SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

PROJEKT

Faza konsenzusa u OLC paradigmi sastavljanja genoma – Sparc

Ana Brassard, Tin Kuculo

Zagreb, Siječanj, 2018

Sadržaj

1. Uvod

Treća generacija tehnologije za sekvenciranje genoma u relativno malo vremena može sekvencirati vrlo duge nizove, no ima visoku razinu pogreška (15-40%). Kako bi se dobilo točne genetske sekvence, potrebno je mnogo puta iznova sekvencirati isto područje genoma te uz pomoć tih dodatnih očitanja smanjiti grešku. Tu se javlja potreba za algoritmom koji će sva ta očitanja usporediti, poredati i spojiti kako bi generirao jedinstven genetski niz s visokom razinom točnosti. Jedan od dostupnih algoritma nalazimo u OLC paradigmi, odnosno *Overlap*, *Layout* i *Consensus* (preklapanje, poravnavanje, konsenzus). Prve dvije faze, preklapanje i poravnavanje, generiraju niz poravnatih očitanja pomoću kojih konsenzus faza generira konačni ispravljeni niz.

Ovaj rad opisuje Sparc algoritam koji vrši fazu konsenzusa u OLC paradigmi i njegovu implementaciju.

2. Sparc algoritam

Sparc algoritam [1] inspiriran je de Brujinovim grafovima [2], te predstavlja jedno njihovo pojednostavljenje razvijeno za poboljšanje kvalitete dugačkih očitanja.

Najjednostavnije opisano, algoritam isprva gradi k-mer grafove od kontiga koje želimo ispraviti te iterativno proširuje graf pomoću mapiranih očitanja na kontige. Naiđe li na potpuno slaganje između osnovong grafa i novog očitanja, tim prijelazima poveća težinu. Na kraju pretražuje prijelaze između početnog i krajnjeg čvora grafa te nalazi onaj put koji ima najveću težinu: taj put odgovara sekvenci s najvećim slaganjem, odnosno vjerojatno najtočnija.

2.1 Početni parametri

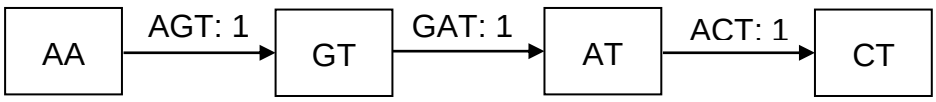
K-mer graf je aciklični usmjereni graf koji ima svoje čvorove i prijelaze. Način gradnje početnog grafa ovisi o parametrima **k** i **g**. Parametar **k** određuje koliko baza zapisujemo u čvorovima, a **g** koliko na prijelazima. Nova očitanja poravnat ćemo s odgovarajućim čvorovima. Parametar g=1 rezultira time da provjeravamo sve čvorove, a uz veći g će biti brže izvođenje ali i manje točan rezultat jer uspoređujemo manje čvorova. Autori algoritma preporučuju odabir sljedećih vrijednosti:

$$k \in [1, 3], g \in [1, 5].$$

2.2 Gradnja početnog grafa

Početni graf gradimo od kontiga tako što grupiramo svakih k baza udaljene g. Primjer izgrađenog grafa vidimo na Slici 1.

Osnovni niz: **AAAGTGATACT**, uz k=2 i g=3



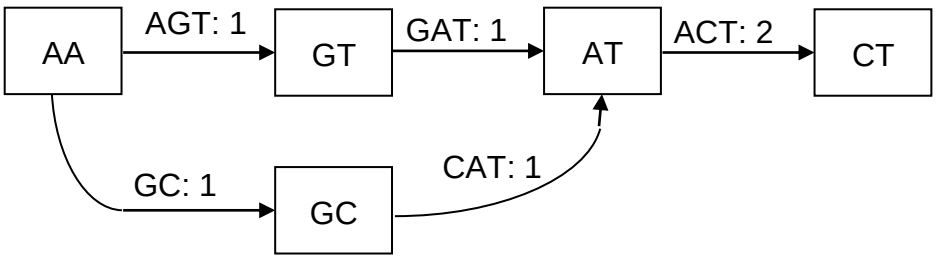
Slika 1. Primjer izgrađenog osnovnog grafa

2.3 Izgradnja ostatka grafa

Nakon toga dodajemo očitanja poravnata na toj poziciji. Kada naiđemo na prijelaz koji već postoji, tom prijelazu povećamo težinu. U protivnom stvaramo novi prijelaz i novi čvor, te nastavljamo poravnavanje od tog čvora. Na Slici 2 je ilustriran primjer dodavanja novog očitanja počevši od korijenskog čvora. Stil prikaza grafa inspiriran je primjerom iz [1].

Osnovni niz: AAAGTGATACT

Novi niz: **AA_GCCATACT**



Slika 2. Demonstracija dodavanja novog očitanja u graf

Ovaj se postupak ponavlja za sva očitanja dok se ne izgradi konačan graf. Na kraju se prođe od početka do kraja grafa kako bi se našao najteži put od početnog do krajnjeg čvora. Taj put odgovara onom putu oko kojih se očitanja najviše „slažu“, dakle najveća vjerojatnost da je to točna sekvenca.

