SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

SEMINAR

Vlastita implementacija filtera: The Logarithmic Dynamic Cuckoo Filter

Fran Ostroški, Elena Wachtler
Voditelj: *Mirjana Domazet-Lošo*

Zagreb, svibanj 2023.

SADRŽAJ

1. Uvod

Ovaj seminar opis je projekta u sklopu kolegija Bioinformatika 1 Fakulteta elektrotehnike i računarstva Sveučilišta u Zagrebu, a predstavlja vlastitu implementaciju takozvanog cuckoo filtera, točnije njegovu dinamičku logaritamsku varijantu.

Cuckoo filter probabilistička je struktura podataka koja se koristi za određivanje pripadnosti zadanog elementa nekome skupu. Sam filter nastao je kao proširenje već postojećih Bloom filtera, a naziv je dobio prema ptici kukavici koja izbacuje jaja iz tuđih gnijezda kako bi ubacila svoja. Iako je riječ o općenitim podatcima, cuckoo filter često se primjenjuje u bioinformatici jer je pogodan za određivanje pripadnosti podnizova nukleotidnih slijedova nekom većem nizu nukleotida ili provjeru nalaze li se podnizovi jednog slijeda u nekom drugom slijedu.

Pripadnost elementa nekom skupu ovom strukturom ne možemo garantirati, jer zbog implementacije algoritma postoji malena vjerojatnost za lažno pozitivne (eng. *false positive*) i lažno negativne (eng. *false negative*) rezultate. Upravo zbog toga što postoji određena vjerojatnost pogreške, cuckoo filter je probabilistički. Međutim, nepripadnost nekom skupu može se pouzdano odrediti.

2. Opis algoritma

Nastanak i evolucija Cuckoo filtera

Cuckoo filtere kao ideju za rješavanje problema koji zahtjevaju nisku stopu lažno pozitivnih primjera predstavili su Fan, Kaminsky i Andersen [3], kao strukturu čija je glavna prednost u odnosu na dotad korištene Bloom filtere sposobnost brisanja. Čak štoviše, uvode cuckoo filtere kao strukturu koja ne samo da omogućuje dinamičko dodavanje i brisanje objekata, već postiže bolje *lookup* performanse, pri tome koristeći manje prostora od dotad optimalnih Bloom filtera. Teoriju koju su predložili kasnije su i dokazali [4] - cuckoo filteri uistinu su nadmoćna struktura za pohranu velikog broja objekata, kada je nužan uvjet niska stopa lažno pozitivnih primjera. Uz to, dodana su i poboljšanja kako bi se smanjila prostorna složenost, i to djelomičnim sortiranjem bucketa te optimizacijom veličine bucketa - najboljim se pokazao (2, 4)-cuckoo filter, što znači da svaki objekt ima dva moguća kandidata za bucket, a svaki bucket u sebi sadrži podatke o četiri otiska (eng. *fingerprint*). O bucketima je više rečeno u nastavku poglavlja gdje je dan detaljan opis rada algoritma uz primjer.

Korak dalje učinili su Chen, Liao, Jin i Wu [2] uvođenjem dinamičkog cuckoo filtera, koji bi istovremeno zadovoljavao i uvjet da se veličina strukture može fleksibilno mijenjati, ali i da operacija brisanja bude pouzdana. Autori tvrde da ovakva implementacija cuckoo filtera koja u sebi sadrži povezanu listu filtera ostvaruje 75% smanjeno zauzeće memorije, poboljšanje u konstrukciji strukture od 50% te ubrzanje u operaciji *query* od čak 80%. Također, kao dodatnu optimizaciju uvode gornje ograničenje na broj relokacija. Relociranje je vizualizirano u nastavku poglavlja gdje je algoritam primijenjen na jednostavan primjer.

