SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 639

Otkrivanje preklapajućih DNA očitanja

Matija Osrečki

Zagreb, veljača 2014.

Umjesto ove stranice umetnite izvornik Vašeg rada.
Da bi ste uklonili ovu stranicu obrišite naredbu `\izvornik`.

SADRŽAJ

1. Uvod

De novo sastavljanje (engl. *de novo assembly*) genoma jedan je od ključnih problema u području bioinformatike i genomike općenito. Sad kada nove tehnologije paralelnog *sekvenciranja* (engl. *shotgun sequencing*) omogućavaju pristup velikim količinama novih, nesastavljenih DNA sekvenci (očitanja), zahtjev je sve veći na razvoju novih, efikasnijih metoda sastavljanja. Uz takav razvoj, nije pretjerano očekivati da će u skorom vremenu biti moguće za pojedinog individualca da ima pristup svojem genomu, što će doprinijeti kvalitetnijim zdravstvenim metodama koje se temelje na genomici.

Sastavljanje genoma problem je u kojem treba veliki skup segmenata DNA spojiti u što kraći i kvalitetniji niz DNA koji će ih sve sadržavati. Problematika koja se javlja jest primarno količina podataka, te količina greške koju sekvenciranje uvede. I ako zanemarimo problem greški u sekvencama, svejedno imamo NP-potpuni problem, a uvođenjem greški problem rezultira dodatnom dimenzijom težine.

Postoji više tehnologija sekvenciranja, od kojih su neke od najpoznatijih *PacBio*, *Sanger*, *Illumina* i *454*. Međusobno se razlikuju po brzini i količini sekvencirane DNA, ali najbitnije prosječnoj duljini očitanja i stupnju pogreške. PacBio je jedna od novijih metoda koja trenutno generira najdulja očitanja prosječne duljine par tisuća, ali i uvodi najveći stupanj pogreške – od 15% do 18%. Sastavljanje dugih Pacbio očitanja je namjera ovog rada.

Dva osnovna pristupa su izgradnja *de Bruijn grafa* koji se temelji na *k-merima* i koja je prikladnija za kraće sekvence, kako zbog činjenice da se informacije izgube prilikom razbijanja na k-mere, kako zbog velikog memorijskog opterećenja takvog pristupa. Drugi pristup jest izgradnja *grafa sekvenci*, odnosno potrebno je naći sva preklapanja između očitanja, kako bi se izgradio graf koji se u konačnici transformira u sastavljen genom. Upravo taj korak traženja preklapajućih DNA očitanja je fokus ovog rada.

2. Opis zadatka

2.1. Osnovne definicije

Neka $\mathcal{R} = \{R_1, R_2, \ldots, R_N\}$ označava skup DNA očitanja, a $N = |\mathcal{R}|$ veličinu skupa $\mathcal{R}$. Ukupnu količinu podataka označavamo s $M = N + \sum_i |R_i|$. Sumi dodajemo N jer je potreban dodatan znak terminacije za svako očitanje.

$|R|$ jest duljina, dok je $R[i]$ znak na mjestu i očitanja R. $\overline{R} = R[|R|] \ldots R[2]R[1]$ je obrnuto očitanje R. Osim toga, neka je $R[i, j] = R[i]R[i + 1] \ldots R[j]$ podniz očitanja R. *Sufiks* $R[i] \ldots R[|R|]$ označavamo sa $suf_i R$, a *prefiks* $R[1] \ldots R[i]$ sa $pre_i R$.

Obrnuti komplement (o.k.) R' očitanja R jest transformacija očitanja koja predstavlja isto očitanje dobiveno sa parnog lanca u suprotnom smjeru. Tako je obrnuti komplement očitanja `AGTTGGCT` upravo `AGCCAACT` (parni nukleotidi su `A-T`, te `C-G`). Na isti način definiramo skup obrnuto komplementiranih DNA očitanja $\mathcal{R}' = \{R'_1, R'_2, \ldots, R'_N\}$.

2.2. Problem

Zadan je skup očitanja $\mathcal{R}$. Potrebno je odrediti sve parove očitanja koji se preklapaju, tj. takve parove da je sufiks jednog od očitanja približno jednak prefiksu drugog, koristeći određenu mjeru sličnosti, npr. *Levenshtein distance* (kasnije opisan). Dodatan zahtjev je da veličina preklapanja mora zadovoljavati neku zadanu minimalnu duljinu t.

Treba imati na umu da očitanja mogu biti u oba smjera, što znači da treba generirati skup obrnuto komplementiranih očitanja $\mathcal{R}'$ te uspoređivati očitanja između ta dva skupa. Konkretnije, treba naći preklapanja sufiksa očitanja iz skupa $\mathcal{R}$ sa prefiksima očitanja iz skupa $\mathcal{R}'$ te obrnuto.

Osim toga, moguće je da će velik broj očitanja biti u potpunosti sadržano u drugim, većim očitavanjima te se ona mogu opcionalno izbaciti ili bar odvojiti od glavnog

skupa očitanja, pošto nisu potrebna u svim daljnjim fazama rekonstrukcije genoma te mogu samo usporiti izvođenje tih faza.

Nad parom očitanja (i, j) sa duljinama preklapanja (k, l) takvim da $k, l \geq t$ formalno definiramo tri moguće vrste preklapanja:

EB (kraj – početak):

Sufiks duljine k očitanja R_i se preklapa sa prefiksom duljine l očitanja R_j.

EE (kraj – kraj):

Sufiks duljine k očitanja R_i se preklapa sa prefiksom duljine l očitanja R'_j.

BB (početak – početak):

Sufiks duljine k očitanja R'_i se preklapa sa prefiksom duljine l očitanja R_j.

Nad svakim parom očitanja treba odrediti najviše jedno, najbolje preklapanje, ako dovoljno dobro preklapanje uopće postoji. Oznake vrste preklapanja označavaju koji dio originalnog očitanja se preklapa s drugim očitanjem.

3. Metode

3.1. Pregled

Dva su glavna problema u oblikovanju algoritma za traženje preklapanja.

Količina podataka. Ovisno o mjeri pokrivenosti genoma, podataka je više ili manje, ali primjerice za ljudski genom je veličine oko 4 milijarde, što za tipičnu mjeru preklapanja $10\times$ implicira od 40 do 50 GB podataka ulaznih podataka. Za takav genom više nije ni moguće indeksirati podatke sa 32-bitnim cjelobrojnim tipom, što znači da za izgradnju sufiksnog polja treba najmanje $9M$ bajtova umjesto $5M$, odnosno oko 400 GB memorije. Vremenska složenost je još veći problem te je algoritam potrebno oblikovati sa vremenskom složenosti $O(M)$.

