SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 6159

Izgradnja biblioteke za poravnavanje parova dugačkih RNA očitanja

Rafael Josip Penić

Zagreb, lipanj 2019.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA
ODBOR ZA ZAVRŠNI RAD MODULA

Zagreb, 13. ožujka 2019.

ZAVRŠNI ZADATAK br. 6159

Pristupnik: **Rafael Josip Penić (0036498133)**
Studij: Računarstvo
Modul: Računarska znanost

Zadatak: **Izgradnja biblioteke za poravnavanje parova dugačkih RNA očitanja**

Opis zadatka:

Glavni izazov u izgradnji alata za mapiranje RNA očitanja je izgradnja preciznog i brzog alata za poravnavanje očitanja s velikim procjepima. U ovom radu potrebno je izraditi biblioteku za poravnavanje RNA očitanja dobivenih alatima za sekvenciranje treće generacije. Biblioteka treba implementirati algoritam za poravnavanje dvaju sekvenci opisan u radu "Optimal sequence alignment allowing for long gaps". Dodatno je potrebno istražiti mogućnost ubrzanja rada biblioteke koristeći vektorizaciju.
Točnost i brzinu izvođenja biblioteke potrebno je usporediti sa trenutno najpopularnijim alatom za poravnavanje RNA očitanja, KSW2. Točnost je potrebno evaluirati podacima koji će biti prethodno pripremljeni i alatom za evaluaciju korištenim u radu "Evaluation of tools for long read RNA-seq splice-aware alignment".
Rješenje mora biti napisano u jezicima C ili C++. Programski kod je potrebno komentirati i pri pisanju pratiti neki od standardnih stilova. Napisati iscrpne upute za instalaciju i izvođenje. Kompletnu aplikaciju postaviti na Github pod jednom od OSI-odobrenih licenci.
U svezi dobivanja detaljnih informacija obratiti se Josipu Mariću, mag. ing.

Zadatak uručen pristupniku: 15. ožujka 2019.
Rok za predaju rada: 14. lipnja 2019.

Mentor:
Mile Šikić
Prof. dr. sc. Mile Šikić

Predsjednik odbora za završni rad modula:
Marko Čupić
Doc. dr. sc. Marko Čupić

Djelovođa:
Izv. prof. dr. sc. Tomislav Hrkać

SADRŽAJ

1. Uvod

1.1. Bioinformatika

Bioinformatika je grana znanosti koja računalne resurse koristi za analizu gena, bjelančevina i ostalih bioloških podataka.

1.2. RNA

RNA je, uz DNA, najvažnija nukleinska kiselina i ima vrlo važnu ulogu u razvoju i svakodnevnom funkcioniranju živih bića te bitnim biološkim procesima kao što je transkripcija. Kao i DNA, RNA je građena od nukleotida, ali za razliku od DNA, koji se sastoji od dva lanca, sastoji se od samo jednoga lanca. Nukleinske baze koje čine RNA lanac su adenin, gvanin, citozin i uracil.

RNA molekule mogu se podijeliti u nekoliko osnovnih skupina s obzirom na njihovu funkciju:

- glasnička RNA (mRNA)
- transportna RNA (tRNA)
- ribosomska RNA (rRNA)
- regulacijska RNA
    - mikro RNA (miRNA)
    - mala jezgrena RNA (snRNA)
    - mala interferirajuća RNA (siRNA)

1.3. Poravnavanje bioloških slijedova

Poravnavanje bioloških slijedova jedan je od najvažnijih problema koje istražuje bioinformatika. Svrha tog postupka je pronaći sličnosti između slijedova koji se porav-

navaju, a ta se informacija zatim može upotrijebiti za pronalaženje evolucijski nepromijenjenih dijelova gena i razne druge stvari.

Da bi se u potpunosti razumio pojam poravnavanja slijedova prvo se mora shvatiti pojam udaljenosti između nizova znakova. Taj pojam prvotno se pojavio u teoriji informacije gdje je definirana **Hammingova udaljenost**. Ako primjerice imamo nizove znakova a i b onda je njihova Hammingova udaljenost broj pozicija u kojima imaju različite znakova. Tako na primjer Hammingova udaljenost nizova "*drvo*" i "*prvo*" iznosi jedan pošto se navedeni nizovi razlikuju u samo jednom znaku.

Mana Hammingove udaljenosti je što je primjenjiva samo kada su nizovi koji se uspoređuju jednakih duljina te je stoga 1965. Levenshtein poopćio problem sličnosti nizova i na nizove proizvoljnih duljina te uveo pojam **udaljenosti uređivanja**. Udaljenost uređivanja između nizova a i b je broj modifikacija koje treba provesti na nizu a da bi se taj niz pretvorio u niz b. Postoje tri operacije koje se mogu izvršiti nad nekim znakom niza:

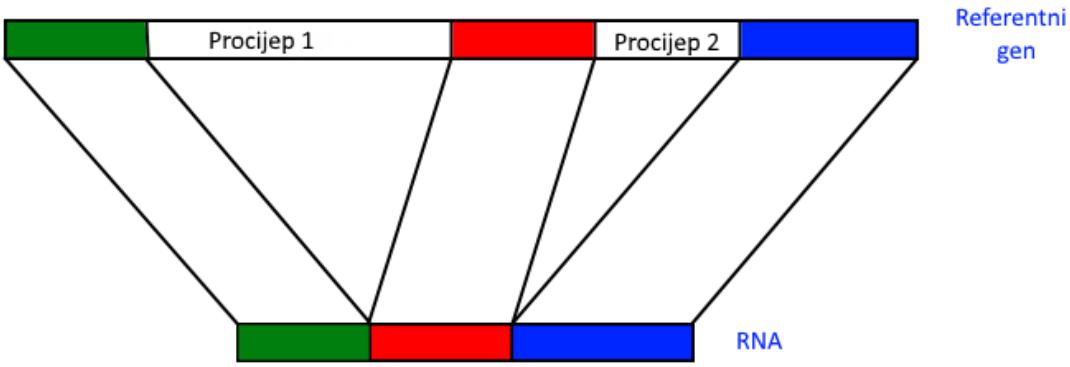
- Umetanje
  - *rob* => ***g****rob*
- Brisanje
  - *s****t****ol* => *sol*
- Zamjena
  - ***p****rvo* => ***d****rvo*

Poravnavanje bioloških slijedova u biti nastoji pronaći najmanju udaljenost uređivanja zadanih nizova te koje su modifikacije nizova dovele do "pretvorbe" prvoga niza u drugi.

Za opis poravnavanja u bioinformatici, koristi se CIGAR niz.

1.4. Poravnavanje RNA očitanja

Specifičnost pri poravnavanju RNA očitanja na referentni gen je mogućnost pojave velikih nizova brisanja (procijepa) koji nastaju kao rezultat raznih bioloških procesa. Zbog toga ne možemo koristiti uobičajene algoritme poravnavanja jer oni neće prepoznati te procijepe. Jedan od najpoznatijih algoritama koji osigurava da se takvi procijepi prepoznaju je Gotohov algoritam, a tim algoritmom inspiriran je i jedan od najpopularnijih alata za poravnavanje RNA očitanja: KSW2.



Slika 1.1: Primjer poravnavanja RNA očitanja na referentni gen

2. Gotohova metoda

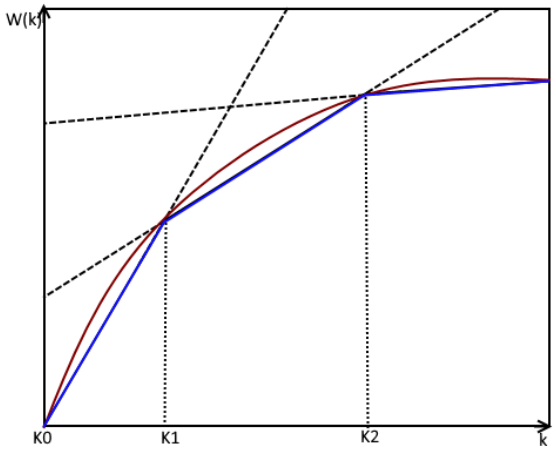
Pri klasičnom poravnavanju brisanje se kažnjava na sličan način kao umetanje ili zamjena, no ponekad nam je takav način kažnjavanja nepoželjan pošto su procijepi (dugački nizovi brisanja ili umetanja) očekivana pojava, kao kod poravnavnja RNA očitanja. U tom slučaju ukupnu kaznu brisanja određujemo konkavnom funkcijom koja će nam osigurati da što je procijep dulji, to brisanja koja dolaze kažnjavamo manje.