2.4 Neki detalji bitni za implementaciju

Umetanje očitanja nije tako jednostavno kao što se čini u osnovnom primjeru.

Očitanje ne počinje uvijek točno tako da prvih k baza odgovara čvoru na poziciji na kojoj je očitanje poravnato. U tom se slučaju odbacuje početne baze viška i traži odgovarajući čvor (ili stvara novi) na poziciji idućeg „sloja“ čvorova.

Dodatno, očitanja mogu imati 2 vrste razlika u odnosu na referentni niz: umetanje i brisanje. U slučaju umetanja, stvara se prijelaz s više od g znakova, to jest točno onoliko znakova koliko je potrebno kako bi se stvorio niz od g znakova ne računajući one umetnute. Dogodi li se brisanje, stvaraju se prijelazi kraći od g znakova, pa čak i prazni prijelazi ako je potrebno. U čvorovima, brisanja označavamo donjom povlakom: _ .

Također treba uzeti u obzir način na koji očitanje završava – ni kraj ne mora točno odgovarati nekom prijelazu kao što je to odgovaralo u primjeru na Slici 2. Ovdje se uzima u obzir par iznimnih slučaja. Prvi slučaj je da očitanje završava čvorom koji prije umetanja nije postojao. Tada taj čvor valja povezati natrag na stablo, kako ne bi postojao put koji prerano završava. To se postiže stvaranjem dodatnog prijelaza s praznim nizom koji povezuje zadnjeg stvorenog čvora i originalnog čvora na istom indeksu. Drugi slučaj je da na kraju ne ostane dovoljno znakova za cijeli novi prijelaz. Tada se odbacuje te posljednje znakove i povezuje se zadnji čvor na osnovni kao u prvom slučaju.

Moguće je i pojavljivanje očitanja koje počinju na nultom indeksu, to jest da bi trebali odgovarati korijenskom čvoru, a imaju različite znakove od trenutnog korijena stabla. Radi rješavanja ovakvih slučajeva, uveden je dodatni čvor prije korijenskog čvora s

negativnim indeksom i bez znakova. Taj novi čvor sada je korijen stabla, te iz njega mogu polaziti različiti prijelazi koji predstavljaju različite moguće početke ispravljenog niza. U istom duhu je dodan i novi krajnji čvor, kako bi se osiguralo da svi prijelazi i čvorovi na kraju vode u jedinstveni krajnji čvor.

Na kraju, postoje varijacije načina na koji se može dodati težina na određene čvorove. Najosnovniji način je jednostavno brojanje koliko prijelaza se tu složilo. S druge strane, s obzirom da je dostupan podatak o tome koliko je sigurno očitanje, moguće je uzimanje u obzir i toga kako bi se dobilo što točniji rezultat. U ovom radu je kao težina korištena srednja vrijednost sigurnosti znakova od kojih se sastoji prijelaz.

3. Implementacija

Projekt je implementiran u C++-u i ne koristi nikakve dodatne pakete osim onih uvedenih na početku u *include* bloku. Kod je rastavljen na 4 datoteke: Sparc.cpp, Utils.h, Dana.h i Node.h. U prvoj se nalazi glavni program, a u ostalim razne pomoćne funkcije i klase pomoću kojih se gradi graf.

3.1 Struktura podataka

Graf je izgrađen pomoću klasa „Node“ i „Edge“, koje predstavljaju čvorove i prijelaze. Svaki čvor u sebi ima pokazivač na niz znakova, indeks koji predstavlja položaj čvora u odnosu na korijenski čvor i pokazivač na listu prijelaza. Prijelaz ima pokazivač na niz znakova, težinu te pokazivač na čvor koji mu je idući, to jest, onaj na koji se povezuje.

Također postoji i klasa „Data“ koja sadrži mehanizme čitanja i pripremanja podataka za obradu.

3.2 Inicijalizacija

Nakon što se dohvati podatke, stvara se početni k-mer graf koji se dobije tako što se jednostavno svakih g nizova duljine k pretvori u čvor, a nizovi između u prijelaz, kao što je to prethodno opisano. Tada efektivno imamo povezanu listu čvorova, samo što prijelazi još imaju neke svoje podatke.

3.3 Dodavanje očitanja

Dodavanje čvorova radi se tako što se iterira kroz moguće indekse na kojima bi moglo biti očitanje, te ako na tom indeksu postoji očitanje, ulazi u petlju dodavanja očitanja na trenutnom indeksu. U protivnom povećava indeks i ponovo provjerava očitanja. Tijekom iteriranja se pamti liste čvorova koji se nalaze razinu ispod i razinu nakon trenutnog indeksa (prethodni i trenutni čvorovi). Petlja dodavanja iterira kroz očitanja na trenutnom indeksu te za svaki indeks vrši sljedeća tri koraka:

1. Ako je potrebno, podreži početak očitanja (da bude poredano uz čvor na razini idućih indeksa)
2. Potraži postoji li već početni čvor među trenutnim čvorovima
    a. POSTOJI: pozovi funkciju dodavanja podniza očitanja koji ne uključuje dio koji odgovara nađenom čvoru, trenutnim čvorom odakle počinje dodavanje (to jest, ovaj nađeni čvor), skupom trenutnih čvorova i nizom kvalitete očitanja
    b. NE POSTOJI: stvori novi čvor na razini na kojoj bi trebao biti, poveži ga s prethodnim čvorom, pa pozovi funkciju kao pod a.

