SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Projekt u sklopu kolegija Bioinformatika 1

The Logarithmic Dynamic Cuckoo Filter

Luka Mucko i Lea Faber

Zagreb, 2023.

Sadržaj

Uvod ... 1

1. Opis i vizualizacija LDCF algoritma ... 2

1.1. Opis algoritma ... 2

1.2. Vizualizacija algoritma ... 3

1.2.1. Slučaj kada je prvi bucket pun ... 5

1.2.2. Izbacivanje žrtve ... 6

2. Analiza LDCF algoritma u brojanju k-mera ... 8

2.1. Točnost ... 9

2.2. Vrijeme izvođenja ... 10

2.3. Utrošak memorije ... 11

Zaključak ... 17

Literatura ... 18

Uvod

Pojava „big data” aplikacija pokazala je važnost načina na koji su elementi nekog skupa (seta) organizirani i spremljeni. Dobro osmišljena struktura skupa omogućuje pristup informacijama elemenata te operacijama koje možemo vršiti nad njima poput testiranja pripadnosti, umetanja i brisanja. Kako bi se skup prikazao na što učinkovitiji način, razvijene su strukture podataka za organizaciju skupova kao što su Bloom filteri i Cuckoo filteri. Međutim, postojeći filteri su ograničeni kada je riječ o predstavljanju skupova s velikim dinamičkim podacima. Kako bi se to riješilo, predložen je dinamički Cuckoo Filter (DCF), ali njegov postupak provjere elemenata linearno povećava troškove izračuna i prostornog zauzeća s rastom veličine skupa. Zbog toga je Fan Zhang predložila novu strukturu podataka za predstavljanje skupa s velikim dinamičkim podatcima, Logaritamski Dinamički Cuckoo Filtar (LDCF), koji smanjuje najgore vrijeme umetanja i testiranja elemenata, kao i prostorno zauzeće, s linearne na logaritamsku razinu. Ona također uvodi kompaktni LDCF dizajn, koji dodatno smanjuje vrijeme umetanja i testiranja elemenata na kompleksnost O(1). [1]

1. Opis i vizualizacija LDCF algoritma

1.1. Opis algoritma

Za stvaranje LDCF objekta potrebno mu je dodijeliti vrijednosti broja „bucket”-a, broja polja svakog bucketa (entries) i veličinu „fingerprint”-a koji se koristi.

Fingerprint dobivamo hashiranjem ulaznog stringa i pretvaranjem tog hasha u niz nula i jedinica.

Pri stvaranju LDCF-a, stvara se inicijalni Cuckoo Filter na razini 0 koji nema prefixa – prefix je string nula i jedinica koji opisuje početne vrijednosti/znamenke spremljenih fingerprinta do razine tog CFa , a u buckete samog CF su spremljeni ostatci tog fingerprinta (bez prefixa).

Pozivanjem funkcija „insert”, „search” ili „remove” LDCF zapravo poziva te iste funkcije ugniježđenog CF razreda.

Svaki CF sadrži pokazivače na 2 iduća CFa – cf0 i cf1. Ako se CF smatra popunjenim onda inicijalizira jedan CF s prefixom kojem je dodana 0 i jedan s prefixom kojem je dodan 1 te gleda znamenku fingerprinta na razini novo stvorenih CFova i pokušava unijeti isti fingerprint u novo nastali CF s prikladnim prefixom.

Funkcija „insert” najprije provjerava jel trenutni CF ima inicijaliziranu djecu (cf0 i cf1), ako ima to znači da je pun te ovisno o prefixu pokušavamo unjeti fingerprint u prikladno dijete. Ako nema listove onda smatramo da u CFu ima mjesta za spremanje vrijednosti.

Kada želimo umetnuti vrijednost fingerprinta u CF najprije izračunamo 2 indeksa bucketa. Prvi je izračunat pomoću sha1, a drugi md5 hashiranja. Ako u prvom bucketu nema niti jedan slobodan entry onda gledamo u bucketu drugog indexa. Ako su oba bucketa zauzeta, ne sadrže string „N...”, dolazi do relokacije žrtve.

Žrtva je odabrana nasumično iz nekog entryja bucketa prvog indexa . Puni fingerprint žrtve je dobiven konkatenacijom prefiksa CFa i vrijednosti žrtvina entryja. Na žrtvino mjesto stavljen je originalan fingerprint te računamo indekse fingerprinta žrtve i ponavljamo postupak.

Konstantom „MAX_RELOCATION” definiramo broj mogućih relokacija „žrtve” prije stvaranja listova/djece tog CFa – time smatramo da je roditeljski CF pun.

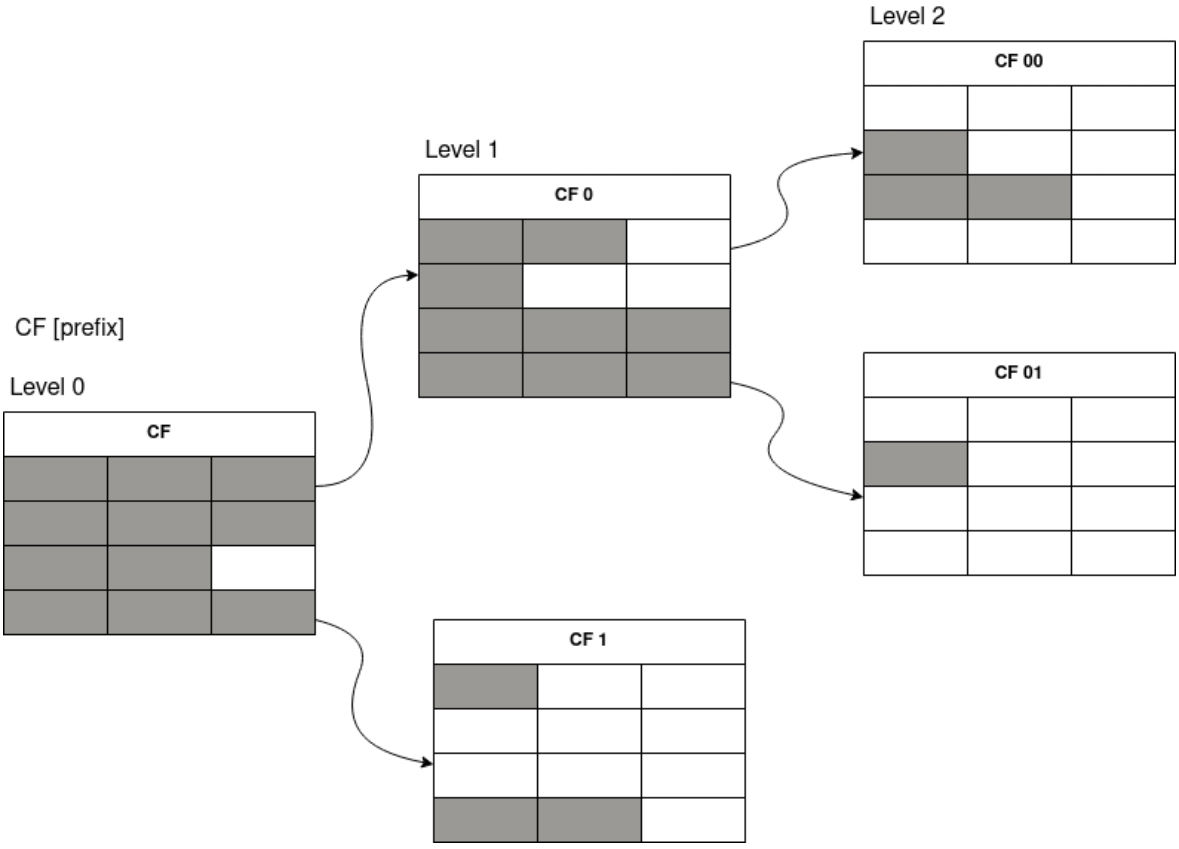
Metoda „search” najprije izračuna fingerprint tražene vrijednosti te od inicijalnog CFa pretražuje stablo ovisno o prefiksima.