Za neku funkciju f kažemo da je konkavna na intervalu od a do b ako za bilo koji $\beta \in [0, 1]$ vrijedi:

$$f((1-\beta)a + \beta b) \geq (1-\beta)f(a) + \beta f(b) \tag{2.1}$$

Primjer konkavne funkcije je logaritamska funkcija.

Unatoč tome što konkavna funkcija daje idealne vrijednosti za cijene procijepa loše ju je koristiti jer je izračunavanje takve funkcije vremenski skupa operacija i stoga se obično pribjegava aproksimaciji konkavne funkcije određenim brojem afinih funkcija.



Slika 2.1: Aproksimacija konkavne funkcije afininim funkcijama. Crvenom linijom je prikazana konkavna funkcija koja se aproksimira, a plavom bojom su prikazane afine funkcije

2.1. Algoritam

Recimo da tražimo poravnanje između niza *a* duljine *M* i niza *b* duljine *N*. Kako bi se našlo optimalno poravnanje, Gotohov algoritam popunjava matricu poravnavanja D rekurzivnom relacijom:

$$D(m,n) = \min\{D(m-1,n-1) + d(a_m, b_n), F(m,n), G(m,n)\} \tag{2.2}$$

gdje za *F(m, n)* i *G(m, n)* vrijedi:

$$F(m,n) = \min_{1\le k\le m}\{D(m-k,n) + w(k)\} \tag{2.3}$$

$$G(m,n) = \min_{1\le k\le n}\{D(m,n-k) + w(k)\} \tag{2.4}$$

d(a_m, b_n) predstavlja funkciju koja vraća mjeru sličnosti između znakova *m*-tog znaka niza *a* i *n*-tog znaka niza *b*, a *w(k)* je konkavna funkcija (ili točnije aproksimacije konkavne funkcije) kojom se kažnjavaju procijepi.

Pošto je konkavna funkcija aproksimirana afinim, funkcija *w(k)* je zapravo "sastavljena" od *L* afinih funkcija oblika $w_i(k) = u_i k + v_i$ gdje vrijedi $K_{i-1} < k \le K_i$ za svaki $i \in [1, L]$. Vrijedi da je $K_0 = 0$ i $K_L = \infty$, a kako bi se očuvala konkavnost bitno je da su zadovoljene nejednakosti $u_i > u_{i+1} \ge 0$, $w_i(K_i) \le w_{i+1}(K_i)$ i $w_i(K_i + \epsilon) > w_{i+1}(K_i + \epsilon)$ gdje je $i \in [1, L - 1]$, a ϵ predstavlja neki pozitivan broj.

Ukoliko su navedena svojstva zadovoljena lako se može uočiti da za $k \le K_{i-1}$ $(i \in [2, L])$ vrijedi:

$$D(m-k,n) + w_i(k) \ge D(m-k,n) + w_{i-1}(k) \tag{2.5}$$

Slična relacije može se uočiti i za slučaj kada je k $\ge K_i$ ($i \in [1, L - 1]$):

$$D(m-k,n) + w_i(k) > D(m-k,n) + w_{i+1}(k) \tag{2.6}$$

Iz (2.5) i (2.6) možemo vidjeti da za svaku afinu funkciju $w_i(k)$ kojom aproksimiramo konkavnu funkciju postoji barem jedna druga afina funkcija (na primjer $w_{i-1}(k)$) koja će zbrojena s *D(m - k, n)* za $k \le K_{i-1}$ ili $k > K_i$ dati rezultat manji ili jednak od *D(m - k, n)* + $w_i(k)$. Tu činjenicu možemo iskoristiti za pojednostavljenje formule (2.3):

$$F(m,n) = \min_{1\le i\le L}\{\min_{K_{i-1}<k\le min(K_i,m)}\{D(m-k, n+w_i(k))\}\} = \min_{1\le i\le L}\{F_i(m,n)\} \tag{2.7}$$

Ako proširimo domenu pojedine funkcije $w_i(k)$ na domenu čitave konkavne funkcije onda se formula za $F_i(m, n)$ može još dodatno pojednostaviti:

$$\begin{aligned} F_i(m,n) &= \min_{1\le k\le m}\{D(m-k,n)+w_i(k)\} \\ &= \min\{D(m-1,n)+w_i(1), \min_{2\le k\le m}\{D(m-k,n)+w_i(k)\}\} \\ &= \min\{D(m-1,n)+w_i(1), \min_{1\le k\le m-1}\{D(m-1-k,n)+w_i(k+1)\}\} \\ &= \min\{D(m-1,n)+v_i, F_i(m-1,n)\}+u_i \end{aligned} \tag{2.8}$$

Gotovo identičnim postupkom možemo dobiti sličnu formulu za $G(m,n)$ i $G_i(m,n)$. Recimo da su afine funkcije kojima se aproksimira konkavna funkcija koja će se koristiti za kažnjavanje umetanja oblika $t_i(k) = c_i k + l_i$. Tada dobivamo formule:

$$G(m,n) = \min_{1\le i\le L}\{G_i(m,n)\} \tag{2.9}$$

$$G_i(m,n) = \min\{D(m,n-1)+l_i, G_i(m,n-1)\}+c_i \tag{2.10}$$

Konačno, ako proglasimo $F_0(m, n) = D(m - 1, n - 1) + d(a_m, b_n)$ i $F_{i+L}(m,n) = G_i(m,n)$ za izračunavanje matrice poravnavanja dobijemo vrlo jednostavnu i lako pamtljivu formulu:

$$D(m,n) = \min_{0\le i\le 2L}\{F_i(m,n)\} \tag{2.11}$$

Opisane jednadžbe pomažu nam pronaći procijepe i za umetanja i za brisanja, no ako radimo s RNA očitanjima pojava procijepa s umetanjima gotovo je nemoguća te stoga umetanja ne moramo nužno kažnjavati konkavnom funkcijom već to možemo činiti klasičnim načinom ili jednom afinom funkcijom. Stoga nam nisu potrebni svi $G_i(m, n)$, i $\in [1, L]$ već će dovoljan biti samo $G_1(m, n)$.

2.2. Pronalaženje optimalnih poravnanja

2.2.1. Prikaz optimalnih poravnanja grafom

Kada poravnavamo dva niza, kao rezultat ne moramo nužno dobiti samo jedno optimalno poravnanje. Zgodan način za prikaz svih optimalnih poravnanja je aciklički graf.

Ako matricu poravnavanja promatramo kao kartezijev koordinatni sustav lako se može vidjeti da svako poravnanje zapravo predstavlja zaseban put od koordinate (0, 0) do (M, N) gdje su M i N duljine nizova koji se poravnavaju. Dijagonalni dijelovi toga puta predstavljaju nizove zamjena (ili "pogodaka"). Horizontalni ili vertikalni dijelovi

puta predstavljaju umetanja, odnosno brisanja. Očito je da za rekonstrukciju poravnanja nije potrebno pamtiti sve koordinate puta koji ga predstavlja već je dovoljno znati samo koordinate u kojima se dogodi "skretanje" (i naravno početna i zadnja koordinata). Takav prikaz poravnanja naziva se kosturni prikaz poravnanja.

Ako su koordinate *(i, j)* i *(m, n)* susjedne u kosturnom prikazu poravnanja onda mogu biti u jednom od 3 odnosa ovisno vrijedi li:

- $i - j = m - n$
  - (i, j) -> (m, n) predstavlja niz zamjena/pogodaka (dijagonalni dio puta)
- $i = m \ \& \ j < n$
  - (i, j) -> (m, n) predstavlja niz umetanja (horizontalni dio puta)
- $j = n \ \& \ i < m$
  - (i, j) -> (m, n) predstavlja niz brisanja (vertikalni dio puta)

Gotohov algoritam u konačnici daje usmjereni graf iz kojeg se mogu izvesti sva optimalna poravnanja. Lukovi grafa predstavljaju nizove zamjena/pogodaka ili nizove umetanja/brisanja, a vrhovi predstavljaju "koordinate" u kojima ti nizovi počinju. Konačni graf sigurno ne sadrži cikluse jer za svaki luk *(i, j)* -> *(m, n)* vrijedi $i \leq m$ i $j \leq n$.