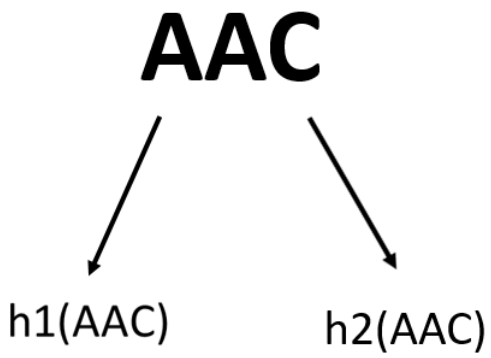
Posljednju nadogradnju ostvarili su Zhang, Chen, Jin i Reviriego [6] ističući linearno rastuću računalnu i prostornu složenost s rastom veličine prostora podataka kao glavni nedostatak dinamičkih cuckoo filtera, te uvodeći višerazinsko binarno stablo kao rješenje tog nedostatka, zbog čega se njihova struktura naziva logaritamskim dinamičkim cuckoo filterom, LDCF. S ovim poboljšanjem, operacije dodavanja novih

objekata i provjera nalazi li se objekt u strukturi (eng. *membership testing*) od složenosti *O(N)* (gdje je *N* veličina skupa podataka) padaju na složenost *O(1)*. Dodatno, ponudili su i varijantu kompaktnog LDCF-a u kojoj je dodatno smanjeno prostorno zauzeće, na način da se zadržava samo najviša razina cuckoo filtera u višerazinskom stablu. U ovome radu u poglavlju 3 analizirane su performanse vlastite implementacije upravo ove, najviše unaprijeđene varijante cuckoo filera - LDCF.

Vizualizacija algoritma na jednostavnom primjeru

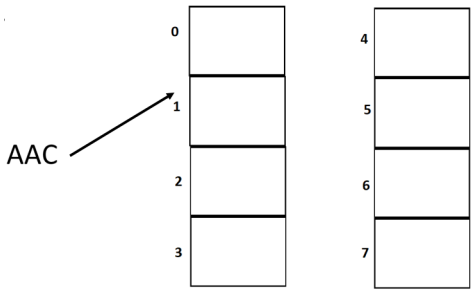
U općenitom slučaju, cuckoo filter sastoji se od nizova memorijskih lokacija koje se nazivaju *bucketima*. Bucketi mogu imati mjesta za više unosa, ali u ovom jednostavnom primjeru svaki bucket moći će primiti po jedan uneseni podatak.

Neka je dan jedan kratak nukleotidni slijed, npr. AACTGAT, te se kao ulazi u filter trebaju unijeti svi njegovi k-meri, za k = 3. To su : AAC, ACT, CTG, TGA i GAT .
U prvom koraku algoritma za svaki unos računaju se dva sažetka (eng. *hash*) pomoću odabranih funkcija sažimanja (tzv. *hash* funkcija).



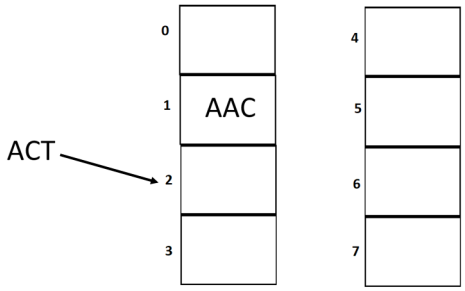
Slika 2.1: Generiranje hasha - Fran Ostroški

U drugom se koraku iz generiranih sažetaka određuje indeks bucketa u koji će biti smješten ulazni niz. To se najčešće određuje uporabom operacije dijeljenja *modulo* s ukupnim brojem bucketa. Kada se generiaju oba indeksa, element će se spremiti na jedno od dvije lokacije s tim indeksima. Svrha drugog sažetka je smanjenje vjerojatnosti kolizije i potrebe za premještanjem. Drugi sažetak ovisi i o trenutnoj poziciji elementa kako bi se osiguralo da se alternative razlikuju.



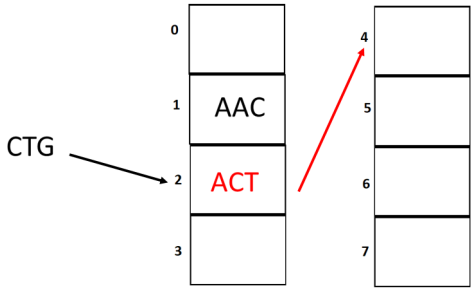
Slika 2.2: Odabir lokacije za niz AAC - prikaz po uzoru na [2]

Idući je na redu niz ACT. Nakon generiranja sažetaka smješta se u prazni bucket. Ovdje još uvijek nema kolizija.