Metoda „remove” je implementirana slično kao i „search”, samo što nakon pronalaska izbriše traženu vrijednost.

Sve funkcije ako su uspješne vračaju boolean vrijednost „true”, a u suprotnom „false”.

1.2. Vizualizacija algoritma

Želimo umetnuti podatak čiji je fingerprint „00110” u LDCF koji već sadrži neke podatke. LDCFova „insert” funkcija najprije provjeri je l' se ovaj fingerprint već nalazi u LDCFu, ako se ne nalazi pokreće insert funkciju inicijalnog (level 0) CFa, a u suprotnom vraća „false”.

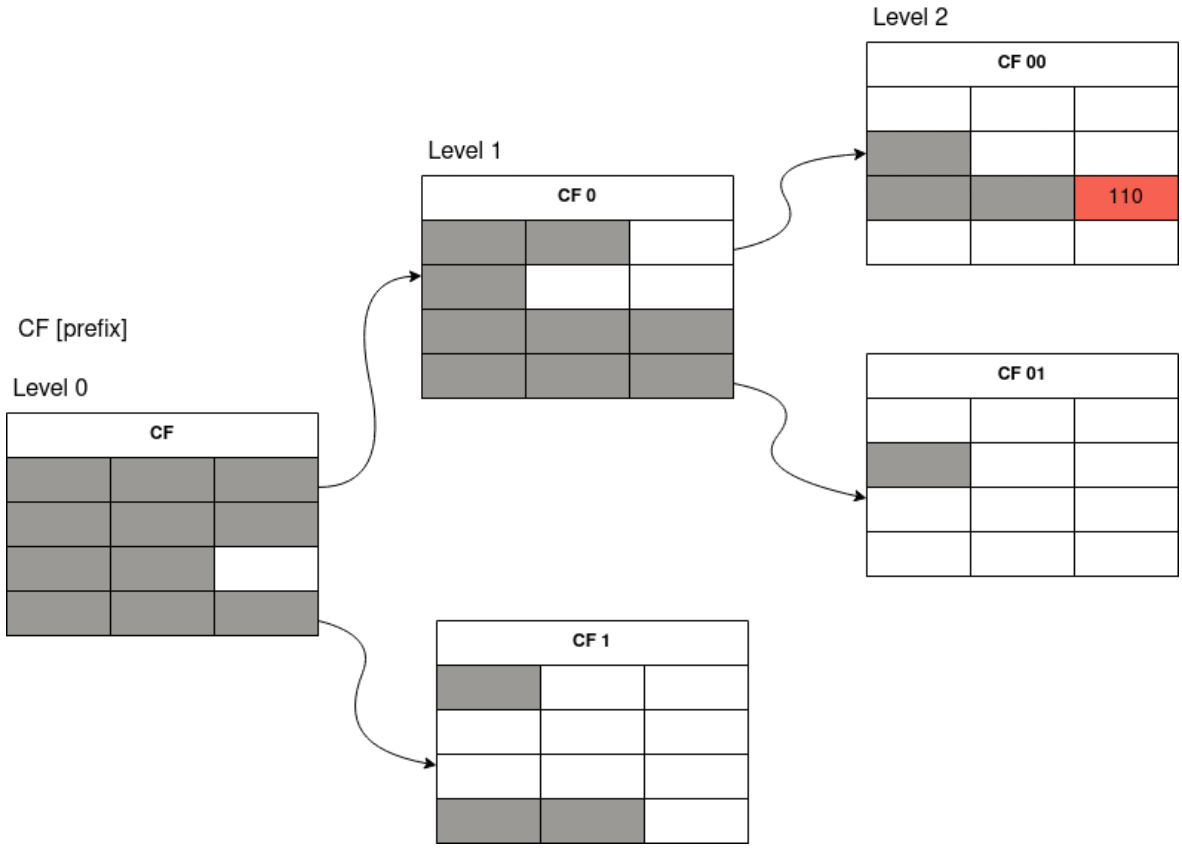


Slika 1. LDCF prije pokretanje funkcije insert

Početni CF prepoznaje da ima već inicijalizirane cf0 i cf1 grane što znači da u njemu samom više nema mjesta te treba umetnuti podatak u neku od ovih grana. CF provjerava koja se znamenka nalazi na fingerprintovom indeksu trenutnog levela, u ovom slučaju na indeksu 0 se nalazi znamenka 0 i po tome odluči da treba nastaviti umetanje fingerprinta u svojoj cf0 grani (CF 0).

Ista radnja se odvija u „CF 0" koja na posljetku provjeri fingerprintevu znamenku na indeksu 1 i kako je ona 0 nastavlja umetanje u svoj cf0 koji je obilježen s prefiksom 00 otkud i dobiva naziv „CF 00".

Kako „CF 00" nema svoje cf0 i cf1 grane smatramo kako u njemu ima mjesta za umetanje te računamo indekse bucketa gdje bismo smjestili fingerprint. Dobivamo indekse 2 i 3 te najprije pokušavamo umetnuti podatak u bucket s indeksom 2. Nalazimo prvi slobodni entry u njemu te tamo upisujemo string fingerprintovog ostatka kad mu maknemo prefiks „00", tj. u entry spremamo „110".