Mjera greške. Postoje nekoliko ključnih tehnologija za DNA očitavanja i ključne razlike su distribucija duljina očitanja i mjera greške. Pacbio je jedna od novijih tehnologija koja generira duga očitanja sa velikom pogreškom – do 18%. Tolika mjera greške stvara razliku između jednostavnog problema i nemogućeg problema, što znači da smo prisiljeni na metode aproksimacije ili heuristike.

Algoritam traženja preklapanja odvija se u sljedećim koracima:

1. **Unos očitanja**. Očitanja se učitavaju iz FASTA datoteke, izbacuju se prekratka očitanja, te se generiraju obrnuto komplementirana očitanja i dodaju u isti skup.

2. **Izgradnja FM-indeksa**. Najprije treba pomoću sufiksnog polja izgraditi Burrows-Wheeler transformacija ulaznih podataka. Zatim treba sortirati sva očitanja te iznad BWT-a izgraditi odgovarajuće strukture podataka uz koje možemo tražiti proizvoljne podnizove jako efikasno.

3. **Određivanje kandidata preklapanja**. Egzaktne pretrage nisu dovoljne zbog visokog stupnja pogreški. Sa sufiksnim filtrom moguće je odrediti skup potencijalnih kandidata preklapanja koji sadržavaju određenu količinu lažnih pozitiva.

4. **Validacija kandidata**. Prethodno generirane kandidate treba validirati, koristeći neku od metoda poravnavanja (engl. *alignment*) dijelova očitanja.

U nastavku slijede detaljniji opisi pojedinih koraka.

3.2. Predobrada

Nakon što učitamo sva očitanja, prvi korak je eliminacija identičnih očitanja. Iako nam takva očitanja mogu koristiti u koraku izgradnje konsenzusa (koji nije dio ovog rada) za provjeru prosječne pokrivenosti dobivenog genoma, za sve ostale korake nisu potrebni. Isto vrijedi za očitanja koja su sadržana u nekom drugom očitanju, iako je takva nešto teže eliminirati ako se uzme u obzir da jedno očitanje može sadržavati drugo sa nekom mjerom pogreške.

Drugi korak predobrade je transformacija znakova iz skupa $\{A, C, G, T\}$ u cijele brojeve na intervalu $[1, 4]$, tim poretkom. To omogućuje direktno indeksiranje raznih pomoćnih polja pomoću vrijednosti znaka, kao i jednostavnu kalkulaciju parnog nukleotida oduzimanjem od 5. Također je pretpostavka da svako očitanje završava sa znakom terminacije `$`, koji ima vrijednost 0. Tako se znakovni niz `ACGTAT$` transformira u `1234140`. Konačno, na taj način je moguće 2 znaka spremiti u jedan bajt.

Zadnji korak je dodavanje obrnuto komplementarnih očitanja u skup. Iako se semantički to može razmatrati kao odvojen skup, kao u definiciji problema, za potrebe programske izvedbe je ovako bolje jer treba provjeravati kandidate preklapanja samo dvaput, dok bi onako bilo triput. Na taj način će se ista preklapanja dvaput pojaviti, ali se lako uklone u zadnjoj fazi i uostalom ne predstavlja usko grlo algoritma.

3.3. FM-indeks

Bilo greški u očitavanjima ili ne (a na našu nesreću ih je previše), potrebna nam je struktura podataka koja će za proizvoljan niz znakova moći odrediti sva mjesta pojavljivanja istog u danom skupu očitanja. Ako je l veličina niza koji tražimo, a o broj pojavljivanja tog niza, FM-indeks nam omogućuje da ih nađemo u vremenskoj složenosti $O(l + o)$, uz određene pretpostavke. To znači da je pretraživanje neovisno od veličine korpusa, za razliku od algoritma Knuth-Morris-Pratt za istu namjenu koji je vremenske složenosti $O(l + m)$, gdje m označuje veličinu korpusa. No treba imati na umu da za KMP ne treba indeksirati korpus prije uporabe.

3.3.1. Burrows-Wheeler transformacija

Burrows-Wheeler transformacija (u nastavku BWT) je usko povezana sa sufiksnim poljem, i iako se može izgraditi nezavisno od istog, mi koristimo sufiksno polje kako bi dobili BWT skupa očitanja. U tu svrhu će sufiksno polje najprije biti objašnjeno.

Sufiksno polje jest ništa više od sortiranog niza svih sufiksa nekog niza znakova. Za niz znakova S duljine n (uključujući `$`) formalno definiramo sufiksno polje SA kao niz duljine n indeksa sa svojstvom da vrijedi $i < j$ ako i samo ako je $suf_{SA[i]}S$ leksikografski manji od $suf_{SA[j]}S$. Pošto je znak terminacije `$` leksikografski manji od ostalih znakova, i svi sufiksi su različite duljine, usporedba bilo koja dva sufiksa se riješi najkasnije sa zadnjim znakom kraćeg sufiksa.

BWT je sličan sufiksnom polju, samo umjesto indeksa sufiksa, pamtimo znak koji prethodi tom sufiksu. Formalno, BWT je niz duljine n za koji vrijedi $BWT[i] = S[SA[i] - 1]$. Za i sa $SA[i] = 1$, $BWT[i] = \$$.

i	BWT	SA	
1	i	12	$
2	p	11	i$
3	s	8	ippi$
4	s	5	issippi$
5	m	2	ississippi$
6	$	1	mississippi$
7	p	10	pi$
8	i	9	ppi$
9	s	7	sippi$
10	s	4	sissippi$
11	i	6	ssippi$
12	i	3	ssissippi$

Slika 3.1: BWT i SA za niz `mississippi`

Ključno svojstvo sufiksnog polja je da će svi identični podnizovi niza znakova S biti na susjednim pozicijama u sufiksnom polju. To je tako jer je očito svaki podniz prefiks nekog sufiksa, i pošto su sufiksi sortirani, isti podnizovi se moraju nalaziti na uzastopnim lokacijama, odnosno na nekom intervalu sufiksnog polja. Upravo to svojstvo nam omogućuje da uz određene strukture možemo efikasno tražiti proizvoljne podnizove.

Definirajmo najprije dvije pomoćne strukture podataka. *Rank(c, p)* (rang) označava broj pojavljivanja znaka c u prvih p znakova niza BWT. *Less(c)* označava broj manjih znakova od znaka c u cijelom nizu BWT. Ako toj vrijednosti dodamo 1, dobijemo indeks na kojem počinju svi sufiksi čiji je prvi znak upravo c. Dok je na upite tipa *Less* jednostavno odgovarati pretprocesiranjem niza BWT, za *Rank* je to teško bez velikog memorijskog opterećenja. Jednostavno rješenje je pamtiti tu vrijednost za recimo svakih 32 ili 64 indeksa, a ostatak brojati. O tome više kasnije, dok zasad pretpostavljamo da je to $O(1)$ operacija.