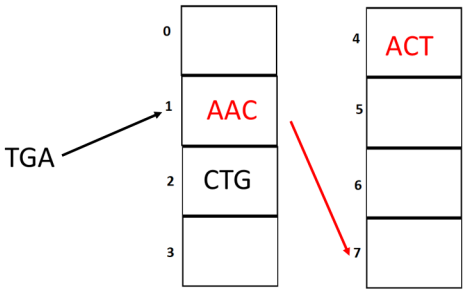
Slika 2.3: Odabir lokacije za niz ACT - prikaz po uzoru na [2]

Za niz CTG dogodila se situacija da je lokacija na koju ga želimo pohraniti već popunjena. Tada se u algoritmu događa "izbacivanje". Obabrani se element izbacuje, umeće se CTG, a za izbačeni element traži se nova lokacija.Novu lokaciju za izbačeni element dobivamo pomoću izračunat h2(CTG).



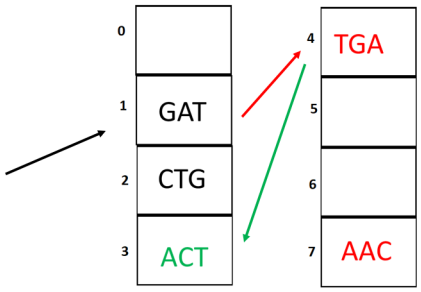
Slika 2.4: Odabir lokacije za niz CTG - prikaz po uzoru na [2]

Za niz TGA dogodila se ista situacija. Za element AAC pomoću h2(AAC) računa se alternativna adresa na koju se pohranjuje.

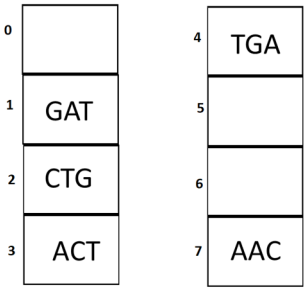


Slika 2.5: Izbacivanje AAC i ubacivanje TGA - prikaz po uzoru na [2]

Za niz GAT također se mora izvršiti izbacivanje starog elementa. Međutim, sada je i adresa na koju bi TGA bio premješten već popunjena. Tada se za element ACT ponovno računa nova pozicija koja zbog korištenja trenutnog indeksa u izračunu neće biti identična.



Slika 2.6: Izbacivanje TGA i ubacivanje GAT, relociranje ACT - prikaz po uzoru na [2]



Slika 2.7: Konačni razmještaj elemenata - prikaz po uzoru na [2]

Može se primijetiti da što je popunjenost veća, raste i vjerojatnost da će doći do izbacivanja i relokacije starijih elemenata u filteru. Dogodit će se situacije da će element koji se premješta biti premješten na već popunjeni bucket. U tom slučaju prijašnji element će se premjestiti, te će postupak nastaviti sve dok se ne pronađe slobodno mjesto

ili se dosegne maksimalan broj operacija premještanja. U slučaju da se dosegne maksimalan broj, to je obično indikator skore popunjenosti filtera. U tom slučaju mogu se poduzeti razne akcije, koje su ponovno različite za svaku implementaciju filtera.

Dodatak opisa rada LDCF

Algoritam LDCF započinje samo jednim CF-om, koji smatramo da je nulta razina budućeg višerazinskog stabla. Pri tome se element dodaje u CF tako da se njegov ranije spomenuti otisak (eng. *fingerprint*) dodaje u bucket. Jednom kada se dogodi da operacija *insert* ne uspije, smatra se da je CF pun, te se dodaju dva nova CF-a. Kada se dodaje novi objekt u jedan od dva nova CF-a, promatra se prvi bit njegova fingerprinta, te prema tome određuje hoće li biti smješten u lijevi novododani CF, za vrijednost prvog bita fingerprinta jednaku nula, ili desni novododani CF, za slučaj kada je prvi bit njegova fingerprinta jednak jedan. Budući da je time vrijednost prvog bita implicitno zapisana u poziciji elementa u stablu, taj se bit uklanja iz fingerprinta i fingerprint se pohranjuje bez njega. Taj se postupak dalje propagira na niže razine.

