SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

BIOINFORMATIKA – PROJEKTNA DOKUMENTACIJA

PRONALAŽENJE VARIJANTI GENA IZ PODATAKA DOBIVENIH SEKVENCIRANJEM

Alen Štruklec

Zvonimir Kučiš

Zlatko Verk

Zagreb, siječanj 2020.

Sadržaj

1. Opis problema

Kao ulazni podatak dan je skup očitanja dobiven sekvenciranjem koji sadrži nekoliko varijanti istog gena. Koristeći algoritme globalnog, poluglobalnog i lokalnog poravnanja potrebno je implementirati vremenski i prostorno optimalno rješenje koje na temelju udaljenosti izračunate jednim od gore navedenih algoritama vrši grupiranje očitanja kako bi se otkrile varijante gena prisutne u uzorku.

2. Opis korištenog algoritma

Mjera sličnosti dvije sekvence može se odrediti na dva načina, Hammingovom udaljenosti ili udaljenošću uređivanja. Hammingova udaljenost predstavlja broj zamjena potrebnih za transformaciju jednog niza u drugi, a udaljenost uređivanja definirana je kao minimalan broj zamjena, umetanja i brisanja potrebnih za transformaciju jednog niza u drugi.
Na primjer:

- Zamjena: mOst -> mAst[1]
- Umetanje: sir -> sVir
- Brisanje: Brod -> rod

Udaljenost uređivanja može se računati na različite načine i u svakom pokušaju moguće je dobiti različito rješenje, zato su razvijeni razni algoritmi za računanje udaljenosti uređivanja od kojih ćemo neke iskoristiti u ovom projektu.

Algoritmi globalnog, poluglobalnog i lokalnog poravnanja temelje se na konceptu dinamičkog programiranja.

Dinamičko programiranje može se koristiti za rješavanje problema koji imaju optimalnu podstrukturu i preklopljenost potproblema. Koristi se tzv. *bottom-up* pristup kojim se konačno rješenje gradi od rješenja istovrsnih manje složenih problema, a kako se već izračunati podatci ne bi ponovno izračunavali, rješenja se spremaju u matricu.

[1] Primjeri preuzeti iz skripte iz bioinformatike

Na slici 1. prikazana je mreža poravnanja koja je rezultat izgradnje matrice, generirana je pomoću alata dostupnog na web-stranici sveučilišta u Freiburgu[2], a ona se gradi na sljedeći način:

Algoritam započinje definiranjem vrijednosti na rubovima (redci i stupci s indeksom 0). Element na poziciji (0, 0) poprimi vrijednost 0, elementi nultog retka poprime vrijednost d*j, a elementi nultog stupca d*i gdje d predstavlja parametar brisanja/umetanja (pomicanje desno/dolje).

Ostali elementi računaju se kao minimum pomaka desno iz lijeve ćelije (V(i, j-1) + d), pomaka dolje iz gornje ćelije (V(i-1, j) + d) i dijagonalnog pomaka koji nagrađuje ako su elementi u oba niza ćelije gore-lijevo jednaki, a kažnjava ako su različiti.

Udaljenost poravnanja je rezultat izračunat za element u zadnjem retku i stupcu (m, n).

Pomicanje u desno i dolje predstavlja pomicanje u jednom nizu, a ostanak na istom elementu u drugom nizu. Micanje po dijagonali predstavlja slaganje ili neslaganje između dva niza.

Parametri algoritma težine su kojima se kažnjava zamjena/neslaganje, preskakanje i slaganje znakova. U ovom primjeru one su jednake 1.

D		T_1	G_2	C_3	A_4	T_5	A_6	T_7
	0	1	2	3	4	5	6	7
A_1	1	1	2	3	2	3	4	5
T_2	2	0	1	2	3	1	2	3
C_3	3	1	1	0	1	2	2	3
C_4	4	2	2	0	1	2	3	3
G_5	5	3	1	1	1	2	3	4
A_6	6	4	2	2	0	1	1	2
T_7	7	5	3	3	1	-1	0	0

Slika 1. Mreža poravnanja

[2] http://rna.informatik.uni-freiburg.de/Teaching/

2.1 Algoritam globalnog poravnanja

Needleman-Wunsch-ov [1] algoritam koristi se za traženje globalnog poravnanja. Globalno poravnanje traži put od početka do kraja oba slijeda.