2.2.2. Reducirani oblik grafa

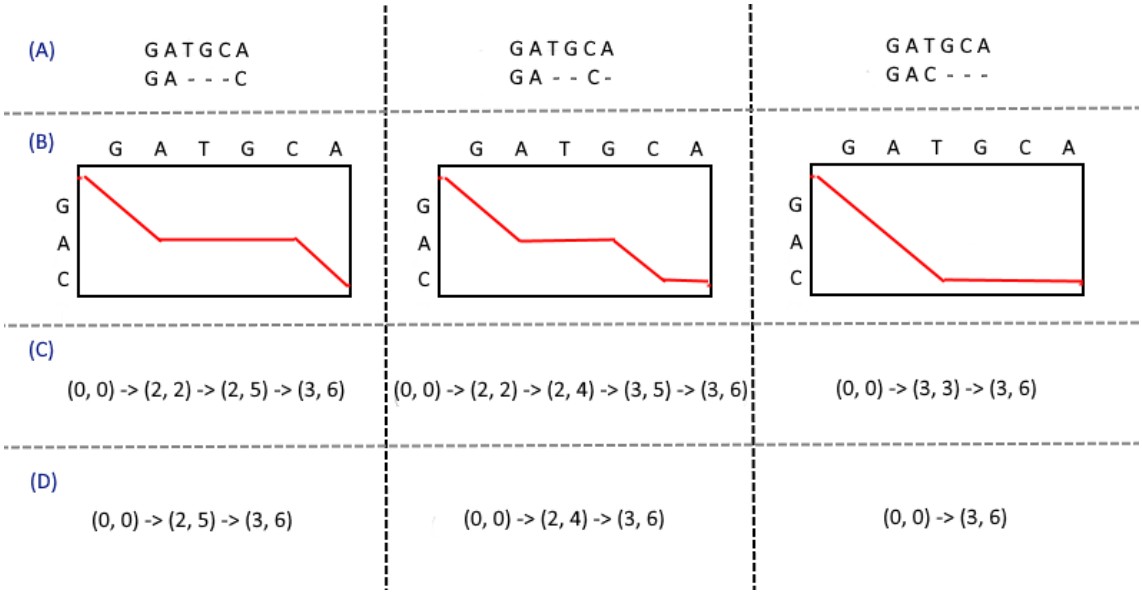
Poželjno svojstvo pri poravnavanju je da niz umetanja nikad nije praćen nizom brisanja i obrnuto. Time također osiguravamo da se u putevima koji predstavljaju poravnanja nikad ne pojave horizontala praćena vertikalom ili vertikala praćena horizontalom. To svojstvo lako je osigurati ako se postavi da je $u_L \geq \max\{d(x, y)\} / 2$.

U slučaju da je navedeno svojstvo zadovoljeno ne moramo pamtit sve koordinate kosturnog prikaza poravnanja nego je dovoljno pamtiti reducirani oblik puta. Recimo da je *(i, j)* -> *(k, l)* dijagonalni dio puta, a *(k, l)* -> *(m, n)* horizontalni ili vertikalni dio. Tada možemo koordinatu *(k, l)* "rekonstruirati" iz koordinata *(i, j)* i *(m, n)* ovisno o njihovom odnosu:

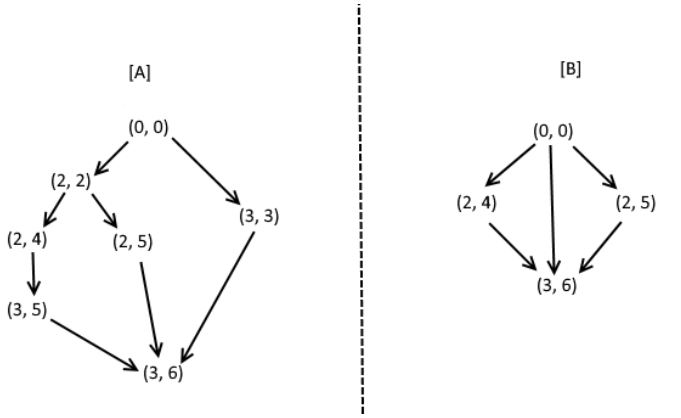
- $i - j > m - n$
  - $k = m$
  - $l = m - i + j$
- $i - j < m - n$
  - $k = n + i - j$
  - $l = n$

2.2.3. Primjer

Kao primjer uzmimo nizove "GAT" i "GATGCA". Neka je, radi jednostavnosti, konkavna funkcija aproksimirana samo s jednom afinom funkcijom i neka su parametri te afine funkcije $u_1 = v_1 = 1$, a funkcija mjere sličnosti d neka je definirana tako da vraća 0 za jednake znakove i 1 za različite. Poravnavanjem ova dva niza zadanim parametrima dobiti ćemo tri optimalna poravnanja kako je prikazano na slici 2.2. Gotohov algoritam kao konačan rezultat daje reducirani graf na slici 2.3.



Slika 2.2: Optimalna poravnanja nizova "GATGCA" i "GAC". (A) prikazuje optimalna poravnanja, (B) prikazuje odgovarajuće puteve tih poravnanja u matrici, (C) prikazuje kosturni prikaz puteva, a (D) prikazuje kako izgledaju reducirani oblici tih puteva.

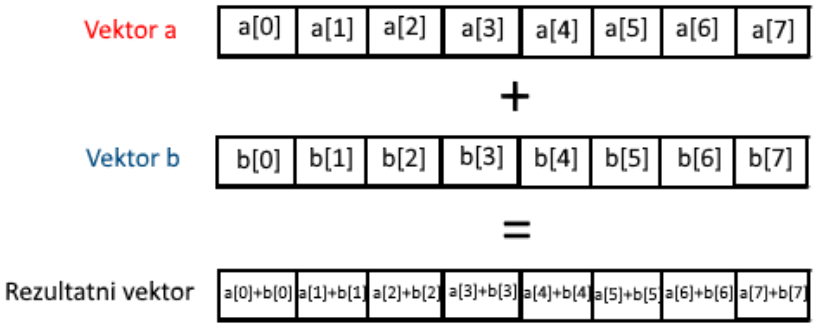


Slika 2.3: Konačan graf optimalnih poravnanja nizova "GATGCA" i "GAC". [A] predstavlja nereduciranu kosturnu reprezentaciju svih optimalnih poravnanja, a [B] prikazuje reducirani oblik konačnog grafa.

3. Vektorizacija

Vektorizacija je zasnovana na principu SIMD (Single Instruction, Multiple Data) računala i omogućava istodobno izvršavanje često korištenih operacija kao što je zbrajanje ili oduzimanje nad određenim skupom podataka. Vektorizacija je podržana na većini modernih procesora.

Recimo da želimo zbrojiti elemente nekih polja *a* i *b*. Ukoliko bi to radili kao i inače elemente bismo zbrajali jednog po jednog. Vektorizacija, s druge strane, taj proces ubrzava tako da se polja *a* i *b* prvo učitaju u posebne registre u procesoru iz kojih zatim sve elemente polja može zbrojiti u jednom potezu i onda rezultantni vektor spremiti na željenu memorijsku lokaciju. Ukoliko je duljina polja veća od duljine vektora (na primjer: vektor može primiti 8 cijelih brojeva, a radimo s poljima duljine 14) tada navedeni postupak ponavljamo obrađujući po onoliko elemenata koliko nam stane u vektor sve dok ne dođemo da kraja polja.