Jedna od glavnih prednosti algoritma LDCF nad DCF jest velika mogućnost paralelizacije. Naime, u DCF-u sve operacije *insert* izvode se na istom CF-u, dok su u LDCF-u grane s CF-ima međusobno neovisne zbog opisanog načina kako se konstruiraju, te se zbog toga operacije *insert* mogu izvoditi istovremeno na neovisnim granama.

O implementaciji

Za implementaciju je korišten programski jezik C++. Ulazni podatci učitavaju se iz datoteke i rezultati se ispisuju u vanjsku izlaznu datoteku. Što se tiče vanjskih knjižnica, korišten je OpenSSL, knjižnica koja je uključena u bio-linux.

Prijašnji radovi [5] sugeriraju da je četiri unosa po bucketu optimalna veličina bucketa, ali u ovoj je konkretnoj implementaciji odabrana konfiguracija s bucketom koji prihvaća maksimalno osam unosa. Tesitiranjem različitih varijacija, zaključeno je da je za takvu implementaciju optimalna manja veličina cuckoo filtera od onih koje predlažu drugi radovi - manji broj bucketa po cuckoo filteru - konkretno, u testovima navedenima u analizi, korišteni su cuckoo filteri koji sadrže šesnaest bucketa.

3. Analiza

U tekstu koji slijedi dana je analiza točnosti, vremena izvođenja i utroška memorije za različite testne slučajeve. Testovi su izvođeni u nekoliko inačica, koristeći:

- sintetske podatke - nizovi duljine od 10^3 do 10^7 znakova
- stvarne podatke - genom bakterije Escherichia coli.

Analiza na sintetskim i stvarnim podatcima

Za analizu na stvarnim podatcima korišten je genom bakterije Esherichia coli. Ova je bakterija modelni organizam te se njezin genom u bioinformatici koristi vrlo često i do danas je u potpunosti sekvenciran za velik dio njezinih sojeva. Konkretno, u nastavku su prikazani rezultati primjene algoritma na genom soja GCA 000798515.1, preuzetog iz baze podataka Ensembl Bacteria [1].

Prvi je test proveden na način da je u ulaznu datoteku upisan nukleotidni slijed duljine 100 nukleotida, uzet iz stvarnog genoma ranije spomenutog soja bakterije Escherichia coli. *K*, veličina k-mera, postavljen je na duljinu od pet nukleotida te su najprije pozvane metode *insert* nad LDCF-om za sve k-mere učitane iz ulazne datoteke. Budući da je slijed duljine 1000, a k-mer duljine 5, broj svih k-mera u slijedu je razlika duljine slijeda i duljine k-mera uvećana za jedan, dakle 96. Nakon dodavanja svih k-mera u LDCF (točnije bi bilo reći da se poziva metoda *insert* nad svim k-merima, ali budući da su svi uspješno dodani, jednostavno je rečeno da su dodani svi k-meri), ti su isti k-meri potraženi (metoda *query*) u generiranoj strukturi LDCF. Kako je i očekivano, svi su k-meri pronađeni u LDCF-u, što znači da nije bilo *false negative* rezultata. To je jedan od najvažnijih kriterija rada LDCF-a. Nakon što je potvrđeno da nema lažno negativnih upita, generirani su sintetski podatci na način da su nasumično odabrani nukleotidi tako da se konačno generira niz od 100 nukleotida. Ti su sintetski podatci potom potraženi u generiranom LDCF-u. Od njih 96, njih je 10 pronađeno u LDCF-u. Ručnom provjerom dokazano je da se svih 10 uistinu nalazilo u stvarnom genomu iz kojega je generiran LDCF, što pokazuje da implementacija radi na malenim skupovima.