Pomoću ove dvije strukture, spremni smo tražiti proizvoljne podnizove. Ideja je sljedeća – niz tražimo iterirajući od zadnjeg do prvog znaka, a u svakom koraku iteracije održavamo interval sufiksnog polja u kojem svi sufiksi počinju upravo sa trenutnim sufiksom niza kojeg tražimo. Ilustrirajmo pretragu niza `iss`:

```
korak   0   1   2   3
        >           -       i   $
                            p   i$
          +   -             s   ippi$
          +   -         >>  s   issippi$
                        >>  m   ississippi$
                            $   mississippi$
                            p   pi$
                        -       i   ppi$
          + > +             s   sippi$
          +   +             s   sissippi$
                  > +       i   ssippi$
        >   >   > +         i   ssissippi$
```

Slika 3.2: Traženje podniza `iss`

Krećemo sa cijelim intervalom, pošto svi sufiksi počinju s praznim nizom. Idemo u nazad, pa u prvoj iteraciji koristimo slovo `s`. Sad u svakoj iteraciji pomoću *Less()* nalazimo prvi sufiks koji počinje s tim slovom, a sa *Rank()* brojimo koliko ih treba uzeti, na temelju broja pojavljivanja tog slova na trenutnom intervalu. Pošto se radi o prvom slovu, novi interval je postavljen na cijeli interval sufiksa koji počinju sa `s`. U drugom koraku opet tražimo `s`, ali ovaj put dva sufiksa preskačemo, jer se nastavljaju sa `i`, i to vidimo jer su izvan trenutnog intervala. Oni koje preskačemo su označeni sa $-$, dok su dodani označeni sa $+$. I u zadnjem koraku eliminiramo još dva sufiksa koji

počinju sa `i` i imamo konačan rezultat. Sve što je potrebno jest pročitati vrijednosti sufiksnog polja da odredimo točne pozicije podniza.

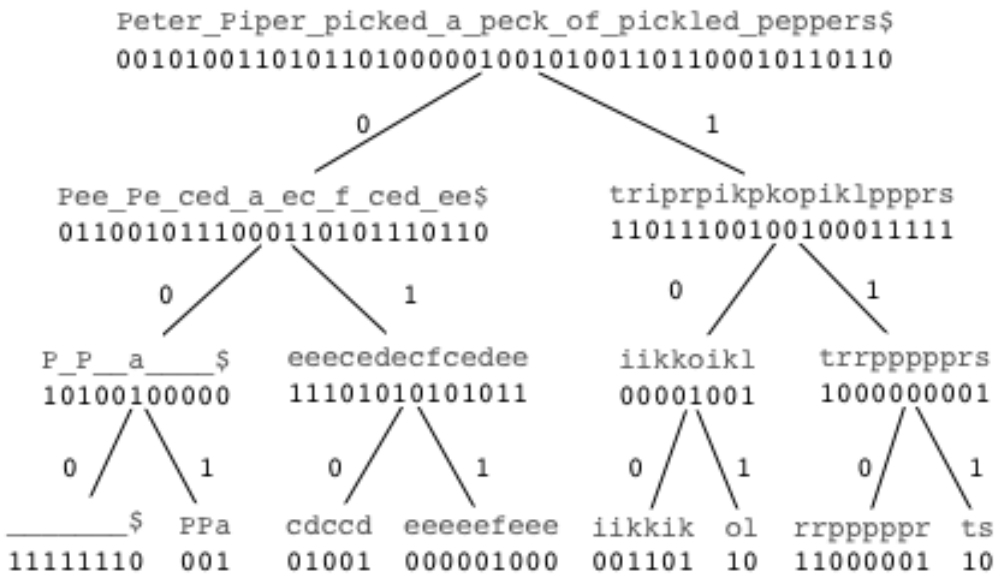
Prijelazi između intervala su opisani sljedećim izrazima:

$$new\_low = Less(c) + Rank(c, low - 1) + 1 \tag{3.1}$$

$$new\_high = Less(c) + Rank(c, high) \tag{3.2}$$

3.3.2. Valićno stablo

Kako bi efikasno mogli odgovarati na upite tipa *Rank*, potrebno je pretprocesirati BWT niz u odgovarajuće strukture podataka. Koristimo valićno stablo, inspirirano valićnom transformacijom, koje nam omogućava da na te upite odgovaramo u vremenskoj složenosti $O(\log \sigma)$, gdje je σ veličina abecede. U našem slučaju, ona je 5 jer je osim $\{A, C, G, T\}$ potrebno pamtiti i znak ^. Prema tome, vremenska složenost je praktički konstantna. Valićno stablo se oslanja na RRR strukturu, opisanu u sljedećoj sekciji, koja omogućava efikasne *Rank* upite nad binarnom abecedom. Pretpostavimo da zasad na takve upite znamo odgovarati.



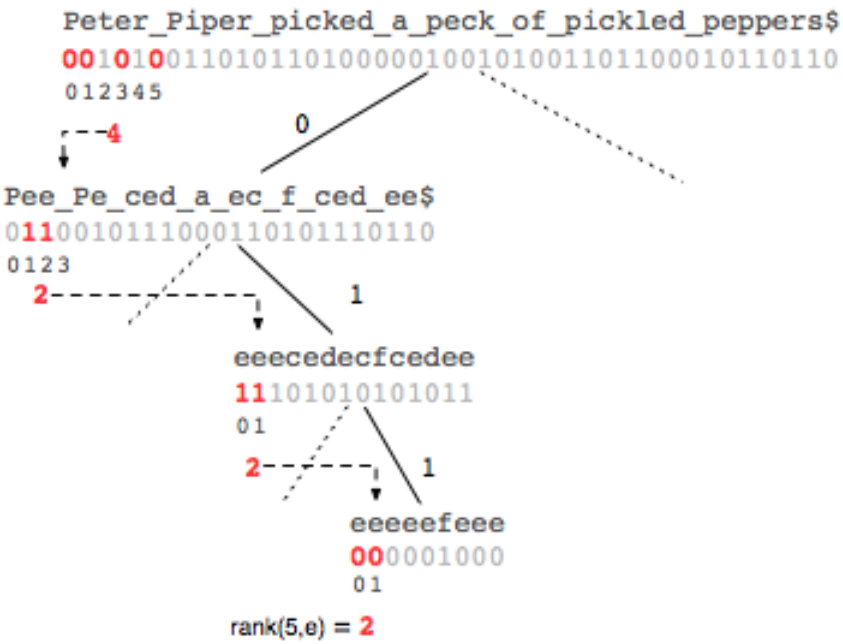
Slika 3.3: Primjer valićnog stabla

Valićno stablo rekurzivno gradimo na sljedeći način:

1. Prvih pola abecede kodiramo sa 0, a drugu polovicu sa 1. $\{\hat{}, A, C, G, T\}$ tako preuzima vrijednosti $\{0, 0, 1, 1, 1\}$.