Slika 3.1: Ilustracija zbrajanja vektora *a* i *b*

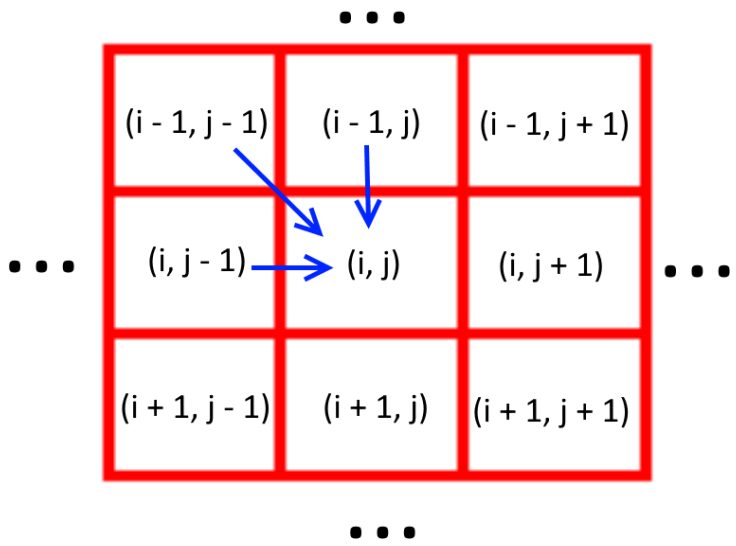
Listing 3.1: Usporedba zbrajanja dvaju polja duljine 8 elemenata u programskom jeziku C++ na uobičajen način (petljom) i vektorizacijom (AVX2)

```
void add_arrays_loop (const int* a, const int* b, int* c) {
    for (int i=0; i<8; i++) {
        c[i] = a[i] + b[i];
    }
}

void add_arrays_vectorized (const int* a, const int* b, int* c) {
    __m256i vector_a = _mm256_loadu_si256((__m256i *)&a[0]);//a -> vector_a
    __m256i vector_b = _mm256_loadu_si256((__m256i *)&b[0]);//b -> vector_b
    __m256i vector_c = _mm256_add_epi32(vector_a, vector_b);//v_c = v_a + v_b
    _mm256_storeu_si256((__m256i *)&c[0], vector_c);//vector_c -> c
}
```

3.1. Primjena vektorizacije pri poravnavanju nizova

Sama vektorizacija nam može uvelike ubrzati proces poravnavnja time da ubrza "popunjavanje" matrice poravnavanja. Ako gledamo pojedinu ćeliju matrice možemo primjetiti da ona ovisi samo o 3 susjedne ćelije. Neka su "koordinate" ćelije čiju vrijednost računamo *(i, j)*. Tada vrijednost te ćelije ovisi o ćelijama $(i-1, j)$, $(i-1, j-1)$ i $(i, j-1)$.



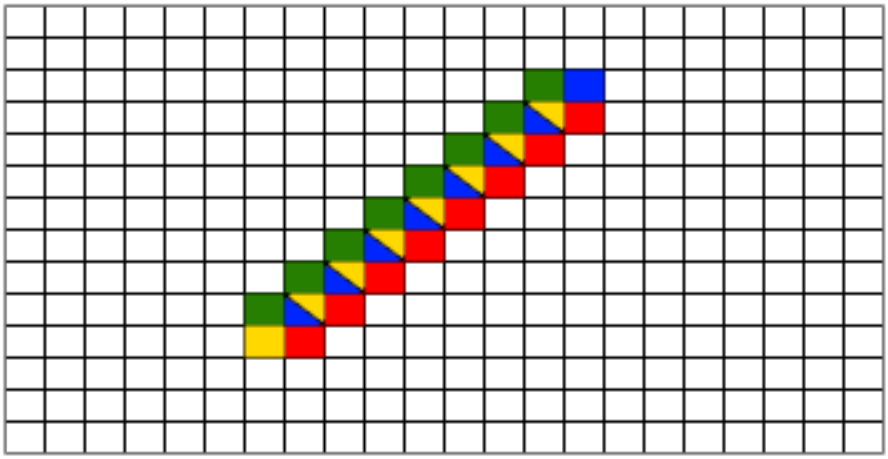
Slika 3.2: Izračunavanje matrične ćelije *(i, j)*

Zbog tog svojstva moguće je provesti vektorizaciju po sporednim dijagonalama

(nadalje ćemo pod pojmom dijagonala misliti na sporedne dijagonale). Numerirajmo dijagonale tako da zbroj koordinata svake ćelije odgovara "indeksu" normale kojoj pripada. Tako na primjer ćelija *(4, 1)* pripada petoj dijagonali, *(0, 0)* nultoj itd. Za izračunavanje vrijednosti ćelija na *k*-toj dijagonali moramo poznavati vrijednosti ćelija na *(k - 1)*-toj i *(k - 2)*-toj dijagonali. To se može lako i dokazati. Ako izračunavamo ćeliju *(i, j)* i ako znamo da se ona nalazi na *k*-toj dijagonali, onda vrijedi $i + j = k$. Već od prije nam je poznato da za izračunavanje *(i, j)* ćelije moramo poznavati vrijednosti u ćelijama *(i - 1, j)* i *(j, i - 1)* koje se nalaze na *(k - 1)*-toj dijagonali jer je $i - 1 + j = i + j - 1 = k - 1$ te *(i - 1, j - 1)* koja je na *(k - 2)*-toj dijagonali zbog jednakosti $i - 1 + j - 1 = i + j - 2 = k - 2$.

Za početak je potrebno nekoliko dijagonala inicijalizirati nevektoriziranim postupkom kako bi vektorizacijsko popunjavanje zapravo imalo "ishodište" iz kojega može krenuti. Početne dijagonale su kratke i stoga taj postupak inicijalizacije nije vremenski skup.

Na slici (3.3) prikazano je koji su dijelovi matrice "aktivni" pri izračunavanju pojedinog segmenta određene dijagonale. Recimo da izračunavamo segment *k*-te dijagonale koji počinje s ćelijom *(i, j)* pri običnom poravnanju koje ima definiranu kaznu umetanja, zamjene i brisanja te nagradu pogotka. Prvo što ćemo učiniti je učitati segment *(k - 1)*-te dijagonale koji počinje ćelijom *(i, j - 1)* (na slici (3.3) označeno žutom bojom) u vektor. Tome vektoru zatim pribrojimo vektor čiji su svi elementi jednaki kazni brisanja i rezultat te operacije spremimo u vektor koji predstavlja segment dijagonale koji trenutno računamo. Nakon toga segment koji smo učitali s dijagonale *(k - 1)* "pomičemo" za jedan udesno, to jest u vektor učitavamo segment s te iste dijagonale, ali koji počinje ćelijom (i - 1, j) (na slici (3.3) označeno plavom bojom). Taj vektor tada zbrojimo s vektorom popunjenim kaznom za umetanje i dobiveni vektor uspoređujemo s ciljnim vektorom te u ciljni vektor spremimo one vrijednosti koje su manje. Sličnu stvar radimo i u zadnjem koraku samo što sada učitavamo segment dijagonale *(k - 2)* koji počinje ćelijom *(i - 1, j - 1)* (na slici (3.3) označeno zelenom bojom) i pridodajemo mu kaznu/nagradu zamjene/pogotka. Postupak ponavljamo sve dok ne dođemo do kraja dijagonale, nakon čega krećemo s izračunavanjem dijagonale *(k + 1)*.



Slika 3.3: Vektorizacija pri popunjavanju matrice poravnavanja. Crvenom bojom označen je vektor koji se trenutno računa, a plavom, žutom i zelenom označeni su vektori iz kojih će se izračunati crveni. Plavi vektor služi za računanje vrijednosti koja dolazi od brisanja, žuti za izračunavanje vrijednosti koja dolazi od umetanja i zeleni za računanje vrijednosti koja dolazi kao rezultat pogotka/zamjene. Na kraju se u crveni vektor sprema najmanja od te 3 vrijednosti.

4. Implementacija

U sklopu ovog završnog rada razvijen je alat OSALG (Optimal Sequence Aligner allowing for Long Gaps) koji je namijenjen za poravnavanje RNA očitanja. OSALG nudi implementaciju klasičnog Gotohovog algoritma te vektoriziranu inačicu poravnavanja s velikim procijepima.

4.1. Implementacija Gotohovog algoritma

Za implementaciju reduciranog usmjerenog grafa opisanog u prošlom poglavlju, OSALG koristi ideju iz Gotohovog rada [7], gdje se graf prikazuje s pomoću dvije liste: primarne i sekundarne. Recimo da poravnavamo neka dva niza duljina M i N te da je konkavna funkcija kažnjavanja brisanja aproksimirana sa L afinih funkcija.