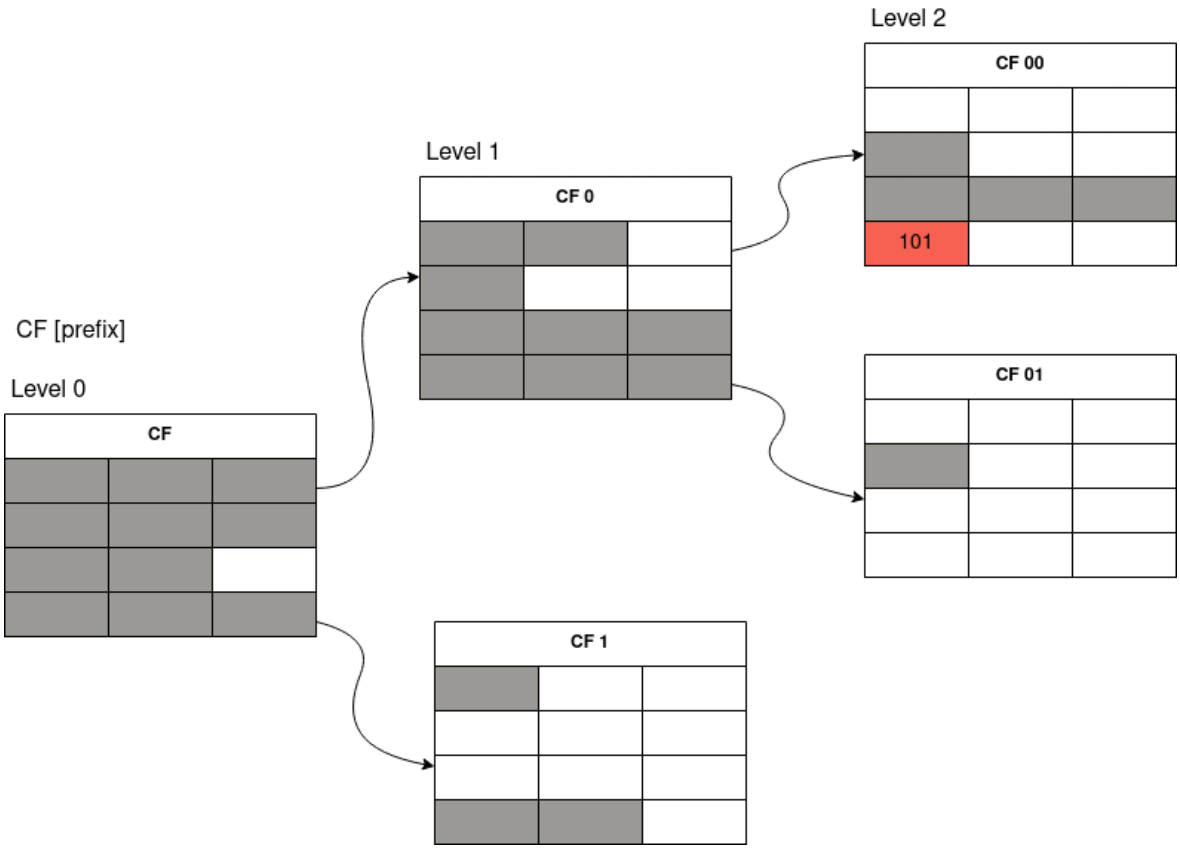
Slika 2 LDCF poslije umetanja 00110

Ako pokušamo unijeti novi podatak s fingerprintom „00101” koji još nije zabilježen u LDCF-u, algoritmom ćemo se opet naći u „CF 00”. Zamislimo 2 slučaja:

1.2.1. Slučaj kada je prvi bucket pun

Fingerprint „00101” generira indekse 2 i 3.

Svi entryiji 2. bucketa su puni te pokušava staviti vrijednost u prvi slobodan entry 3. bucketa.



Slika 3 LDCF poslije umetanja 00101

1.2.2. Izbacivanje žrtve

Slučaj 1

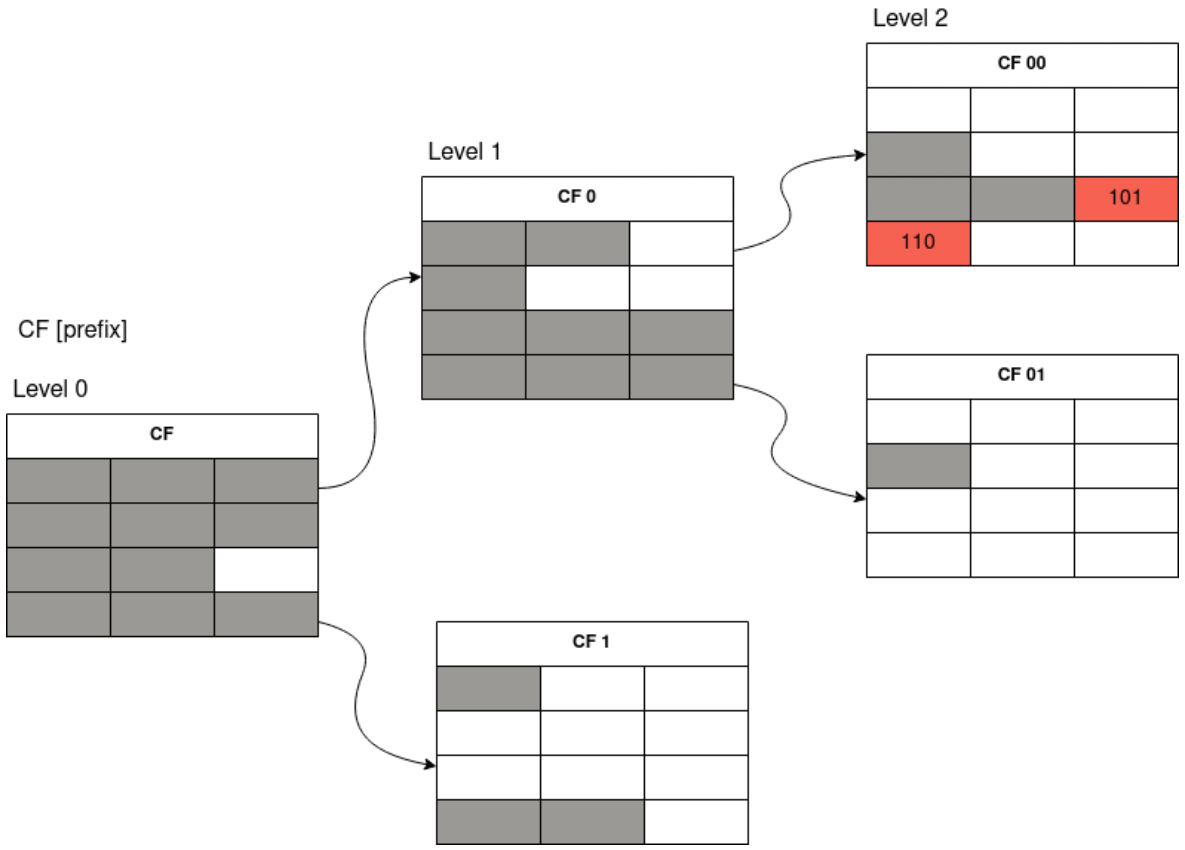
Fingerprint „00101” generira indekse 2 i 2.

Kako u oba slučaja su svi entryiji bucketa puni potrebno je nasumično izabrati „žrtvu”.

Npr. uzmimo da je za žrtvu izabrana 3. ćelija u koju smo prethodno spremili „110”.

Oporavimo žrtvin puni fingerprint spajanjem prefiksa CFa, u ovom slučaju „00” i sadržaja entryja „110” kako bismo dobili „00110”.

Na žrtvino mjesto spremimo početnu vrijednost koju želimo spremiti te računamo žrtvine indekse (2 i 3) kako bismo njenu vrijednost pospremili u neki drugi bucket što i uspijevamo u bucket 3.