2. Niz znakova odvajamo na dva niza, po jedan za 0 i 1. `ACGTATT^` postaje `ACA` i `GTTT^`.
3. Rekurzivno gradimo valićno stablo za ta dva niza, dok veličina abecede ne postane 1 ili 2. Kao lijevo dijete u stablu dodamo čvor za 0, a kao desno čvor za 1.

Za niz znakova "`peter_piper_picked_a_peck_of_picked_peppers$`" dobijemo stablo na slici 3.3.



Slika 3.4: Odgovaranje na upit *Rank(e, 5)*

Kako bi odgovorili na *Rank(c, p)* upit, u stablu krećemo od korijena. U svakom koraku najprije odredimo kojem bitu (0 ili 1) pripada znak, te računamo bitovni rang za istu poziciju. Ako u trenutnom čvoru tom bitu pripada samo znak koji tražimo, dobili smo rezultat. Inače se krećemo u dijete trenutnog čvora koje pripada odgovarajućem bitu. Vrijednost znaka ostaje ista, ali za poziciju koristimo vrijednost bitovnog ranga.

Slika 3.4 prikazuje kako odgovaramo na upit *Rank(e, 5)*. Treba imati na umu da ovdje indeksi počinju od vrijednosti 0.

1. U korijenu rješavamo upit *Rank(0, 5) = 4*. Krećemo se lijevo u podstablo 0. Nova pozicija je 3.

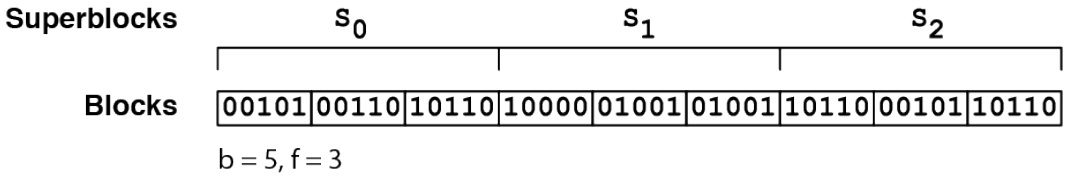
2. Rješavamo upit *Rank(1, 3) = 2*. Idemo desno u podstablo 1. Nova pozicija je 1.

3. Rješavamo upit *Rank(1, 1)* = *2*. Idemo desno u podstablo 1. Nova pozicija je 1.

4. Rješavamo upit *Rank(0, 1)* = *2*. Pošto je broj znakova u trenutnom čvoru 2, 2 je ujedno i rješenje.

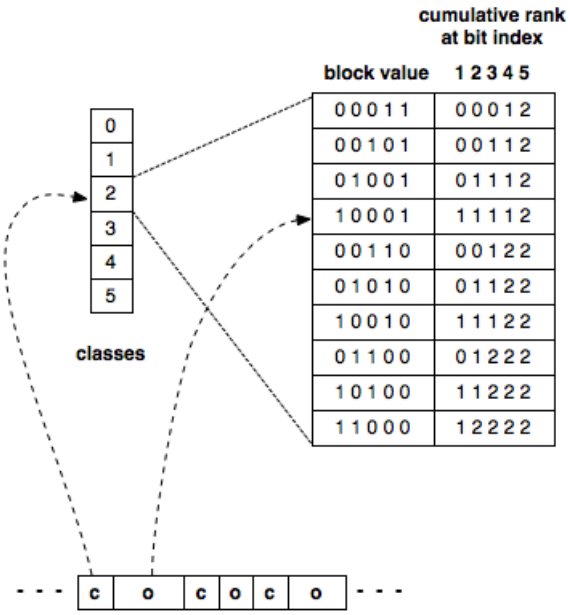
3.3.3. RRR struktura

U prethodnoj sekciji je objašnjeno kako za BWT izgraditi valićno stablo za efikasne *Rank* upite, ali preostaje objasniti kako efikasno odgovarati na rang upite za binarne nizove. RRR je statička struktura koja omogućava efikasne upite i ujedno vrši kompresiju ulaznog niza.



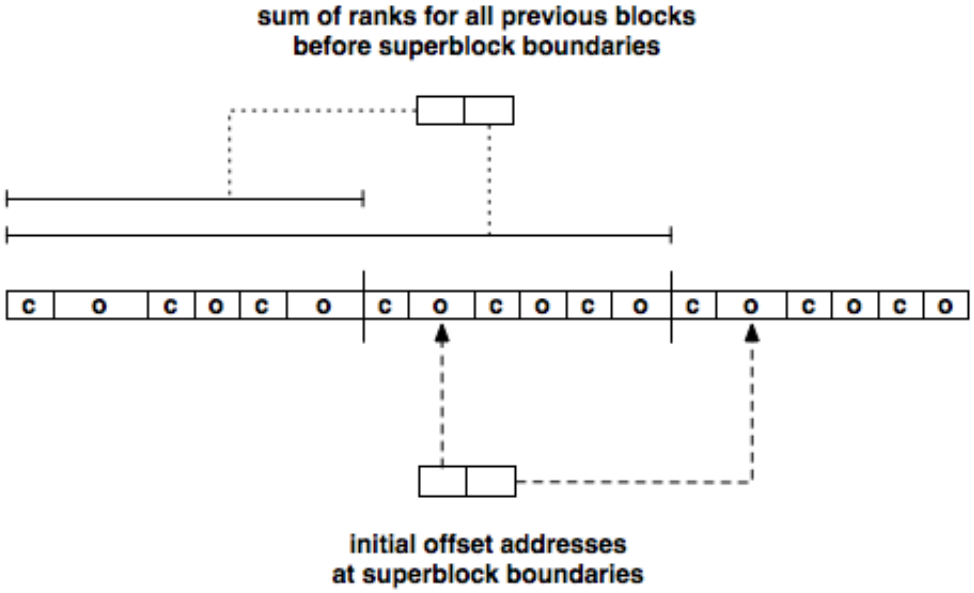
Slika 3.5: Podjela na blokove i super-blokove.

Ulazni niz bitova se najprije podijeli na blokove fiksne veličine, te se blokovi grupiraju u veće super-grupe. Na slici 3.5 je prikazana takva podjela za blokove veličine $b = 5$ bitova i super-blokove veličine $f = 3$ blokova.