Osnovna ideja je da, prolazeći matricom, u primarnoj listi pamtimo koordinate ćelija koje predstavljaju završetak dijagonalnog segmenta na putu nekog poravnanja (koje ne mora nužno biti optimalno) gledano iz koordinate (M, N) jer su nam upravo takve koordinate kandidati da budu vrhovi reduciranog grafa kojega tražimo. Također ćemo uz te koordinate pamtiti i poseban pokazivač $P_i(m, n)$ (za koordinatu (m, n)) gdje je $i \in [0, L + 1]$ i ovisi o optimalnom smjeru iz te koordinate. Ti "pokazivači" su zapravo parovi cjelobrojnih vrijednosti oblika (p, q) koji označavaju indeks elementa u primarnoj(i sekundarnoj, ali do toga ćemo doći malo kasnije) listi i izračunavati ćemo ih kako iteriramo po matrici. Da bi malo bolje razjasnili što su ovi pokazivači i čemu služe, pogledajmo primjer. Neka smo na primjer došli do koordinate (x, y) u matrici. Ta koordinata leži na putu nekog poravnanja dvaju zadanih nizova. To poravnanje ne mora nužno biti optimalno jer to, u trenutku računanja vrijednosti vezanih za koordinatu (x, y), ni ne možemo znati jer matricu izračunavamo redak po redak. Na prvi pogled bi mogli misliti da je dovoljno znati samo gdje vodi najoptimalniji smjer iz te koordinate. No to nije slučaj jer se informacije o tome gdje vode ostali smjerovi (koji nisu optimalni) koriste pri izračunavanju smjerova iz susjednih koordinata. Valja za naglasiti da se ovdje pod pojmom smjera misli na iduću koordinatu na koju ćemo

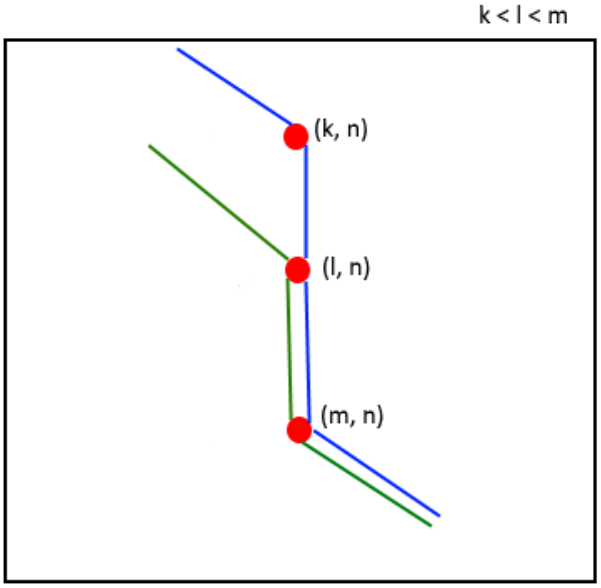
naići prateći put poravnanja koja je ujedno i završetak nekog dijagonalnog segmenta tog puta. Takva koordinata će zasigurno biti u primarnoj listi jer matricu iteriramo po retcima, a u primarnu listu dodajemo baš takve koordinate pa u pokazivač stavljamo vrijednost indeksa primarne liste na kojem se nalazi taj element. Radi jednostavnosti, recimo da je u ovom primjeru $L = 2$. Da bi zapamtili gdje vodi horizontalni put (umetanje) koristimo pokazivač $P_3(x, y)$ ($P_{L+1}(x, y)$), a za dijagonalni smjer koristimo $P_0(m, n)$. Kod vertikalnog smjera, situacija se pomalo komplicira kako brisanja kažnjavamo aproksimiranom konkavnom funkcijom. Rezultat toga je da nam vertikalni smjer zapravo predstavlja dva "smjera" ovisno o tome je li kada krenemo iz koordinate vertikalno gore naiđemo na procjep (smjer određen sa $P_2(x, y)$) ili na "regularna" brisanja (smjer određen sa $P_1(x, y)$).

Uz primarnu, koristimo i sekundarnu listu. Glavna uloga sekundarne liste je pobrinuti se za grananja optimalnih puteva kao što je prikazano na slici (4.1). Ako uzmemo primjer prikazan na slici, vidljivo je da, kada bi koristili samo primarnu listu bez ikakvih dodatnih lista, prilično nezgodno bi bilo zapamtiti oba prikazana poravnanja (puta). Stoga u sekundarnu listu dodajemo pokazivač na element u primarnoj listi koji nam predstavlja jedan put, a pokazivač koji imamo već u primarnoj listi (uz koordinatu (m, n)) pokazivat će na drugi put (na neki drugi indeks primarne liste) te na prethodno spomenuti element iz sekundarne liste. Time ćemo se iz elementa primarne liste koji čuva (m, n) koordinatu moći "povezati" i na jedan i na drugi put.

Što se implementacijskih detalja tiče, primarna lista sastoji se od "SAVE1" strukture podataka koja predstavlja uređenu četvorku podataka (*m, n, p, q*). Uređeni par (*m, n*) označava koordinatu ćelije u kojoj se događa "skretanje" poravnanja, a *p* služi kao pokazivač na element u primarnoj listi i njime se zapravo realiziraju lukovi u grafu. Element četvorke *q* je pokazivač na element sekundarne liste sastavljene od "SAVE2" strukture podataka koja je oblika (p, q) gdje je *p* pokazivač na element primarne liste, a *q* pokazivač na element sekundarne liste. Ukoliko neki od pokazivača poprimi vrijednost -1, to znači da ne pokazuje na ništa.

Algoritam (1) pokazuje pseudokod Gotohovog algoritma. $F_i(m, n)$ označava "vrijednost pojedinog smjera". Tako na primjer $F_0(m, n)$ predstavlja vrijednost koju stvara dijagonalan smjer (iz ćelije *(m-1, n-1)*), $F_i(m, n)$ gdje je $i \in [1, L]$ predstavlja vrijednost vertikalnog smjera (iz ćelije *(m - 1, n)*) i $F_{L+1}(m, n)$ vrijednost horizontalnog smjera. Kao $D(m, n)$ uzima se $\min_{0 \leq i \leq L+1} \{F_i(m, n)\}$. Sukladno tome se postavljaju *boolean* vrijednosti $E_i(m, n)$ ovisno o tome je li $F_i(m, n) = D(m, n)$ za neki *i*.

Kao što je ranije rečeno, u primarnu listu se dodaje koordinata (m, n) ukoliko ona predstavlja početak dijagonalnog segmenta (krenuvši iz koordinate (0, 0)) na putu po-



Slika 4.1: Segment puta poravnanja u matrici. Plava i zelena boja predstavljaju dva različita optimalna poravnanja.

ravnanja. U takvom slučaju lako je zaključiti da vrijedi sljedeće: $E_0(m+1, n+1) = true$ i $E_i(m, n) = true$ za neki $i > 0$. Za dodavanje u primarnu listu brine se funkcija *adr* koja se ujedno koristi i za ažuriranje vrijednosti pokazivača $P_0(m+1, n+1)$.