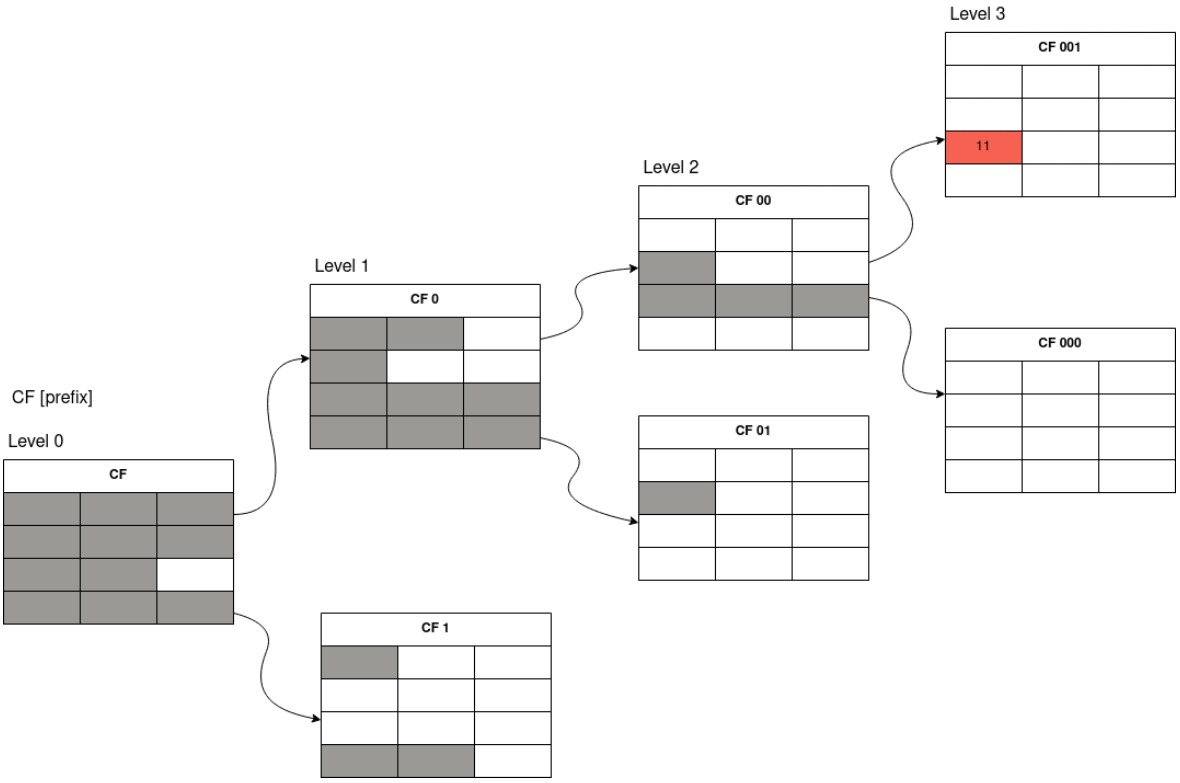


Slika 4 LDCF poslije izbacivanja žrtve i umetanja 00101, slučaj 1

Slučaj 2

Da je „žrtva" generirala indekse bucketa čiji su svi entryiji popunjeni proces izbacivanja nove žrtve bi se ponavljao sve dok se ne nađe slobodan entry ili se taj proces ponovi onoliko puta koliko je definirano u konstanti „MAX_RELOCATION".

Npr. proces izbacivanja i smještanja žrtve se ponovio „MAX_RELOCATION" puta jer su svi fingerprinti žrtava generirali indekse 2 i 2, a vidimo da je bucket 2 pun. Recimo da je fingerprint zadnje zabilježene žrtve „00111". Kako smatramo „CF 00" punim, on stvara 2 nova CFa – „CF 000" i „CF 001" te kako je 3. znamenka posljednje žrtve „1" pokreće se insert metoda tog fingerprinta u „CF 001" koja sprema vrijednost „11" u svoj bucket na indeksu 2.

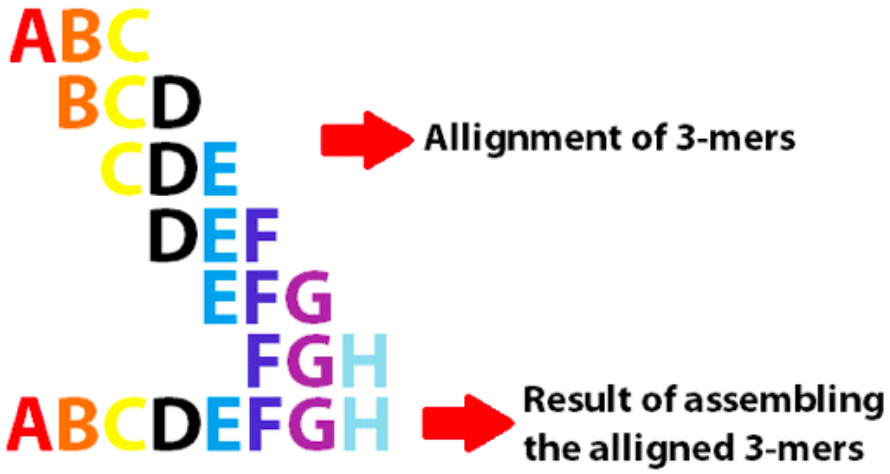


Slika 5 LDCF poslije izbacivanja žrtve i prebacivanje u novi CF

2. Analiza LDCF algoritma u brojanju k-mera

U bioinformatici k-mer je podniz duljine k iz neke biološke sekvence. Za svaki znak u nizu, osim onih koji su udaljeni k-1 od kraja niza, možemo definirati k-mer koji započinje u tom znaku i završava u k-tom znaku dalje.

npr:



Slika 6 3-meri u nizu ABCDEFGH. [8]

Svaki niz duljine L sadrži L-k+1 k-mera, a broj mogućih kmera u genomu je 4^k.

LDCF ćemo iskoristiti u svrhu brojanja k-mera. Svaki k-mer iz genome E-coli. ćemo dodati u LDCF objekt. LDCF prilikom ubacivanja podataka vraća True ili False ako je objekt dodan u LDCF, tj. nije prije postojao u LDCF-u. Tako možemo brojati jedinstvene k-mere u genomu.

2.1. Točnost

Postoje mnogi alati koji broje k-mere u nekom genomu. Jedan takav alat je Jellyfish. Jellyfish koristi Bloom filtere za brojanje k-mera.

Jellyfish se može pokrenuti uz sljedeće naredbe:

$ jellyfish count -m <k> -o <output.jf> -s <hashsize> <input.fasta>

$ jellyfish stats <output.jf>

Usporedimo rezultate LDCF-a i Jellyfisha.