Slika 3.6: Lookup tablica za vrijednosti blokova

Zatim mijenjamo svaki blok sa dvije vrijednosti, C za oznaku klase i O za odmak. Klasa C označava da blok ima C postavljenih bitova, i na temelju nje se određuje broj bitova odmaka O, čime se postiže kompresija. U odgovarajućoj tablici na slici 3.6 za sve moguće vrijednosti blokova računamo kumulativni broj postavljenih bitova do svakog bita u bloku.



Slika 3.7: Super blokovi

Osim toga, za svaki super-blok pamtimo adresu na početak bloka, kao i broj postavljenih bitova do početka tog bloka, kao što je prikazano na slici 3.7

Kako bi odgovarali na upite, najprije izračunamo indeks bloka i super-bloka za zadanu poziciju. Rang do početka super-bloka već imamo izračunat, te je samo potrebno šetat se po blokovima do bloka u kojem se zadani bit nalazi, koristeći vrijednosti C za šetanje i računanje privremenog ranga. Kad dođemo do traženog bloka, koristimo preračunatu tablicu da bi izračunali rang do zadanog bita.

3.3.4. Traženje egzaktnih preklapanja

Kako bi našli egzaktna preklapanja koristeći gore definirane strukture, najprije treba objasniti kako izgraditi FM-indeks iz skupa očitanja. Sufiksno polje za više nizova se naziva generalizirano sufiksno polje i ono osim pozicije u nizu, mora pamtiti i o kojem nizu je riječ. No, vidjet ćemo da za naše potrebe to nije potrebno, jer ćemo isključivo tražiti prefikse očitanja.

Zamislimo da imamo dva različita znaka terminacije: `$` i `^`, koji su oba manja od svih ostalih znakova, te `$` < `^`. Dovoljno je uzeti skup od N očitanja $\mathcal{R}$ i izgraditi BWT iz niza koji dobijemo konkatenacijom svim očitanja: $\hat{}R_1\hat{}R_2\hat{}\cdots\hat{}R_N\$$. Primijetimo da će prvi element sufiksnog polja biti `$`, te da nakon toga slijedi N elemenata, odnosno svako očitanje prefiksirano sa `^`, sortirano leksikografski. Nakon toga slijede svi sufiksi, a jednaki sufiksi će biti međusobno konzistentno sortirani ovisno o drugim elementima.

Sada bilo koji prefiks možemo jednostavno naći, tako da na početak prefiksa kojeg tražimo dodamo `^`. Osim toga, nije ni potrebno pamtiti sufiksno polje, već samo niz indeksa sortiranih leksikografski na temelju odgovarajućih očitanja. Dobiveni interval smanjimo za 1 zbog početnog `$` te pročitamo indekse sortiranih nizova na tom intervalu kako bi odredili prefikse.

Kako bi našli sva preklapanja, za svako očitanje idemo od kraja održavajući interval za trenutni sufiks, te provjeravajući za svaki sufiks dovoljne duljine ima li preklapajućih prefiksa. Ako je na kraju iteracije interval veći od 1, to znači da postoji barem jedno očitanje koje potpuno sadržava trenutno, i moguće je trenutno označiti kao sadržano.

Pseudokod 1 prikazuje rad ovog algoritma.

3.4. Sufiksni filtar

3.4.1. Levenshtein udaljenost

Pošto baratamo sa očitanjima koji imaju ugrađen visok stupanj pogreške, potrebno je definirati metodu određivanja sličnosti, odnosno različitost znakovnih nizova. Prikladna metoda jest upravo Levenshtein udaljenost, popularnije zvana udaljenost uređivanja (engl. *edit distance*). Levenshtein udaljenost može se opisati kao minimalan broj operacija koje je potrebno obaviti na jednom nizu kako bi bio jednak drugom nizu. Moguće operacije su supstitucija, brisanje i dodavanja znaka.

```
CGTA-GCTAAGGT
CCTATGCT-GCGT
```

Slika 3.8: Optimalno poravnanje nizova `CGTAGCTAAGGT` i `CCTATGCTGCGT`.

Na primjeru 3.8 prikazano je poravnanje dva niza DNA greške 5. Iako postoje metode koje daju različite vrijednosti kazne za razne operacije, za DNA ovo predstavlja

Algorithm 1 Traženje egzaktnih preklapanja

```
function Prev(fm, c, lo, hi)
    nlo ← fm.Less(c) + fm.Rank(c, lo − 1) + 1
    nhi ← fm.Less(c) + fm.Rank(c, hi)
     return (nlo, nhi)
end function
function FindExactOverlap(read, fm, order, t)
    lo, hi ← (1, Size(fm))
    i ← 0
    for all c ∈ Rev(read) do
        i ← i + 1
        lo, hi ← Prev(fm, c, lo, hi)
        if i ≥ t then
            plo, phi ← Prev(fm, ^, lo, hi)
            if plo ≤ phi then
          AddOverlaps(read, order, plo, phi, i)
            end if
        end if
    end for
    if hi − lo ≥ 1 then
          AddContained(R)
    end if
end function
```

dovoljno dobru aproksimaciju. Za dva niza duljina n i m, Levenshtein udaljenost računamo primjenom dinamičkog programiranja nad rekurzivnom relacijom:

$$LH(i,j) = \min \begin{cases} LH(i-1,j-1) + D(i,j) \\ LH(i-1,j) + 1 \\ LH(i,j-1) + 1 \end{cases} \tag{3.3}$$

$$LH(i,0) = i, \;\; 0 \leq i \leq n \tag{3.4}$$

$$LH(0,j) = j, \;\; 0 \leq j \leq m \tag{3.5}$$

$$D(i,j) = \begin{cases} 0 & S_1[i] = S_2[j] \\ 1 & \text{inače} \end{cases} \tag{3.6}$$

Za sufiksna polja nije potrebno eksplicitno poravnavati dva niza, niti računati Levenshtein udaljenost, ali u fazi validacije jest. Algoritam je vremenske složenosti $O(nm)$ i memorijske $O(n+m)$ bez rekonstrukcije. U nastavku Levenshtein distance između nizova A i B označavamo sa $ed(A,B)$.