Funkcija *link* s druge strane, brine se za spajanje puteva iz istoga smjera. Recimo na primjer da imamo dva puta $(k, n) -> (m, n)$ i $(l, n) -> (m, n)$ gdje je $k < l < m$ i da su koordinate (k, n) i (l, n) koordinate u kojima se događa "skretanje puta" za neko optimalno poravnanje. U navedenom primjeru znamo da vrijedi:

$$D(m, n) = F_i(m, n) = D(k, n) + w_i(m - k) = D(l, n) + w_i(m - l) \tag{4.1}$$

Na sličan način možemo doći i do formule za $F_i(l, n)$:

$$F_i(l, n) = D(k, n) + w_i(l - k) \tag{4.2}$$

Iz formula (4.1) i (4.2) možemo izvesti da vrijedi:

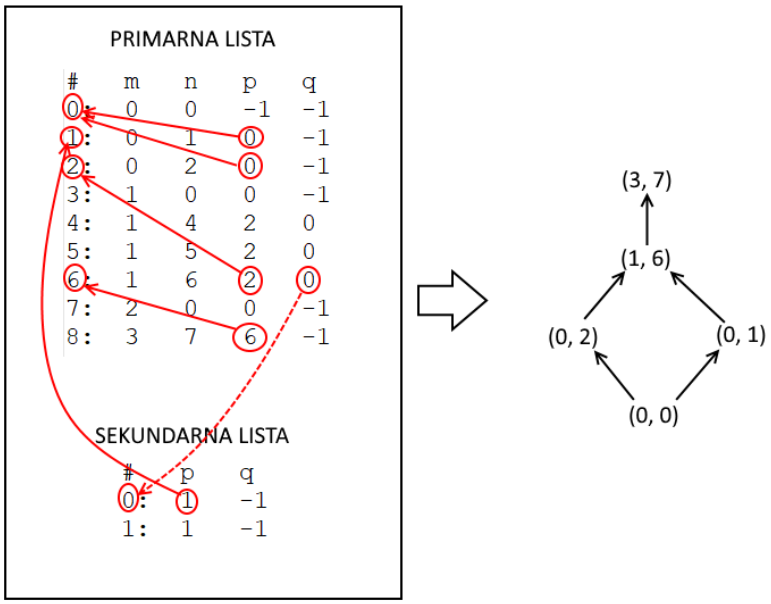
$$D(l, n) = D(k, n) + w_i(m-k) - w_i(m-l) = D(k, n) + u_i * (l-k) = F_i(l, n) - v_i \tag{4.3}$$

Iz čega dobivamo konačnu jednakost:

$$F_i(l, n) = D(l, n) + v_i \tag{4.4}$$

To znači da ćemo pri svakoj koordinati *(l, n)* provjeravati je li zadovoljena formula (4.4). U slučaju da je jednakost zadovoljena, zvat ćemo funkciju *link* koja će se pobrinuti da se eventualna grananja grafa spreme u sekundarnu listu i povežu sa postojećim vrhovima.

U konačnici dobivamo popunjenu primarnu i sekundarnu listu koje nam predstavljaju konačni reducirani graf iz kojega se mogu izvesti sva optimalna poravnanja (Slika (4.2)). Ako na primjer u primarnoj listi imamo spremljenu četvorku (5, 1, 3, 0) i neka koordinata (5, 1) leži na putu optimalnih poravnanja. To znači da je u grafu koordinata (5, 1) povezana s koordinatom koja je na indeksu 3 primarne liste te koordinatom na koju pokazuje element sekundarne liste na indeksu 0.



Slika 4.2: Reprezentacija grafa (koji se dobije kao rezultat poravnavanja nizova "GAT" i "CGGCTAA") preko dviju lista

Kod OSALG-a konkavna funkcija je aproksimirana s dvije afine funkcije ($L = 2$), a parametri aproksimacije iznose $u_1 = 4$, $v_1 = 1$, $u_2 = 2$ i $v_2 = 13$. Pogotci se nagrađuju vrijednošću -2, a zamjene se kažnjavaju vrijednošću 4.

Zanimljivo je da se graf optimalnih poravnavanja gradi kako punimo matricu i stoga ne trebamo čuvati cijelu matricu u memoriji već je dovoljno pamtiti samo dva retka matrice: redak koji trenutno računamo i redak prije njega jer time imamo sve potrebne informacije za izračunavanje pojedinog elementa matrice.

Algorithm 1 Gotohov algoritam

$primary\_list.add(0, 0, -1, -1)$
for $m = 1; m \leq M; m = m + 1$ **do**
  for $n = 1; n \leq N; n = n + 1$ **do**
    for $i = 1; i \leq L; i = i + 1$ **do**
      $F_i(m, n) = \min(D(m-1, n) + v_i, F_i(m-1, n)) + u_i$
      if $D(m-1, n) + v_i < F_i(m-1, n)$ **then**
        $P_i(m, n) = P_0(m-1, n)$
      else if $D(m-1, n) + v_i == F_i(m-1, n)$ **then**
        $P_i(m, n) =$ link$(m-1, n, i)$
      else
        $P_i(m, n) = P_i(m-1, n)$
      end if
    end for
    $F_{L+1}(m, n) = \min(D(m, n-1) + v_{ins}, F_{L+1}(m, n-1)) + u_{ins}$
    if $D(m, n-1) + v_{ins} < F_{L+1}(m, n-1)$ **then**
      $P_{L+1}(m, n) = P_0(m, n-1)$
    else if $D(m, n-1) + v_{ins} == F_{L+1}(m, n-1)$ **then**
      $P_{L+1}(m, n) =$ link$(m, n-1, L+1)$
    else
      $P_{L+1}(m, n) = P_{L+1}(m, n-1)$
    end if
    $F_0(m, n) = D(m-1, n-1) + d(a_m, b_n)$
    $D(m, n) = \min(F_i(m, n)\{i \in [0, L+1]\})$
    for $i = 0; i \leq L+1; i = i + 1$ **do**
      $E_i(m, n) = (D(m, n) == F_i(m, n))$
    end for
    if $E_0(m, n)$ and $E_i(m-1, n-1)$ for some i > 0 and not $E_0(m-1, n-1)$ **then**
      $P_0(m, n) =$adr$(m-1, n-1)$
    else
      $P_0(m, n) = P_0(m-1, n-1))$
    end if
  end for
end for
$P_{last} =$adr(M, N)

Algorithm 2 Funkcija adr(m, n)

P_{result} = (primary_list.size(), -1)
for $i = 0; i \leq L + 1; i = i + 1$ **do**
  if $E_i(m, n)$ **then**
    add_into_primary_list($(m, n, P_i(m, n))$)
  end if
end for
return P_{result}

Algorithm 3 Funkcija link(m, n, i)

P_{result} = ($P_0(m, n).p$, secondary_list.size())
add_into_secondary_list($P_i(m, n)$)
return P_{result}

4.2. Implementacija vektorizacije

OSALG implementira vektorizaciju poravnavanja opisanu u prethodnom poglavlju. No kako u tom postupku matricu punimo dijagonalu po dijagonalu, javlja se problem što u običnoj matrici poravnavanja ćelije koje su uzastopne na određenoj dijagonali (primjerice *(i, j)*, *(i - 1, j + 1)*) nisu na uzastopnim memorijskim lokacijama. Budući da bismo prvo morali segment dijagonale učitati u nekakvo privremeno polje i tek nakon toga u vektor, operacije učitavanja i spremanja vektora u memoriju pomalo su nezgodne. Takav je proces vremenski skup i upitno je bi li vektorizacija uopće pripomogla ubrzanju izvođenja. Zbog navedenog problem, prisiljeni smo transformirati matricu poravnavanja u oblik u kojemu će ćelije dijagonale biti na uzastopnim memorijskim lokacijama.

To ćemo učini tako da stvorimo matricu u kojoj retci predstavljaju dijagonale izvorne matrice (Primjer je slika (4.3)). No tu nam se javlja novo pitanje: kolike su dimenzije takve matrice?

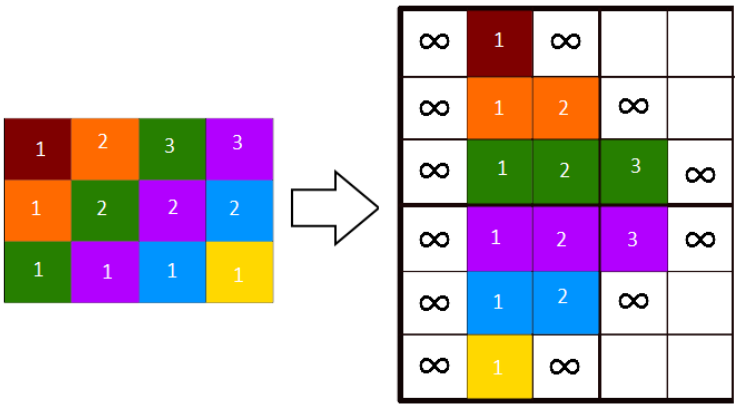
U prošlom poglavlju smo definirali da neka ćelija matrice *(i, j)* pripada *(i+j)*-toj dijagonali što znači da matrica dimenzija $(M + 1)x(N + 1)$ ima *M + N + 1* dijagonala (jer je ćelija *(M, N)* na *(M + N)*-toj dijagonali, a kako dijagonale numeriramo od 0, indeksu posljednje dijagonale dodajemo jedan da dobijemo ukupan broj dijagonala). Time smo riješili problem broja redaka željene matrice.