Tablica 1 broj kmera u genomu

K	LDCF	Jellyfish
10	927256	927256
20	5298013	5298013
50	5342621	5342621
100	5370269	5370269
200	5394027	5394027
500	5424848	5424848

Tablica 2. broj kmera u genomu

LDCF je pokrenut sljedećom naredbom:

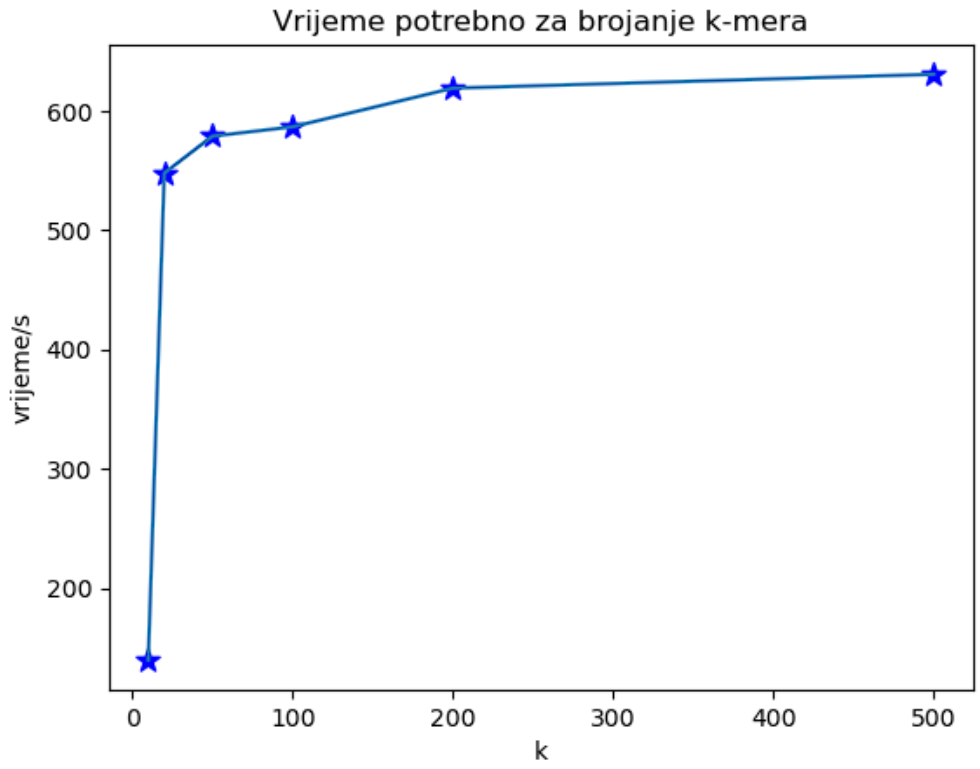
$ ./LDCF.out 10000 100 256 <k> <input>

Bitno je uzeti dovoljno veliku veličinu fingerprinta da bi LDCF brojao točno.

2.2. Vrijeme izvođenja

Uz pomoć <chrono> smo izmjerili vrijeme potrebno za dodavanje svih k-mera u LDCF.

LDCF se izvodio na 6-jezgrenom procesoru s 16 GB RAM-a.



Slika 7 Vrijeme potrebno za dodavanje svih k-mera iz genoma u LDCF.

Tablica 3 Vrijeme potrebno za brojanje k-mera

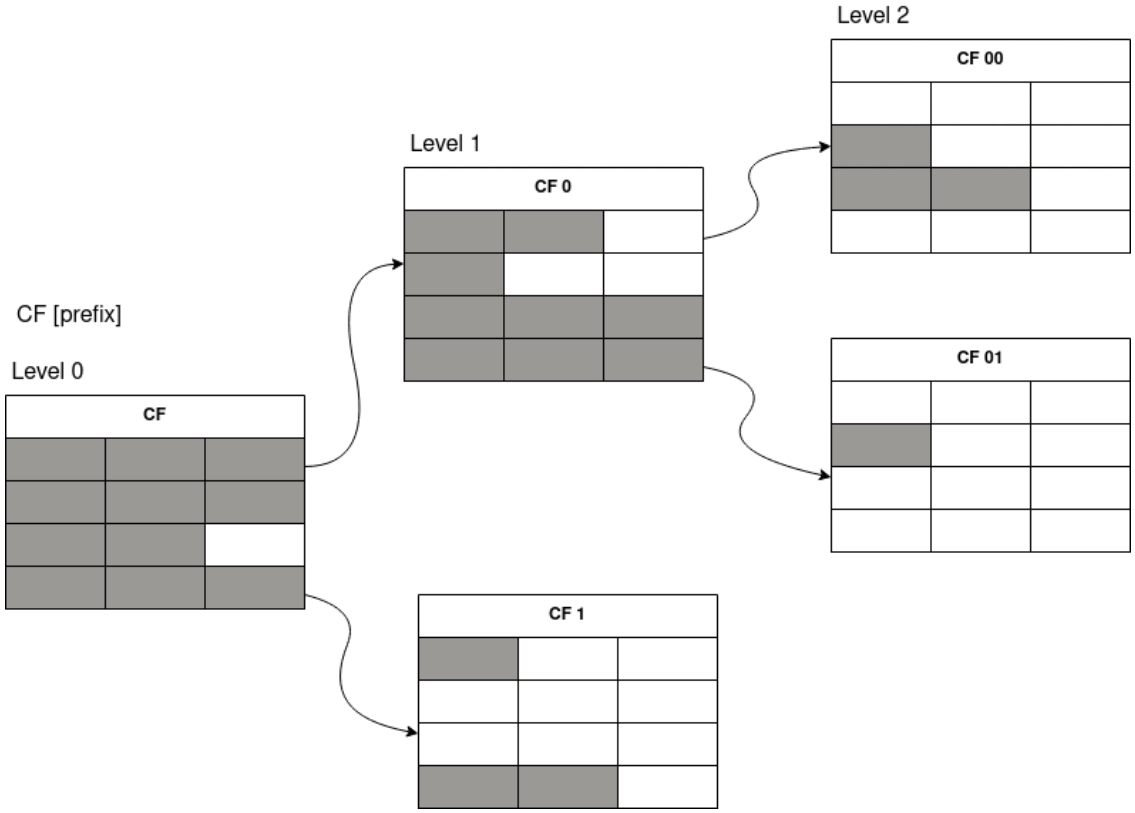
k	10	20	50	100	200	500
Vrijeme/ms	139426	547919	578915	586751	619190	630851

Performansa ove implementacije nije idealna. Za sve testirane k-mere osim 10-mera, vrijeme izvođenja je bilo oko 10 minuta.

2.3. Utrošak memorije

Utrošak memorije mjerimo kao alociranu memoriju za cijeli LDCF bez članskih varijabli.

Ukupna memorija programa je tada zbroj memorije svakog CF-a u LDCF-u.