Na sličan način provedeni su i ostali testovi, koji su zbog jednostavnosti i preglednosti prikazani tablično.

Tablica 3.1: Rezultati testiranja na stvarnim i sintetskim podatcima

r. br. testa	dulj. ul. nuk. slijeda *(10^x)*	duljina k-mera	vrijeme izvođenja operacije insert *(s)*	vrijeme izvođenja operacije query *(s)*
1	3	10	0.1139	0.0893
2	3	25	0.0932	0.0900
3	3	50	0.0962	0.0878
4	3	100	0.1141	0.0869
5	3	200	0.1013	0.0761
6	3.698	5	0.7838	0.4435
7	3.698	20	3e-06	0.2798
8	3.698	50	2e-06	0.2936
9	3.698	100	1e-06	0.2060
10	3.698	200	2e-06	0.2021
11	4.398	5	7.7431	2.2935
12	4.398	20	3.12	3.87
13	4.398	50	3.06	3.84
11	4.699	5	23.2000	4.6064
12	4.699	50	28.6508	55.4289
11	5.398	5	23.2000	4.6064

Napomena: Podatci su odvojeni prema duljini nukleotidnog slijeda, a potom prema duljini k-mera. Kategorija vremena izvođenja izračunata je preko biblioteke ctime.

Zauzeće memorije procjenjuje se prema broju zauzetih cuckoo filtera, kao i broju bucketa koji svaki cuckoo filter sadrži, i to prema sljedećoj formuli:

(((floor(Bajtovi inputa / (bucketsize * singletablelength)) +1) * bucketsize * singletablelength) / 1024) 1024 (MB)

4. Zaključak

Logaritamski dinamički cuckoo filter kao struktura uistinu je nadmoćna nad svojim ranijim varijantama. Ne samo da omogućuje efikasno pohranjivanje novih unosa, već i njihovo uklanjanje iz strukture LDCF, što ju znatno ističe nad ranijim strukturama korištenima u bioinformatici, poput Bloom filtera. Vlastita implementacija ostvaruje zadovoljavajuće rezultate kako na stvarnim, tako i na sintetskim podatcima. Uspješno pohranjuje sljedove duge do testiranih milijun nukleotida dugih sljedova,

5. Literatura

[1] Ensembl bacteria. `http://bacteria.ensembl.org/index.html`. Pristupljeno: svibanj, 2023.

[2] Hanhua Chen, Liangyi Liao, Hai Jin, i Jie Wu. The dynamic cuckoo filter. U *2017 IEEE 25th International Conference on Network Protocols (ICNP)*, stranice 1–10, 2017. doi: 10.1109/ICNP.2017.8117563.

[3] Bin Fan, David G. Andersen, i Michael Kaminsky. Cuckoo filter: Better than bloom. *login Usenix Mag.*, 38(4), 2013.

[4] Bin Fan, Dave G. Andersen, Michael Kaminsky, i Michael D. Mitzenmacher. Cuckoo filter: Practically better than bloom. U *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, stranica 75–88, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450332798. doi: 10.1145/2674005.2674994.

[5] Pedro Reviriego i Salvatore Pontarelli. Perfect cuckoo filters. U *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '21, stranica 205–211, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450390989. doi: 10.1145/3485983.3494852.

[6] Fan Zhang, Hanhua Chen, Hai Jin, i Pedro Reviriego. The logarithmic dynamic cuckoo filter. U *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, stranice 948–959, 2021. doi: 10.1109/ICDE51399.2021.00087.

6. Sažetak

U ovome radu dan je pregled vlastite implementacije rješenja problema opisanog u znanstvenim radovima koja je napravljena u sklopu projekta iz kolegija Bioinformatika 1 na Fakultetu elektrotehnike i računarstva Sveučilišta u Zagrebu. Dan je opis algoritma, motivacija za njegovo korištenje, kao i primjer primjene na odabrani nukleotidni slijed. Nadalje, analiziran je rad algoritma u ovoj implementaciji u odnosu na dosadašnje implementacije, njegove prednosti i nedostatci.