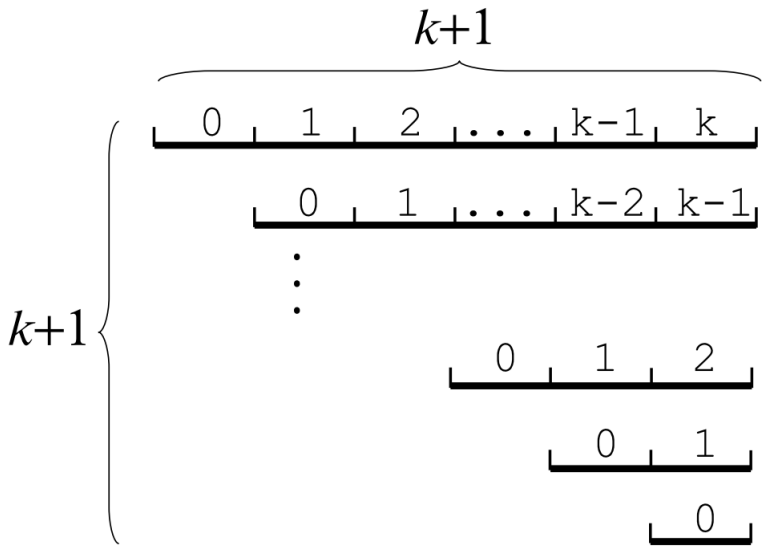
3.4.2. Osnovna svojstva sufiksnog filtra

Prvo promatramo preklapanje dva niza A i B sa najvećom fiksnom Levenshtein udaljenošću k. Sufiksni filtar podijeli niz A na $k+1$ faktora $A_1 A_2 \cdots A_{k+1}$. Za bilo koji niz B postoji optimalna podjela na faktora $B_1 B_2 \cdots B_{k+1}$ takva da je $ed(A,B) = \sum_{i \in [1,k+1]} ed(A_i, B_i)$. Interval faktora $A_i A_{i+1} \cdots A_j$ označavamo sa $A[i,j]$ za bilo koji $0 < i \leq j \leq k+1$. Za nizove A i B kažemo da se *preklapaju* na intervalu $[i,j]$ ako je $ed(A[i,j], B[i,j]) \leq j - i$. Također, nizovi A i B se *jako preklapaju* na intervalu $[i,j]$ ako se preklapaju na svakom prefiksu $[i,j']$ tog intervala, gdje je $j' \in [i,j]$.

Prvo ključno svojstvo je da ako je $ed(A,B) \leq k$, onda postoji $i \in [1, k+1]$ takav da se A i B jako preklapaju na intervalu $[i, k+1]$.

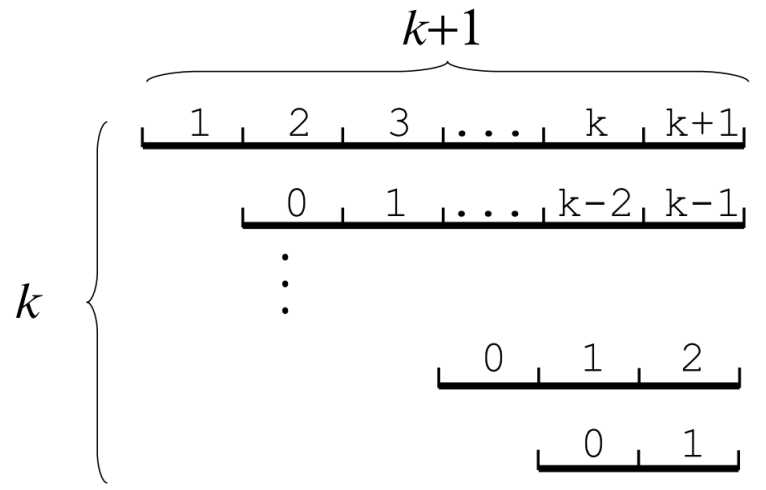
Drugo svojstvo je da ako je $ed(A,B) \leq k$, neka je $[1,i]$ najdulji prefiksni interval nad kojim se A i B ne preklapaju, tj. $ed(A[1,i], B[1,i]) \geq i$. Onda se A i B jako preklapaju na intervalu $[i+1, k+1]$.

Iz tih svojstava možemo deducirati prvi oblik sufiksnog algoritma za određivanje kandidata. Niz A podijelimo na $k+1$ faktora jednake veličine i za svaki sufiks $A[i, k+1]$ tražimo jaka preklapanja, tako da za prvi faktor u svakom sufiksu dodijelimo maksimalnu udaljenost 0 koju povećamo za 1 na granici svakog faktora. Slika 3.9 prikazuje sufikse koje tražimo i odgovarajuće dopuštene greške, tj. udaljenosti za



Slika 3.9: Prva verzija sufiksnog filtra

prefiks svakog sufiksa. Ključni problem koji ovakav pristup predstavlja jest da ne možemo utjecati na veličinu zadnjeg faktora. Pošto je zadnji sufiks najkraći, za očekivati je da će potencijalno rezultirati najvećim broj kandidata. Upravo zato modificiramo sufiksni filtar da za prvi prefiks dopusti grešku više, ali ukloni zadnji sufiks.



Slika 3.10: Druga verzija sufiksnog filtra

Dosad smo pričali o preklapanju s fiksnom greškom k, no mi radimo sa drugačijim parametrima, konkretno mjerom pogreške ϵ i minimalnom duljinom preklapanja t. Preklapanje duljine l je dobar ako je greška najviše ϵl i $l \geq t$. Ako je očitanje za čije sufikse tražimo preklapanje duljine n, potencijalna preklapanja mogu biti duljine

u intervalu $[t, n]$. Zato za duljinu faktora biramo najmanji mogući od svih:

$$p = \min_{l=t}^{n} \left\lceil \frac{l}{\lceil \epsilon l \rceil + 1} \right\rceil \tag{3.7}$$

Preostaje odrediti kako za određeni sufiks naći sve kandidate.

3.4.3. Određivanje kandidata preklapanja

Prethodno smo pokazali koncept sufiksnog filtra. Preostaje pokazati kako to povezati sa FM-indeksom i primijeniti za traženje kandidata preklapanja između sufiksa jednog očitanja sa više prefiksa drugih očitanja preko FM-indeksa.

Najprije treba primjetiti da algoritam traženja egzaktnih preklapanja radi od kraja prema početku, što se kosi s idejom iz sufiksnih filtra, gdje iterativno testiramo prefiks svakog sufiksa. No, sva svojstva iz sufiksnog filtra su simetrična, tako da možemo okrenuti i zapravo koristiti prefiksne filtre.

Očitanje implicitno razbijamo na faktore jednake veličine i iteriramo od zadnjeg faktora prema prvom, pozivajući za svaki potprogram koji rekurzivno isprobava sve moguće kombinacije unutar zadane pogreške. Očitanje implicitno razbijamo na faktore jednake veličine i iteriramo od zadnjeg faktora prema prvom, pozivajući za svaki potprogram koji rekurzivno isprobava sve moguće kombinacije unutar trenutne zadane pogreške.