Slika 8 LDCF

Na l-

tom levelu neki CF s veličinom fingerprinta će zauzimati:

#buckets * #entries * (fingerprint_size – level) / 8 [B]

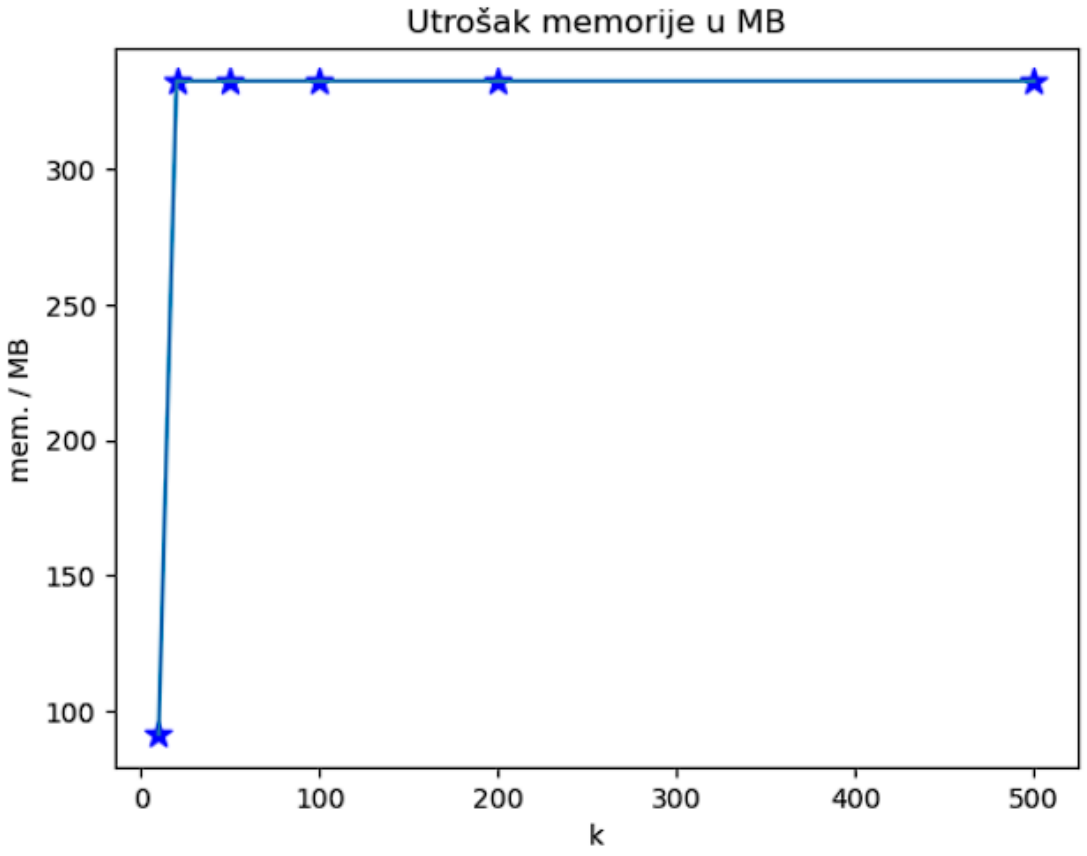
Za LDCF na slici 8. uz fingeprint size fp alocirana memorija će biti:

4 * 3 * (fp + 2 * (fp-1) + 2 * (fp-2)) [b]

Uz dane veličine k-mera i parametre LDCF-a s kojim je bio pokrenut dobivamo sljedeće utroške memorije:

K	10	20	50	100	200	500
Zauzeće memorije /MB	91.3143	332.594	332.594	332.594	332.594	332.594
Broj CF-a	3	11	11	11	11	11

Vidimo da je 11 CF-a u LDCF-u sa 10,000 bucketsa, 100 polja u svakom bucketu i veličinom fingerprinta 256b dovoljno da sadrži sve jedinstvene 20, 50, 100, 200 i 500 - mere u genomu E. Coli.



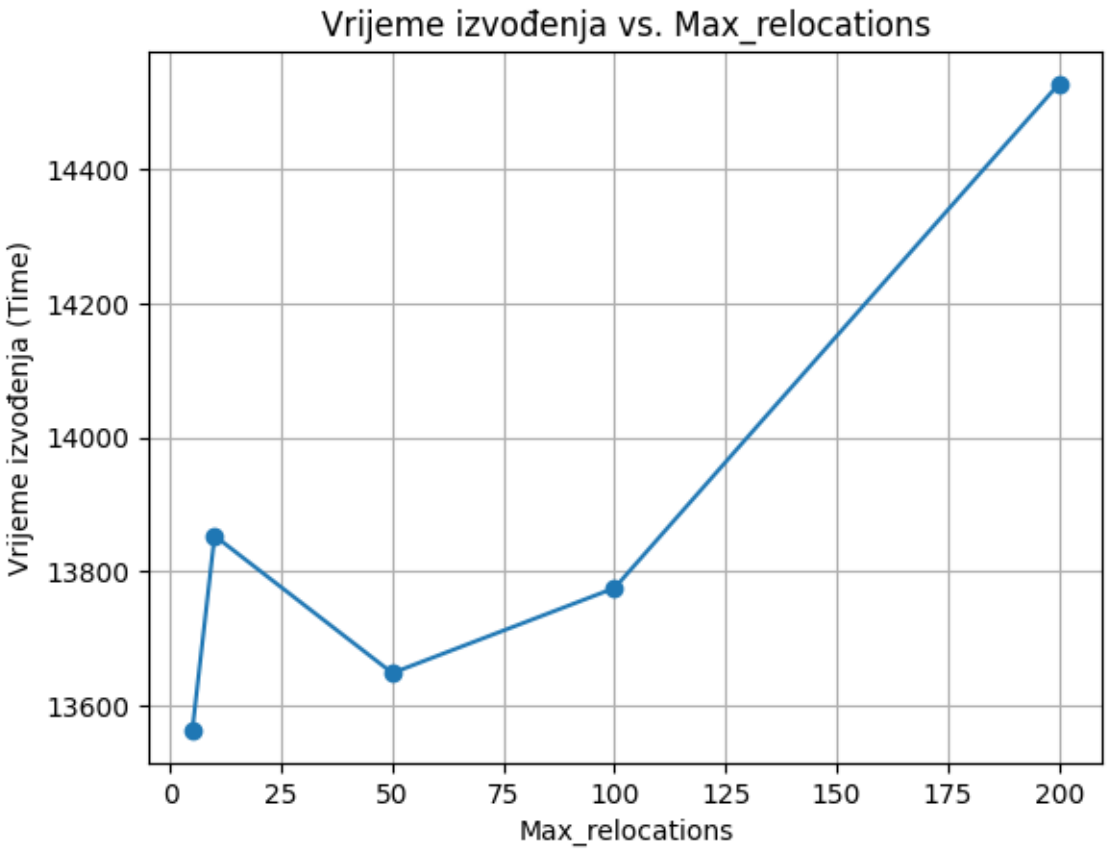
Slika 9 Utrošak memorije za brojanje k-mera.

