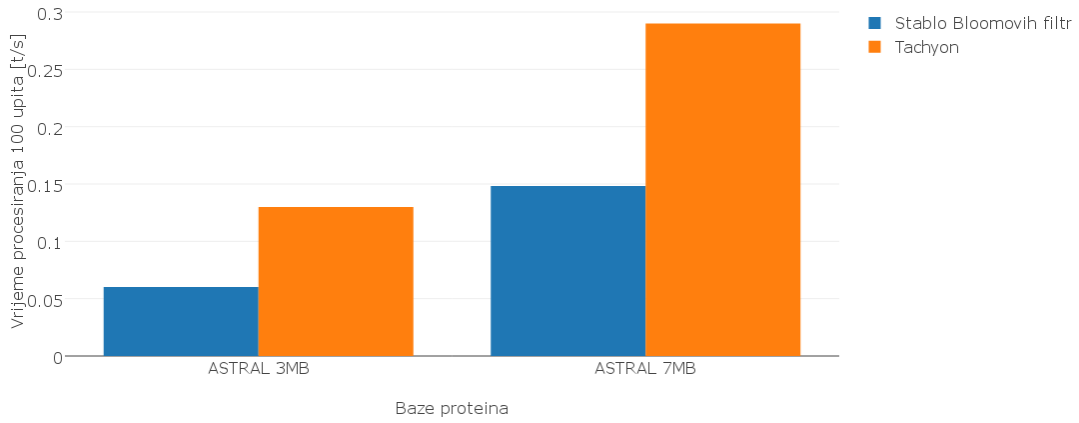
Slika 6.5: Memorijsko zauzeće



Slika 6.6: Prosječan broj lažno pozitivnih rezultata



Slika 6.7: Maksimalan broj lažno pozitivnih rezultata



Slika 6.8: Usporedba brzina upita

7. Zaključak

Stablo Bloomovih filtara je nov i inovativan pristup pretraživanja sličnih nizova u velikoj bazi. Iako može biti korišten za pretragu bilo kakvih nizova, prezentirali smo njegovu uspješnost u pretrazi baze proteinskih lanaca. Rezultati su pokazali da može biti i do nekoliko puta brži od najbržih dosadašnjih rješenja pretrage sličnih nizova. U rješavanju zadatka, suočili smo se s glavnim problemom ove strukture, a to je zasićenje filtra. Ovdje taj problem nismo pokušali riješiti na neki sofisticiraniji način zbog toga što se radi o relativno kratkim nizovima, a i rješenje za takav skup još nije osmišljeno. Ipak, povećavali smo duljinu filtra do te granice da ne dobijemo preveliko zauzeće memorije, a da dobijemo znatno brže rezultate. Stoga, najvažnija stavka za poboljšanje ovakve strukture je pronalazak novih načina smanjivanja zasićenosti filtra.

Implementacija ovog rada dostupna je na: *https://github.com/lukadante/sbt*

LITERATURA

[1] John-Marc Chandonia;Naomi K. Fox;Degui Zhi;Gary Hon;Loredana Lo Conte;Nigel Walker;Patrice Koehl;Michael Levitt;and Steven E. Brenner. Astral sequences & subsets, 2016. URL `http://scop.berkeley.edu/astral/ver=2.06`.

[2] Fox NK;Brenner SE;Chandonia JM. Scope: Structural classification of proteins—extended, integrating scop and astral data and classification of new structures. 2014.

[3] Brad Solomon;Carl Kingsford. Fast search of thousands of short-read sequencing experiments. 2016. URL `http://www.nature.com/nbt/journal/v34/n3/full/nbt.3442.html`.

[4] Eli Mitzenmacher, Michael; Upfal. *Probability and computing: Randomized algorithms and probabilistic analysis*. Cambridge University Press, 2005.

Stablo Bloomovih filtara za spremanje sljedova

Sažetak

Kako u rješenjima problema pretraživanja sličnih nizova još uvijek postoji mjesta za napredak i poboljšanje, u ovom smo radu istražili jedan novi pristup rješavanju tog problema. Rješenje o kojem je riječ naziva se stablo Bloomovih filtra te nudi poboljšanja nad dosadašnjim metodama pretrage sličnih nizova. Zauzvrat ima i mane, koje su specifične za ovu strukturu, kao što je prezentiranje lažno pozitivnih rezultata.

Ključne riječi: Bloomov filter, stablo, proteinski lanci, slični nizovi

Tree of Bloom filters

Abstract

As today's inexact matching algorithms have plenty of room for improvement, in this work we explored a new approach to solve inexact matching problems. The name of approach we explored is Bloom filter tree and it offers improvements over existing inexact matching algorithms. In return, it has its drawbacks, like presenting false positive results.

Keywords: Bloom filter, tree, proteins, similar sequences