Rekurzivni potprogram će simulirati rad algoritma 1, ali će se granati za sve moguće kombinacije supstitucija, brisanja znakova ili dodavanja istih, sve unutar dozvoljene greške. Na prijelazu faktora dozvoljenu grešku povećavamo za 1. Osim toga, nakon što duljina preklopljenog sufiksa prijeđe prag duljine preklapanja nakon što je i preklopljen barem jedan cijeli faktor, algoritam će u svakom koraku testirati ima li prefiksa drugih očitanja koji odgovaraju prefiksu trenutnog sufiksa. Takve prefikse dodajemo u listu kandidata preklapanja za daljnju validaciju.

Pseudokod algoritma 2 prikazuje pojednostavljenu rekurzivnu metodu koja za određeni prefiks sufiksa nalazi sve kandidate. Statičke varijable koje se ne mijenjaju su izbačene iz liste argumenata zbog preglednosti, te su samo zadržane varijable stanja. Varijabla $start$ označava gdje završava trenutni prefiks, dok pos označava koliko smo se pomaknuli u lijevo od početne pozicije.

Prava implementacija ne koristi rekurziju već pretraživanje u širinu nad stanjem $(low, high, pos)$ kako bi do svakog stanja došli u minimalnom broju koraka i tako izbjegli ponavljanje istih.

Algorithm 2 Određivanje kandidata sufiksa

function $NeegzaktnoPreklapanje(pos, lo, hi, error)$
  if $lo > hi$ **then return**
  end if
  $overlap\_size \leftarrow Size(read) - start + pos$
  if $pos \geq factor\_size \wedge overlap\_size \geq t$ **then**
    $plo, phi \leftarrow Prev(fm, \hat{}, lo, hi)$
    if $plo \leq phi$ **then**
      $AddCandidates(read, order, plo, phi, error, overlap\_size)$
    end if
  end if
  if $pos = start$ **then return**
  end if
  if $pos + 1 \mod factor\_size = 0$ **then**
    $nerror \leftarrow error + 1$
  else
    $nerror \leftarrow error$
  end if
  if $error > 0$ **then**
    $NeegzaktnoPreklapanje(pos + 1, lo, hi, nerror - 1)$
  end if
  for all $c \in \{A, C, G, T\}$ **do** $nlo, nhi \leftarrow Prev(fm, c, lo, hi)$
    if $c = read[start - pos]$ **then**
      $NeegzaktnoPreklapanje(pos + 1, nlo, nhi, nerror)$
    else if $error > 0$ **then**
      $NeegzaktnoPreklapanje(pos + 1, nlo, nhi, nerror - 1)$
      $NeegzaktnoPreklapanje(pos, nlo, nhi, error - 1)$
    end if
  end for
end function

3.4.4. Filtriranje kandidata

Nakon što je algoritam 2 pokrenut za sve prefikse svih očitanja, ono što će se dogoditi jest da će se ista preklapanja pojaviti više puta, sa manjim razlikama u duljini preklapanja i preostaloj dozvoljenoj greški na istom.

Pretpostavimo da za neko očitanje R_T tražimo kandidate i da je pronađeno potencijalno preklapanja sa očitanjem R_O, duljine preklapanja l na sufiksu očitanja R_T sa preostalom greškom e. Osim toga, pretpostavljamo da je duljina neprekloplјenog prefiksa od R_T mnogo veća od e. Osim ovog, gornji algoritam će pronaći i kandidate oblika $(l+1, e-1)$ za svaki $0 < i \leq e$, koje je potrebno izbaciti.

Nepotrebne kandidate izbacimo sljedećim postupkom:

1. Za svaki jedinstveni triplet (r_i, r_j, t), gdje t označava tip preklapanja, grupiramo sve kandidate i sortiramo uzlazno po duljini preklapanja l i silazno po preostaloj greški e.

2. Svaku grupu zatim nezavisno filtriramo. Za neki kandidat C, $C.l$ označava duljinu preklapanja a $C.e$ preostalu dozvoljenu grešku.

3. Iteriramo po sortiranim kandidatima, provjeravajući za svaki kandidat Cl postoji li kandidat Ch takav da vrijedi $Ch.l < Cl.l$ i $Cl.l - Ch.l <= Ch.e - Cl.e$. Ako Ch postoji, brišemo Cl iz liste kandidata i nastavljamo dalje.

Osim toga, izbacujemo sve kandidate preklapanja nekog očitana sa samim sobom, preklapanja između dva obrnuto komplementirana očitanja i duplikate kandidata preklapanja tipa EE i BB, pošto su te dvije vrste preklapanja komutativne.

3.5. Validacija

Zadnji korak je testirati svako potencijalno preklapanje. Najprije za svako preklapanje treba odrediti točnu mjeru pogreške. Ako na trenutak zanemarimo da postoje obrnuti komplementi, onda za par očitanja (i, j) za koji postoji kandidat preklapanja duljine l_1, znamo da je prvi prekloplјen na sufiksu, a drugi na prefiksu.

Treba primijetiti kako ne znamo duljinu preklopa na prefiksu drugog očitanja, dok za sufiks prvog točno znamo. Zbog toga određujemo maksimalnu duljinu preklapanja na prefiksu drugog očitanja kao $l_2 = l \times (1_1 + \epsilon)$ te maksimalnu dozvoljenu pogrešku $e_m = l_1\epsilon$. Zatim pokrećemo metodu koja poravnava sufiks prvog očitanja duljine l_1 i prefiks drugog očitanja duljine l_2 sa ograničenim pojasom greške e_m.

Pošto preklapanje na prefiksu može završiti bilo gdje, varijanta poravnavanja koje pokrećemo pokušava preklopiti cijeli prvi niz znakova sa prefiksom drugog niza. Takvom metodom odredimo ispravljenu duljinu l_{2C} prefiksa drugog očitanja koji se preklapa, kao i samu vrijednost pogreške e. Također, metoda poravnavanja prima parametar e_m kako bi se postigla bolja vremenska složenost. Svi kandidati sa $e > e_m$ se izbacuju.

Sada još preostaje izračunati pouzdanost preklapanja kao $l_1 + l_{2C} - e$, te za svaki jedinstveni par očitanja sa barem jednim preostalim kandidatom biramo onaj sa najvećom pouzdanosti.

4. Rezultati

4.1. Metoda validacije

Za testiranje su korištena očitanja uzorkovana iz referentnog genoma pomoću alata `readsim`[1]. Taj alat prilikom simulacije očitanja za svako očitanje ispiše približnu poziciju u referentnom genomu.

Skripta `reads_overlaps` računa skup preklapanja na temelju tih lokacija sa namjenom da se koristi kao referentni prilikom testiranja prave implementacije.