2.4. Usporedba performansi ovisno o parametrima

Broj izbacivanja žrtve prije stvaranja novog CF-a, ukoliko je odabrani bucket pun. (k=100)

Tablica 4 Ovisnost memorijskog zauzeća o max_relocations

Max_relocations / info	5	10	50	100	200
Vrijeme izvođenja / ms	13563	13853	13648	13775	14528
Veličina u MB	2.57635	2.58368	2.58365	2.59833	2.58362
Broj CF-a	695	697	697	701	697

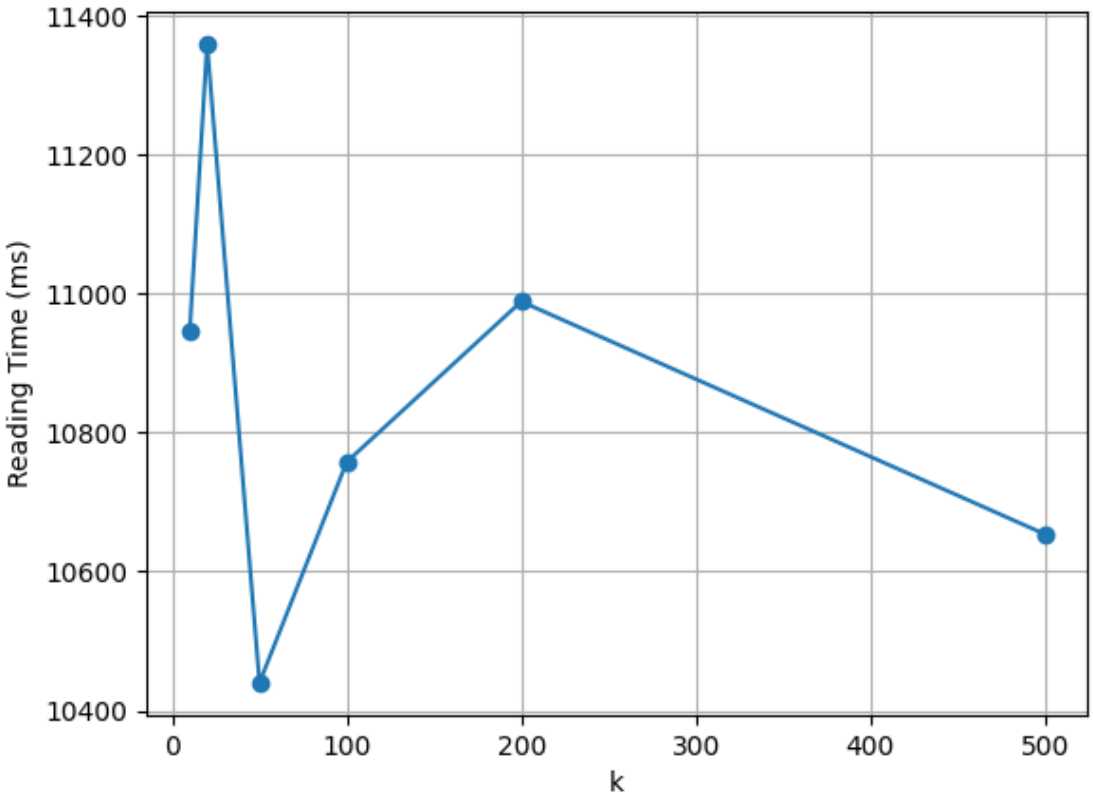


Vrijeme izvođenja u ovisnosti o veličini k-mera uz novu konfiguraciju.