Algoritam izgleda gotovo identično algoritmu koji je objašnjen u prethodnom odjeljku, samo što umjesto minimuma pomaka traži maksimum.

Implementacija algoritma u programskom jeziku Python prikazana je u nastavku.

```
class NeedlemanWunsch():
    def __init__(self, scoring=[5,-3,-5]):
        self.scoring = scoring
    """
        Run Needleman-Wunsch algorithm on seqA and seqB
    """
    def run(self, seqA, seqB,):
        self.seqA = seqA
        self.seqB = seqB
        self.m_rows = len(seqA) + 1
        self.m_cols = len(seqB) + 1

        self.matrix = [[None for i in range(self.m_cols)] for i in range(self.m_rows)] # Initiating Score Matrix

        # Populate matrix
        for i in range(self.m_rows):
            self.matrix[i][0] = self.scoring[2] * i
        for j in range(self.m_cols):
            self.matrix[0][j] = self.scoring[2] * j

        # Calculate similarity
        [self.score(i, j) for i in range(1, self.m_rows) for j in range(1, self.m_cols)]

        o_score = self.matrix[i][j]
        return o_score

    """
            Calculate similarity with the maximum value from the following :
            D_(i-1)_(j-1) + MATCH     if seqA_i == seqB_i
            D_(i-1)_(j-1) + MISSMATCH if seqA_i != seqB_i
            D_(i-1)_(j)   + GAP       if seqB_i == -
            D_(i)_(j-1)   + GAP       if seqA_i == -
    """
    def score(self, i, j):
        score = self.scoring[0] if (self.seqA[i-1] == self.seqB[j-1]) else self.scoring[1]
        h_val = self.matrix[i][j-1] + self.scoring[2]
        v_val = self.matrix[i-1][j] + self.scoring[2]
        d_val = self.matrix[i-1][j-1] + score
        o_val = [h_val, d_val, v_val]  # h=1, d=2, v=3

        self.matrix[i][j] = max(o_val)
```

Slika 2. Isječak programskog koda

2.2 Polu-globalno poravnanje

U slučaju da želimo pronaći preklapanje niza s podnizom nekog drugog niza, koristimo polu-globalno poravnanje. Ono ne penalizira praznine na početku ili kraju pojedinog niza. Praznine na početku niza omogućavamo tako da ih ne penalizira prilikom inicijalizacije (nulti redak/stupac inicijaliziran s 0), a praznine na kraju niza omogućavamo tako da na kraju poravnanja uzimamo maksimalnu vrijednost u zadnjem retku ili stupcu, ovisno o tome nad kojim nizom želimo omogućiti praznine na kraju.

2.3 Lokalno poravnanje

Algoritam lokalnog poravnanja, Smith-Waterman [2], traži regije u čije će poravnanje imati najveći rezultat. Praznine na početku sljedova se ne penaliziraju (nulti redak i stupac inicijalizirani na 0) i u slučaju da vrijednost pojedinog elementa padne ispod 0, ta vrijednost se zamjenjuje s 0.

Rezultat poravnanja je najveći element u matrici i predstavlja kraj regije s maksimalnim rezultatom poravnanja sljedova.

U nastavku je prikazana implementacija algoritma lokalnog poravnanja u programskom jeziku Python.

```
class SmithWaterman():
    def __init__(self, match, mismatch, gap):
        self.match = match
        self.mismatch = mismatch
        self.gap = gap

    """
        Calculate simiraity of seqA and seqB
        Aligment 'local' or 'glocal'
    """
    def score(self, seqA, seqB, alignment='local'):

        H = np.zeros((len(seqA) + 1, len(seqB) + 1), np.int) #Initialize matrix

        """
            Calculate score with the maximum value from the following :
            D_(i-1)_(j-1) + MATCH     if seqA_i == seqB_i
            D_(i-1)_(j-1) + MISSMATCH if seqA_i != seqB_i
            D_(i-1)_(j)   + GAP       if seqB_i == -
            D_(i)_(j-1)   + GAP       if seqA_i == -
            0 (zero)
        """
        for i, j in itertools.product(range(1, H.shape[0]), range(1, H.shape[1])):
            match = H[i - 1, j - 1] + (self.match if seqA[i - 1] == seqB[j - 1] else + self.mismatch)
            gap = H[i - 1, j] + self.gap
            mismatch = H[i, j - 1] + self.gap
            #Calculate glocal without zero or local with zero
            H[i, j] = max(match, gap, mismatch, 0) if alignment == 'local' else max(match, gap, mismatch)

        return H.max() if alignment == 'local' else H[len(seqA), ].max()
```