Pronaći broj stupaca nove matrice nešto je složeniji problem jer duljine dijagonala variraju. Tako je na primjer nulta dijagonala duga jednu ćeliju, peta dijagonala

6 ćelija itd. Očito je da bi bilo zgodno imati nekakvu relaciju koja povezuje indeks dijagonale i duljinu te iste dijagonale. Recimo da imamo neku matricu dimenzija $(M+1)x(N+1)$ te tražimo duljinu *i*-te dijagonale. Tada vrijedi:

$$LEN(i) = \begin{cases} i+1 & , i < \min\{M+1, N+1\} \\ \min\{M+1, N+1\} & , \min\{M+1, N+1\} \leq i < \min\{M+1, N+1\} + |M-N| \\ M+N+1-i & , \text{inače} \end{cases} \tag{4.5}$$

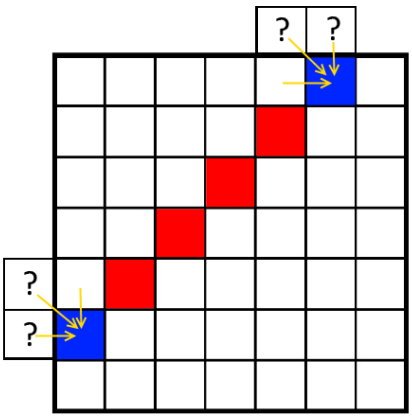
Iz prethodno navedene formule lako se može izvesti da je maksimalna duljina dijagonale $\min\{M+1, N+1\}$. Na prvi pogled mogli bi reći kako taj broj predstavlja broj stupaca vektorizacijske matrice, no to nije slučaj jer, kako radimo s vektorima, vrijednosti učitavamo grupno. Primjerice, ako je vektor duljine 8 elemenata, a $\min\{M+1, N+1\}$ iznosi 21, problem će se javiti kod učitavanja posljednjeg vektora (segmenta) dijagonale jer pri tome možemo "zagaziti" u illegalan memorijski prostor zbog toga što 21 nije djeljiv s 8. To ćemo spriječiti tako da $\min\{M+1, N+1\}$ "zaokružimo" na prvi broj koji je veći od njega, a djeljiv s duljinom vektora. Taj broj ćemo još uvećati za dva (iz razloga koji će biti objašnjeni kasnije) i time konačno dobivamo broj stupaca vektorizacijske matrice. Ukoliko $\min\{M+1, N+1\}$ nije djeljiv s duljinom vektora, broj stupaca možemo smanjiti za 1 jer u tom slučaju zadnji stupac nećemo koristiti.



Slika 4.3: Pretvorba klasične matrice poravnavanja dimenzija 4x3 u oblik pogodan za vektorizaciju. Radi jednostavnosti, veličina vektora je 2

Pri izračunavanju dijagonale vektorizacijom nejasna je situacija s rubnim ćelijama dijagonale (ćelije oblika *(0, i)* ili *(i, 0)*) jer one imaju samo jednog susjeda koji utječe

na njih, dok sve ostale ćelije imaju tri. Taj problem se može riješiti na vrlo jednostavan način tako da se da početku i na kraju svake dijagonale dodaju ćelije koje sadrže izuzetno veliki cijeli broj (prikazano kao beskonačnost na slici (4.2)). Time će se osigurati da i rubne ćelije imaju tri susjeda od kojih dva zapravo neće imati nikakav utjecaj pri izračunu pošto efektivno sadrže beskonačnost. Ovakav pristup je prihvatljiv ukoliko radimo globalno poravnavanje. U slučaju da želimo izvršiti poluglobalno ili lokalno poravnavanje, rubne ćelije je potrebno postaviti "ručno" jer se u tom slučaju traže posebni početni uvjeti.



Slika 4.4: Pri vektorizacijskom izračunavanju vrijednosti ćelija dijagonale posebno problematično može biti izračunavnje rubnih ćelija (označeno plavom bojom) pošto njima "nedostaju" dva susjeda koja utječu na vrijednost ćelije

OSALG radi s vektorima duljine 256 bita i za vrijednost ćelija koristi *short int* (2 bajta) ili *int* (ovisno o načinu rada) što znači da u jedan vektor stane 16, odnosno 8 vrijednosti. Za ostvarenje vektorizacije korištena je AVX2 (Advanced Vector Extensions 2) tehnologija budući da ona, za razliku od AVX-a i većine ostalih vektorizacijskih biblioteka, omogućava detaljniji rad s cijelim brojevima. Za poravnavanje vektorizacijom, OSALG koristi tri matrice.

5. Rezultati i diskusija

Kako bi otkrili kolika je dobra točnost i brzina OSALG-a, potrebno ga je usporediti s trenutno najpopularnijim alatom za poravnavanje s dugačkim procijepima: KSW2. Ispitivanja su izvršena na računalu s procesorom Intel Core i5-6200u@ 2.3GHz i s 8GB RAM. Podaci na kojima se izvršavaju testovi su sekvencirana RNA očitanja varirajućih duljina (100-3000).

5.1. Ispitivanje točnosti

Kako bi ispitali točnost alata, potrebno je provjeriti koliko se duljine i pozicije procijepa koje daje OSALG podudaraju s onima koje kao rezultat daje KSW2. Zbog toga ćemo napisati skriptu koja će locirati procijepe (označeni s znakom *N*) te za svaki procijep odrediti koliko je on "udaljen" od prethodnog procijepa (ili početka CIGAR-a ako je procijep prvi) tako da prebroje podudaranja/zamjene i brisanja koja mu prethode.

OSALG i KSW2 uspoređeni su na 50 testnih sekvenci. U tablici (5.1) predstavljeni su rezultati usporedbe.

Iz rezultata je vidljivo da nevektorizirana inačica OSALG-a daje nešto točnije rezultate nego vektorizirana inačica. Na prvi pogled moglo bi se reći da vektorizacija na nekim nizovima daje poprilično netočne rezultate jer nalazi više procijepa nego što ih stvarno ima, no ako se malo bolje pogleda može se vidjeti da je odmak između susjednih procijepa jako mali. To znači da vektorizirani OSALG ponekad "razbija" veće procijepe na više manjih.

Kako OSALG pri vektorizaciji nudi mogućnost korištenja vektora duljine 8 ili 16, vrijedi za naglasiti kako pri poravnavanju dugačkih nizova moramo koristiti vektore duljine 8 inače lako može doći do "prelijevanja"(*overflow*) što će rezultirati netočnim rezultatom.

Tablica 5.1: Usporedba rezultata KSW2, OSALG (vektorizirani) i OSALG (nevektorizirani). Svaki redak tablice predstavlja rezultat poravnavanja određene sekvence. Rezultati su prikazani u obliku <redni broj procijepa>: (<duljina procijepa>, <odmak procijepa>)

KSW2	**OSALG (vektorizacija)**	**OSALG (bez vektorizacije)**
1: (36554, 1417)	1: (36554, 1420)	1: (36552, 1417)
1: (36554, 308)	1: (36554, 314)	1:(36554, 309)
1: (36554, 82)	1: (36554, 87)	1:(36554, 82)
1: (36554, 132)	1: (6681, 139) 2: (29874, 6)	1:(36554, 134)
1: (1841, 875)	1: (128, 881) 2: (1715, 8)	1:(1839, 878)
1: (36554, 1056)	1: (1838, 1064) 2: (4104, 6) 3: (30609, 8)	1:(33476, 1064) 2: (3070, 7)
1: (16114, 450)	1: (16114, 452)	1:(16114, 450)

5.2. Ispitivanje brzine izvođenja

Što se tiče brzine izvođenja, logično je za očekivati da vektorizacija ubrza izvođenje programa. Ostaje samo pitanje koliko. Brzina izvođenja OSALG-a ispitana je na 7649 nizova, a rezultati izvođenja su prikazani u tablici (5.2).

Iz rezultata je vidljivo da je vektorizacija ubrzala poravnanje oko tri puta što je i bilo za očekivati s obzirom da se vektorizacijom obavlja više operacija istovremeno.