Funkcija dodavanja za dani niz, počevši od danog čvora, ubacuje očitanje kroz tri faze:

1. Stvori sve čvorove i prijelaze koji će uvesti novi niz (bez obzira postoji li već neki čvor ili ne), te ih zapiši u mape. Ključ mape je par koji se sastoji od niza čvora i indeksa, a vrijednost je pokazivač na čvor koji ima taj niz i taj indeks.
2. Iteriraj kroz stablo razinu po razinu, počevši od datog niza trenutnih čvorova. Ako se naiđe na čvor koji je stvoren u 1., zamijeni pokazivač čvora u listi s pokazivačem na trenutni čvor. Ovdje osiguramo da ne stvaramo nove čvorove ako oni već postoje.
3. Ako je potrebno, dodaj prazni prijelaz na kraju prijelaza koji pokazuje natrag na čvor osnovnog niza (zatvaramo put)
4. Iteriraj kroz listu čvorova i prijelaza iz 1. i 2., svaki čvor spojiti s idućim pomoću odgovarajućeg prijelaza

3.4 Nalaženje najtežeg puta

Sada kada je graf potpuno izgrađen, može se pozvati funkcija nalaženja najtežeg puta. Najteži put nalazi se na način da se za svaku razinu stabla zapisuje najteži put do svakog čvora u idućoj razini. Kada se prelazi na novu razinu, zaboravlja se prethodne najteže puteve i zapisuje samo nove puteve koji uključuju novu trenutnu razinu. Funkcija također dodaje novi put do razmatranog čvora samo ako put do njega već nije nađen, ili ako je nađen, da je trenutni put veće težine od prethodno nađenom puta. Na kraju, kada se dođe do zadnje razine stabla, odnosno do zadnjeg čvora, u listi nađenih puteva ostaje samo baš onaj put od početka grafa do kraja koji je najteži. Taj niz zapisujemo u izlaznu datoteku.

4. Rezultati

Algoritam implementiran onako kako je dosad opisano, nažalost, ne daje zadovoljavajuće rezultate uz veliko testno očitanje (genom Escherichia Coli), jer znatno prelazi dano vremensko ograničenje. Ipak, predstavljamo neke testove uz kraći niz za koje daje relativno dobre rezultate u Tablici 1.

Mjereno vrijeme nije realno vrijeme već CPU vrijeme koje se mjeri od prvog pokretanja programa do trenutka kada je gotovo zapisivanje rezultata u izlaznu datoteku.

Točnost se procjenjuje pomoću alata dnadiff koji je dio paketa MUMmer [3].

Sva pokretanja koriste oko 10GB radne memorije, no to je jednostavno maksimalno ograničenje virtualnog stroja unutar kojeg se pokreće i nije nužno da zaista koristi toliko.

Test	**Dužina osnovnog niza**	**Vrijeme izvođenja**	**Točnost**
Originalni autori: skup E. coli - **k=1, g=1**	10600kB	30s	98%
Skup „lambda“ – **k=2, g=5**	48kB	27s	89.86%
Skup „lambda“ – **k=2, g=3**	48kB	37s	92.32%
Skup „lambda“ – **k=1, g=1**	48kB	47s	93.72%
Skup E. Coli – **k=2, g=3**	4800kB	>2h	-

Tablica 1. Prikaz rezultata provedenih testova

5. Zaključak

Implementiranje bioinformatičkih algoritama vrlo je izazovan zadatak zbog svoje osjetljivosti na efikasnost. Nije dovoljno implementirati algoritam kako je opisan na apstraktnoj razini, već se mora i prilagoditi implementaciju da bi izvođenje bilo što brže, te memorijska potrošnja što manja. Svako malo poboljšanje dolazi do izražaja budući da su podaci nad kojim se radi ogromni nizovi znakova koji sami mogu biti veliki nekoliko stotina megabajta, pa čak i u gigabajtima. U ovom radu opisan je pokušaj takve jedne implementacije koja uspješno izvršava algoritam, no nije dovoljno efikasan. Da bi implementacija mogla biti konkurentna trenutno postojećim, potrebno je još intenzivnog rada nad njom kako bi se identificirala područja uskog grla i ispravile performanse.

6. Literatura

[1] M. Z. Ye C, »Sparc: a sparsity-based consensus algorithm for long erroneous sequencing reads,« *PeerJ,* br. 4:e2016, 2016.

[2] B. E. V. Zerbino DR, »Algorithms for de novo short read assembly using de Bruijn graphs,« *Genome Research,* svez. 18(5), pp. 821-829, 2008.

[3] S. Kurtz, A. Phillippy, A. L. Delcher, M. Smoot, M. Shumway, C. Antonescu i S. S. L, » Versatile and open software for comparing large genomes,« *Genome Biology,* br. 5:R12, 2004.