Slika 3. Isječak programskog koda

3. Optimizacija

3.1 Ubrzanje Python-a C funkcijama

Python je relativno spor jezik, jer se ne prevodi nego interpretira u stvarnom vremenu. Zbog toga nije prigodan u primjenama u kojima nam je vrijeme izvođenja bitno. Da bi riješili taj problem koristili smo Cython. Cython nam omogućuje da pojedine dijelove koda definiramo u C jeziku te tako definirane funkcije zovemo iz Pythona.

Testiranjem i pokretanjem programa utvrdili smo da najviše vremena Python koristi na računanje sličnosti prema prije navedenim algoritmima. Iz tog razloga smo prepisali navedene algoritme u C kao zasebne funkcije te generirali Python modul koji kasnije možemo lako koristi unutar samog Python koda.

Da bi vidjeli koliko je ubrzanje samih algoritama u C-u, razvili smo sintetički test. Svaki algoritam prima dva ulazna niza znakova oba duljine 70 znakova:

“ACTGAGAGATAGAGTCAGCTACGTCGATCGACTAGCTACGATCGACTGAGAGATAGAGTCAGCTACGACG”

“ACGCTAGCATCGATCGATCGATCGATCAGTCAGCTACGATCGATCGATCGCTGCTAGCTACGATCGATTC”

Test smo izvrtjeli za globalni i lokalni algoritam. Za svaki od njih smo vrtjeli iteraciju šaljući im iste podatke, te mjerili koliko treba da petlja završi. Ispod su prikazane tablice 1. i 2. mjerenja, ubrzanje smo računali tako da smo podijelili trajanje u Python-u s trajanjem u C-u.

Broj iteracija	Python trajanje [s]	C trajanje [s]	Ubrzanje
1000	5.95	0.013	458
10000	57.29	0.131	437
100000	539.49	1.332	405

Tablica 1. Testiranje globalnog poravnavanja

Broj iteracija	Python trajanje [s]	C trajanje [s]	Ubrzanje
1000	11.24	0.016	702
10000	110.51	0.16	690
100000	1029.06	1.595	645

Tablica 2. Testiranje lokalnog poravnavanja

Vidimo da smo postigli veliko ubrzanje koristeći C funkcije umjesto Python-a. Također ubrzanje opada s više iteracija jer moramo uračunati pregrijavanje jezgre, pa samim time pad performansi računala.

Nakon što smo testirali brzinu izvođenja na različitom broju iteracija, napravili smo i test na različitim duljinama ulaza. U ovom testu nema iteracija nego samo izračun sličnosti između dva ulaza. Također imamo dvije tablice, jednu za globalni algoritam tablica 3. i tablica 4. za lokalni.

Duljina niza	Python trajanje [s]	C trajanje [s]	Ubrzanje
100	0.011	4*10e-5	275
1000	1.22	0.004	305
10000	123.62	0.48	257

Tablica 3. Testiranje različitih duljina sekvenci na globalnom algoritmu

Duljina niza	Python trajanje [s]	C trajanje [s]	Ubrzanje
100	0.021	5.5*10e-5	381
1000	2.21	0.005	442
10000	214.68	0.56	383

Tablica 4. Testiranje različitih duljina sekvenci na lokalnom algoritmu

3.2 Odbacivanje sekvenci

U ulaznoj datoteci su definirani zapisi brojnih očitanih sekvenci. Da bi ubrzali rad algoritma odlučili smo izdvojiti najzastupljenije duljine sekvenci, te odbaciti ostale. Time nam se skup podataka za obradu drastično smanjuje te ubrza izvođenja programa. Kako ne bi uzeli samo jednu duljinu pa zapravo možda odbacili neke korisne sekvence uzeli smo i one čija je duljina +/- 5 od referentne najzastupljenije duljine sekvenci.