Skripta `validate` čita dva seta preklapanja – referentni (rezultat prethodne skripte) i testni (rezultat glavnog programa).

Neka je N_R broj preklapanja u referentnom skupu, N_T broj preklapanja u testnom skupu i N_P zajedničkih preklapanja, odnosno presjek dva skupa. Definiramo sljedeće mjere validacije:

Preciznost. Omjer zajedničkih preklapanja i testnog skupa:

$$P = N_P/N_P \tag{4.1}$$

Odziv. Omjer zajedničkih preklapanja i referentnog skupa:

$$R = N_R/N_P \tag{4.2}$$

F1-mjera. Harmonička sredina preciznosti i odziva:

$$F1 = \frac{2PR}{P+R} \tag{4.3}$$

Osim toga, analizirane su vrijednosti $F_{0.5}$ (pristrana preciznosti) i F_2 (pristrana odzivu). Pošto je za očekivati da će biti lažnih pozitiva koji se ne mogu izbjeći zbog dijelova genoma koji se ponavljaju, držimo da je veći prioritet odziv od preciznosti, i prema tome je $F_{0.5}$ važnija mjera.

[1] http://sourceforge.net/projects/readsim/

4.2. Testiranja

Testiranja su izvođena na serveru s dva Intel® Xeon® E5645 procesora sa po 6 jezgri (12 virtualnih) na 2.40 GHz.

Za testiranje je korišten dio genoma *E. coli* veličine oko 100K nukleotida. Generirana su očitanja duljina $\{100, 200, 500, 1000, 2000, 5000\}$ sa prosječnom pokrivenošću $10\times$. Radi se o simulaciji ispravljenih PacBio očitanja prosječne greške 1%.

Glavnom programu je zadana maksimalna dopuštena greška 2% sa dodatnim faktorom dopuštene greške od $2\times$ prilikom validacije. Osim toga, minimalna duljina očitanja je postavljena na 60, a minimalna duljina preklapanja na 30.

Tablica 4.1: Prvi dio rezultata

	broj očitanja	broj kandidata	broj preklapanja	vrijeme (m/s)
100	6209	35530	22417	30.458s
200	3975	26849	15336	1m 45.028s
500	1762	11616	6099	7m 41.898s
1000	947	7065	3715	27m 24.110s
2000	457	3334	1643	138m 44.042s

Tablica 4.2: Drugi dio rezultata

	preciznost	odziv	F_1	$F_{0.5}$	F_2
100	80.03%	67.20%	73.06%	77.09%	69.43%
200	88.99%	69.31%	77.92%	84.21%	72.52%
500	94.59%	73.52%	82.73%	89.46%	76.95%
1000	97.12%	73.81%	83.88%	91.35%	77.53%
2000	98.17%	75.83%	85.57%	92.71%	79.45%

Rezultati su prikazani u tablicama 4.1 i 4.2. Kao što možemo vidjeti, vrijeme eksponencijalno raste sa duljinom očitanja, iako ih za veće duljine ima drastično manje. S druge strane, iznenađujuće je da za preciznost drastično raste sa veličinom očitanja.

5. Zaključak

Dok je FM-indeks sjajna struktura koja omogućava izetne brzine egzaktne pretrage, još treba puno raditi na razvoju metoda koje će bolje tolerirati veće greške bez gubitka kvaliteta koje FM-indeks pruža.

Tako se pokazalo da je naivna implementacija sufiksnog filtra bez dodatnih modifikacija poprilično loša upravo za dulja očitanja. Također, niska vrijednost praga duljine preklapanja u kombinaciji s visokim stupnjem pogreške gotovo da čini sufiksno polje potpuno beskorisnim u kontektnu brzine izvođenja.

No, iako ovakav pristup možda nije zasad dao dobre rezultate za PacBio očitanja, postoji mogućnost da je mnogo prikladnije za očitanja s manjom pogreškom ili rad sa ispravljenim očitanjima, što bi svakako trebalo testirati.

Također, postoji nekoliko analiza koje bi trebalo napraviti:

- Analiza raspršenosti hash vrijednosti stanja tokom pretraživanja u širinu.
- Upotreba bolje hash tablice za čuvanje vrijednosti stanja.
- Analiza korelacije vrijednosti stanja i rezultirajućih kandidata.
- Istražiti tehnike rezanja prostora stanja.
- Daljnje optimizacije na nižoj razini.

LITERATURA

Alex Bowe. Multiary wavelet trees in practice. Magistarski rad, School of Computer Science and Information Technology RMIT University, Melbourne, Australia, 2010.

Heikki Hyyrö. Explaining and extending the bit-parallel approximate string matching algorithm of Myers. Technical report, 2001.

Eugene W. Myers. The fragment assembly string graph. 2005.

Gene Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM*, 46:1–13, 1999.

N. Nagarajan i M. Pop. Sequence assembly demistified. 2013.

G. Nong, S. Zhang, i W. H. Chan. Linear suffix array construction by almost pure induced-sorting. 2009.

Jared T. Simpson i Richard Durbin. Efficient construction of an assembly string graph using the fm-index. 2010.

Jared T. Simpson i Richard Durbin. Efficient de novo assembly of large genomes using compressed data structures. 2011.

Niko Välimäki, Susan Ladra, i Veli Mäkinen. Approximate all-pairs suffix/prefix overlaps. 2012.

Otkrivanje preklapajućih DNA očitanja

Sažetak

U vrijeme brzog razvoja tehnologija sekvenciranja DNA, sve je veća potreba za sastavljanjem istih. Ovaj rad se bavi implementacijom modernih metoda za otkrivanje preklapanja DNA očitanja kao dio OLC (engl. *overlap – layout – consensus*) pristupa problemu. Primjenom FM-indeksa baziranog na valićnim stablima i uporabom sufiksnog filtra za traženje kandidata preklapanja, napravljena je uspješna C++ implementacija. Iako originalno namjenjen za testiranje na PacBio očitanjima, rezultati su pokazali da je metoda prikladnija s manjom pogreškom.

Ključne riječi: De novo sastavljanje genoma, FM-indeks, sufiks filtar.

Finding overlapping DNA reads

Abstract

In the time of fast development of DNA sequencing methods, there is a growing need for assembling genomes. The goal of this thesis is to build an efficient implementation of modern methods to detect read overlaps as part of OLC (overlap – layout – consensus) framework for genome assembly. Using FM-index based on wavelet trees and suffix filters to find overlap candidates, we have successfully made a C++ implementaion. Although initally intended to be tested on and work with PacBio sequences, tests have shown that this method might be better suited for reads with smaller error.

Keywords: De novo assembly, FM-index, suffix filter.