Buckets = 64

Entries=8

Fingerprint=256

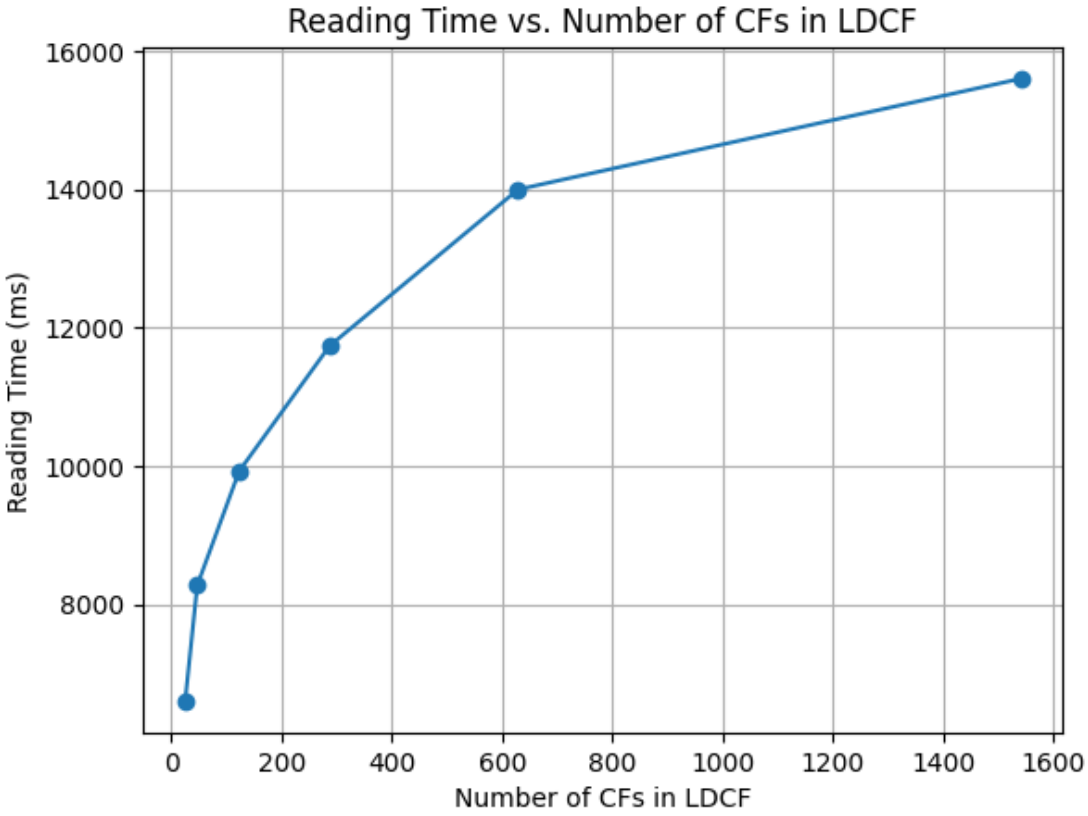


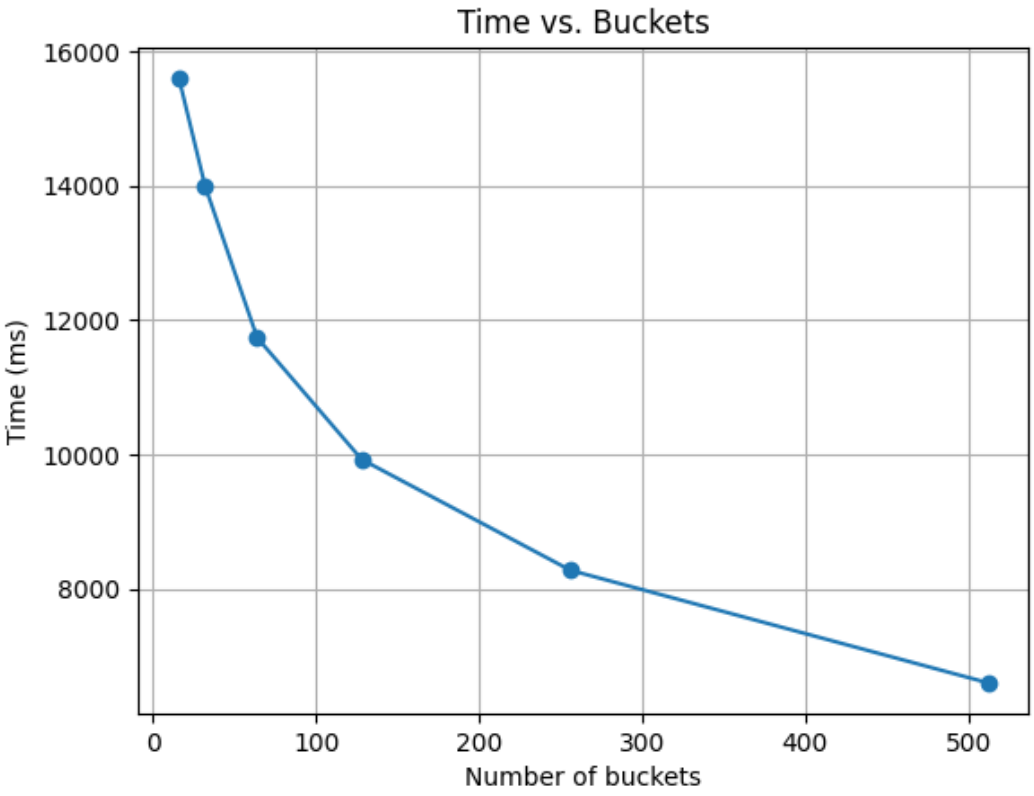
Slika 5 Vrijeme izvođenja u ovisnosti o veličini k-mera.

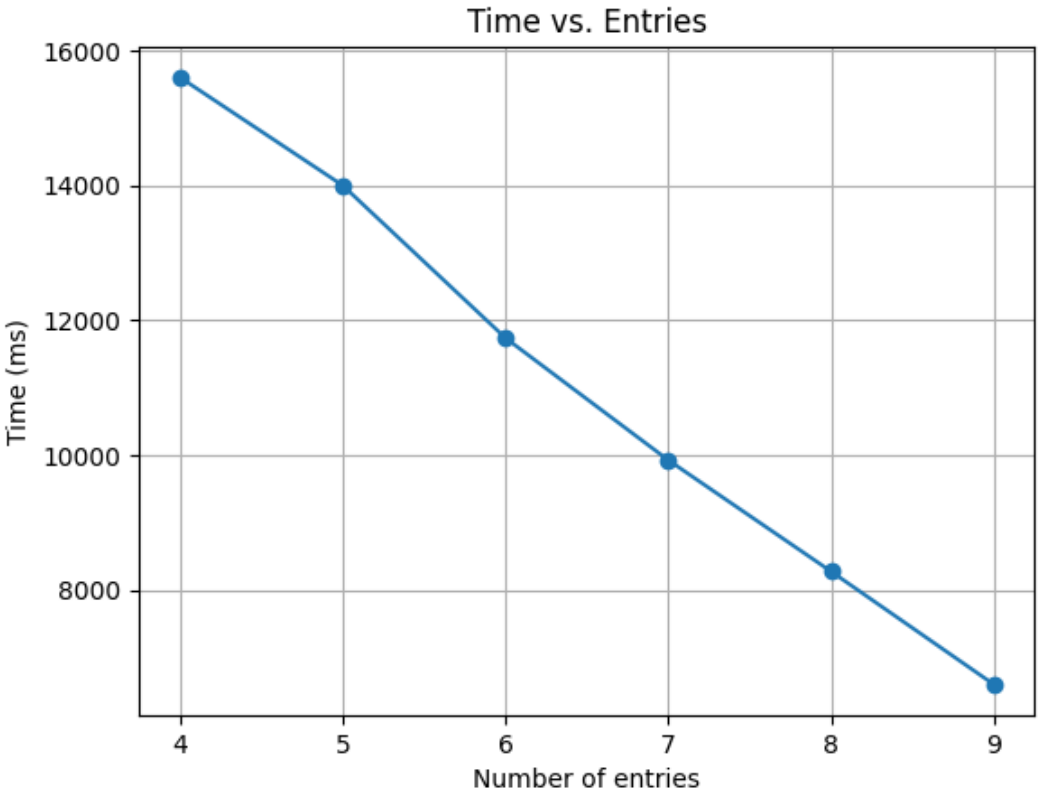
Vrijeme izvođenja u ovisnosti o broju bucketsa i entrija u LDCF-u.

Izvođene konfiguracije su uz fingerprint = 256 i k = 100 bile:

Buckets	Entries
16	4
32	5
64	6
128	7
256	8
512	9







Vidimo da se uz povećanjem bucketa i entrija vrijeme izvođenja se smanjuje.

I vrijeme izvođenja se povećava s brojem CF-a u LDCF-u.

Zaključak

Na temelju predloženog dinamičkog Cuckoo Filtera (DCF) koji ima ograničenja u pogledu troškova izračuna i prostornog zauzeća, Fan Zhang je predložila Logaritamski Dinamički Cuckoo Filter (LDCF). LDCF predstavlja napredniju strukturu podataka za organizaciju skupova s velikim dinamičkim podacima.

U usporedbi s DCF-om, LDCF ima značajno poboljšano vrijeme umetanja i testiranja elemenata te smanjeno prostorno zauzeće. Također predlaže kompaktni LDCF dizajn koji dodatno smanjuje složenost operacija na konstantnu razinu.

Implementacija algoritma za LDCF u našem radu je pokazala sporo vrijeme izvođenja. Međutim, LDCF ostaje obećavajuća struktura podataka koja može biti korisna u big data okruženjima.

S obzirom na prednosti LDCF-a u odnosu na postojeće filtre, kao što su Bloom filteri i Cuckoo filteri, LDCF ima potencijal za širu primjenu u industriji koja se bavi analizom i obradom velikih skupova podataka.

Literatura

obvezno navesti popis literature i navesti izvore unutar teksta (3 boda)

[1] Zhang et al. The Logarithmic Dynamic Cuckoo Filter doi: https://ieeexplore.ieee.org/document/9458864

[2] Chen et al. 2017. The dynamic cuckoo filter; https://ieeexplore.ieee.org/abstract/document/8117563

[3] Fan et al. 2013. Cuckoo Filter: Better Than Bloom; https://www.cs.cmu.edu/~binfan/papers/login_cuckoofilter.pdf

[4] GIRC (2018). Strain: Sakai substr. RIMD 0509952. E. Coli genome (RefSeq:GCF_000008865.2) https://www.ncbi.nlm.nih.gov/data-hub/taxonomy/562/

[5] Fan et al. 2014. Cuckoo Filter: Practically Better Than Bloom; http://www.cs.cmu.edu/%7Ebinfan/papers/conext14_cuckoofilter.pdf

[6] K-mer, Wikipedia https://en.wikipedia.org/wiki/K-mer

[7] Jellyfish https://github.com/gmarcais/Jellyfish

[8] https://bitesizebio.com/38250/de-novo-dna-sequencing-and-the-special-k-mer/