Ako nam je ulaz datoteka J30_B_CE_IonXpress_006.fastq u njoj je zapisano 3818 sekvenci, odbacivanjem sekvenci taj broj se smanjuje na 2167. Možemo vidjeti da se broj podataka za obradu smanjio za 43% posto. To neće uvijek biti slučaj da je tako veliki postotak odbačen, ali u većini slučaja nam se isplati odbacivati sekvence radi ubrzanja.

4. Grupiranje

Nakon izvršavanja algoritma poravnavanja dobivamo matricu sličnosti. Prvo je potrebno transformirati matricu sličnosti u matricu udaljenosti. Primijenjeno je min-max skaliranje na matrici te zatim smo svaku vrijednost oduzeli od jedan. Za grupiranje podataka koristili smo DBSCAN [3] algoritam koji ne zahtijeva poznavanje broja grupa unaprijed. Također može pronaći grupe različitih veličina i oblika.

Algoritam zahtjeva dva parametra: ε (eps) i minimalni broj primjera da čine jednu grupu. Rad algoritma počinje s nasumičnom točkom koja još nije posjećena, nakon toga se uzima ε-susjedstvo i provjerava se ako susjedstvo sadrži dovoljan broj primjera. Ako postoji dovoljan broj primjera stvara se grupa, a ako nema početna točka je označena kao šum te se može kasnije upasti u neku grupu. Taj se postupak ponavlja dok nisu su svi primjeri označeni kao grupa ili šum.

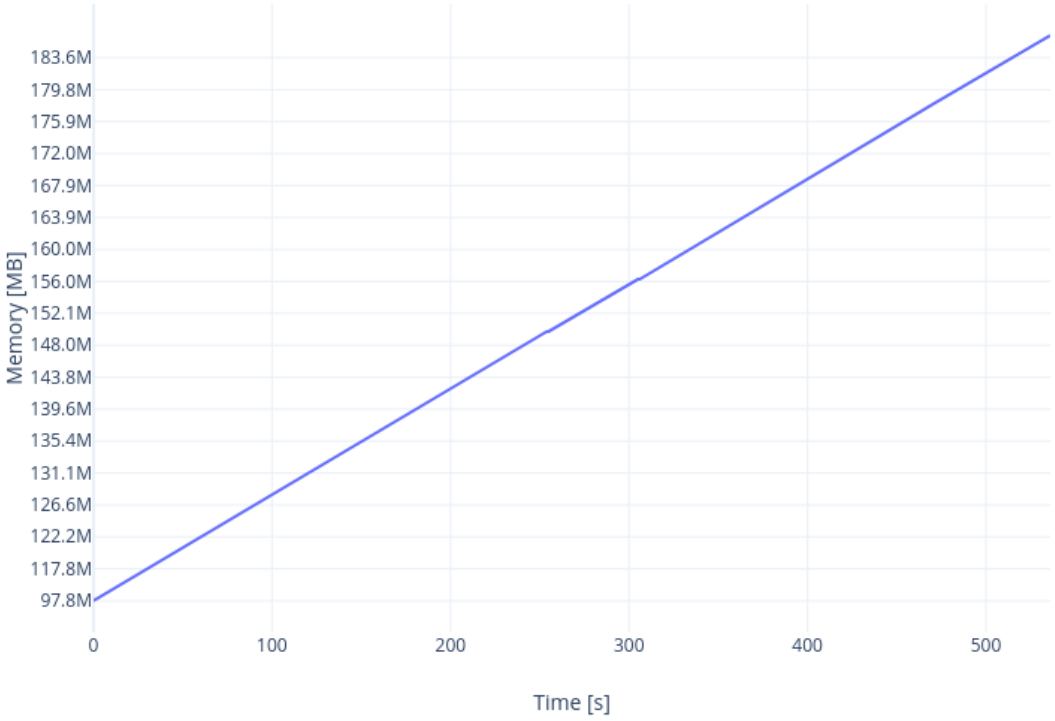
Nakon dobivanja svih grupa potrebno je odrediti centroide. Za svaku grupu gradi se matrica udaljenosti svih primjera u njoj. Primjer koji je najmanje udaljen od svih u grupi smatramo centroidom za tu grupu.