Međutim, zanimljivo je za primjetiti kako nismo dobili veliko ubrzanje kada veličinu vektora povećamo s 8 na 16. Ubrzanje, naravno, postoji, ali s obzirom da smo vektor čak dvostruko povećali, ubrzanje od 5-6 sekundi je poprilično malo. Jedan od glavnih razloga je što povećanjem veličine vektora povećavamo i broj "nepotrebnih" računanja na rubnim vektorima. Tako ćemo primjerice za neku dijagonalu čija duljina dijeljena s 16 daje 1, na rubu imati vektor čiji je zapravo samo jedan u element u matrici, dok je ostalih 15 elemenata van nje.

Tablica 5.2: Usporedba brzine izvođenja vektorizacijske i nevektorizacijske inačice alata OSALG (5 pokretanja)

	OSALG (vektor veličine 16)	**OSALG (vektor veličine 8)**	**OSALG (bez vektorizacije)**
1	59.526 s	72.115 s	262.809 s
2	60.712 s	64.727 s	260.533 s
3	61.455 s	65.547 s	251.418 s
4	61.919 s	66.183 s	255.391 s
5	58.511 s	68.362 s	257.301 s

Tablica 5.3: Vremena izvođenja alata KSW2 s podacima kojima je testirana i vremenska učinkovitost OSALG-a (5 pokretanja)

	KSW2
1	33.661 s
2	33.476 s
3	37.413 s
4	30.155 s
5	35.608 s

6. Zaključak

Ovim radom smo istražili kako poravnavati RNA očitanja kod kojih se mogu pojaviti dugački procijepi.

Vidjeli smo kako Gotohov algoritam rješava problem procijepa i kako može na zgodan način pronaći sva optimalna poravnanja tako da konačno rješenje prikaže u obliku usmjerenog grafa.

Također smo istražili i kako se vektorizacija može upotrijebiti za ubrzanje procesa poravnavanja. Zaključili smo da će nam vektorizacija po dijagonalama dati najveće ubrzanje i zbog toga smo morali učiniti određene modifikacije matrice poravnavanja.

Razvili smo biblioteku koja omogućava poravnavanje nizova Gotohovim algoritmom, a nudi i mogućnost ubrzanja poravnavanja vektorizacijom. Kao veličinu vektora koristi 256 bita što znači da u vektor stane 8 ili 16 elemenata ovisno o tipu podataka koji se koristi za čuvanje numeričkih vrijednosti (*int* ili *short int*). Za realizaciju vektorizacije odabrana je AVX2 biblioteka. Osim AVX2, na raspolaganju smo imali biblioteke SSE, SSE4.1, SSE4.2, itd. no kako one podržavaju samo rad s realnim brojevima, odlučili smo se za AVX2 koji može raditi i s cijelim brojevima.

Točnost biblioteke smo usporedili s trenutno najpoznatijim alatom za poravnavanje s dugačkim procijepima: KSW2. Nevektorizirana inačica se pokazala kao nešto točnija, ali i vektorizirana inačica daje zadovoljavajuće rezultate samo što ponekad ima sklonost "razbiti" veće procijepe na više manjih. Što se tiče brzine, vektorizacija je očekivano ubrzala brzinu izvođenja (oko 3 puta) u odnosu na nevektoriziranu inačicu, ali povećanjem veličine vektora s 8 na 16 elemenata nismo dobili drastično veće ubrzanje ponajviše iz razloga što se povećanjem veličine vektora povećava i broj računanja elemenata izvan matrice.

Vektorizacija je relativno neistraženo područje u bioinformatici, ali je korisna tehnologija za poboljšavanje performansi alata. Nadamo se da će se OSALG pokazati kao koristan alat za korištenje i da će u budućnosti pripomoći u razvoju bioinformatičkih alata s vektorizacijskim tehnologijama.

LITERATURA

[1] Bioinformatics: Gotoh Algorithm. URL http://anythingtutorials.blogspot.com/2016/04/bioinformatics-gotoh-algorithm.html.

[2] Intel intrinsics guide, . URL https://software.intel.com/sites/landingpage/IntrinsicsGuide/.

[3] Fundamentals of AVX Programming, . URL https://blog.triplez.cn/avx-avx2-learning-notes/.

[4] SIMD vectorization. URL https://arcb.csc.ncsu.edu/~mueller/cluster/ps3/SDK3.0/docs/accessibility/sdkpt/cbet_1simdvector.html.

[5] Lomont Chris. Introduction to Intel Advanced Vector Extensions. 2011.

[6] Osamu Gotoh. An improved algorithm for matching biological sequences. 1982.

[7] Osamu Gotoh. Optimal sequence alignment allowing for long gaps. 1990.

[8] Michał Pióro i Deepankar Medhi. Concave link dimensioning functions. 2004. URL https://www.sciencedirect.com/topics/computer-science/concave-function.

[9] Hajime Suzuki i Masahiro Kasahara. Acceleration of nucleotide semi-global alignment with adaptive banded dynamic programming. stranice 3–5, 2017.

[10] Hajime Suzuki i Masahiro Kasahara. Introducing difference recurrence relations for faster semi-global alignment of long sequences. stranice 34–39, 2018.

[11] Mirjana Domazet-Lošo i Mile Šikić. *Bioinformatika*. 2013.

[12] Shinsuke Yamada i Osamu Gotoh i Hayato Yamana. Improvement in accuracy of multiple sequence alignment using novel group-to-group sequence alignment algorithm with piecewise linear gap cost. 2006.

[13] Heng Li. ksw2. URL https://github.com/lh3/ksw2.

[14] Marcela Miyazawa. Sequence alignment/linear space alignment/four russian algorithm. 2005.

[15] Matt Scarpino. Crunching numbers with AVX and AVX2. 2016. URL https://www.codeproject.com/Articles/874396/Crunching-Numbers-with-AVX-and-AVX.

[16] Hajime Suzuki. adaptivebandbench, . URL https://github.com/ocxtal/adaptivebandbench.

[17] Hajime Suzuki. diffbench, . URL https://github.com/ocxtal/diffbench.

Izgradnja biblioteke za poravnavanje parova dugačkih RNA očitanja

Sažetak

Jedan od glavnih problema kojima se bavi bioinformatika je poravnavanje bioloških sljedova. Ponekad je, kao kod poravnavanja RNA očitanja, potrebno prepoznati duge procijepe brisanja što klasični algoritmi poravnavanja nisu u stanju učiniti. Kako bi se osiguralo da se takvi procijepi prepoznaju za kažnjavanje brisanja koristi se konkavna funkcija. Kako je izračunavanje konkavne funkcije relativno skup proces, ona se aproksimira s više afinih funkcija. Najpoznatiji algoritam koja otkriva procijepe je Gotohov algoritam.

Vektorizacija je tehnologija koju podržava većina modernih procesora i omogućava da se na nekom većem skupu podataka neke operacije kao što je zbrajanje izvode istovremeno. Upravo zbog toga, vektorizacija može uvelike ubrzati postupak poravnavanja nizova.

U sklopu ovog završnog rada razvijen je alat OSALG koji nudi implementaciju Gotohovog algoritma te vektorizacijsko poravnavanje nizova s prepoznavanjem dugačkih procijepa.

Ključne riječi: bioinformatika, RNA, RNK, vektorizacija, C++, poravnavanje, AVX2, SSE, Gotoh

A Programming Library for Mapping Long RNA Reads

Abstract

One of the main problems of bioinformatics is sequence alignment. Sometimes, as when aligning RNA sequences, it is necessary to find long deletion gaps which is something most classic alignment algorithms cannot do. To make sure such gaps get found, we use concave penalty function. Calculating value of concave function in specific point is complex and slow process so we approximate it with a certain number of affine functions. Most popular algorithm allowing for long gaps is Gotoh's algorithm.

Vectorization is technology supported by most modern processors and it allows us to do simple operations such as addition on bigger data at the same time. Because of that, vectorization can efficiently speed up sequence alignment process.

We developed tool called OSALG, an implementation of Gotoh's algorithm.

Keywords: bioinformatics, RNA, vectorization, C++, alignment, AVX2, SSE, Gotoh