5. Testiranje i analiza rješenja

Za početak smo testirali ukupno izvođenje program te prikazali rezultate u tablici. Dok na slici 4. vidimo utrošak memorije(RAM-a) kroz vrijeme izvođenja programa.

Naziv	Globalni algoritam trajanje [s]	Lokalni algoritam trajanje [s]
J1_S_CE_IonXpress_018	0.59	0.70
J10_L_CE_IonXpress_030	114.49	142.91
J29_B_CE_IonXpress_005	362.40	425.44
J30_B_CE_IonXpress_006	615.71	745.70

Tablica 5. Prikaz vremena izvođenja programa



Slika 4. Potrošnja memorije ovisno o vremenu

Za analizu točnosti koristit ćemo skup poznatih rezultata. U ispisu pronađenih sekvenci nalaze se početnice i završnice, to su sekvence karakteristične za svaki gen. Poravnanje dviju sekvenci se najčešće prikazuje u CIGAR formatu, gdje tablica 6. prikazuje elemente CIGAR zapisa.

Oznake u ispisu	Značenje
=	Baze na sekvencama se podudaraju
D	Brisanje baze na očitanju
I	Umetanje baze u očitanje
X	Baze na sekvencama se ne podudaraju
S	Baze na sekvenci su uklonjene

Tablica 6. Opis elementa CIGAR formata

Na uzorku J30_B_CE_IonXpress_006, za kojeg imamo poznat rezultat, dobivamo tri varijante gena koristeći globalni algoritam. Slijedi prikaz rezultata, broj u zagradi predstavlja duljinu sekvence.

Consensus (296)

GATCCTCTCTCTGCAGCACATTTCCTGGAGTATGCTAAGAGCGAGTGTCATTTCTCCAACGGGA
CGCAGCGGGTGCGGTTCCTGGACAGATACTTCTATAACCGGGAAGAGTACGTGCGCTTCGACAG
CGACTGGGGCGAGTTCCGGGCGGTGACCGAGCTGGGGCGGCCGTCCGCCAAGTACTGGAACAG
CCAGAAGGATTTCATGGAGCAGAAGCGGGCCGAGGTGGACACGGTGTGCAGACACAACTACGG
GGTTATTGAGAGTTTCACTGTGCAGCGGCGAGGTGACGCGAA

Consensus (296)

GATCCTCTCTCTGCAGCACATTTCCTGGAGCATCTTAAGGCCGAGTGTCATTTCTTCAACGGGAC
GGAGCGGATGCAGTTCCTGGCGAGATACTTCTATAACGGAGAAGAGTACGCGCGCTTCGACAG
CGACGTGGGCGAGTTCCGGGCGGTGACCGAGCTGGGGCGGCCGGACGCCAAGTACTGGAACAG
CCAGAAGGAGATCCTGGAGCAGCACGGGGCAGAGGTGGACAGGTACTGCAGACACAACTACG
GGGTCGGTGAGAGTTTCACTGTGCAGCGGCGAGGTGACGCGAA

Consensus (295)

GATCCTCTCTCTGCAGCACATTTCCTGATGTATACTAAGAAAGAGTGTCATTTCTCCAACGGGAC
GCAGCGGGTGGGGCTCCTGGACAGATACTTCTATAACGGAGAAGAGTTCGTGCGCTTCGACAG
CGACTGGGGCGAGTTCCGGGCGGTGACCGAGCTGGGGCGGCCGGACGCCGAGGCTGGAACAGA
CAGAAGGAGCTCCTGGAGCAGAGGCGGGCCGCGGTGGACACGTACTGCAGACACAACTACGGG
GTTATTGAGAGTTTCACTGTGCAGCGGCGAGGTGACGCGAA

Prva sekvenca koja je dobivena, a zastupljena je s 1168 primjera u grupi, potpuno se podudara s poznatim rješenjem (CIGAR: 27S 249= 20S). Druga sekvenca razlikuje se od poznate za jednu zamjenu (27S 189= 1X 59= 20S) te je zastupljena s 910 primjera. Zadnja sekvenca, koja je zastupljena sa samo 14 primjera, daje CIGAR ispis 27S 145= 1D 2= 1I 6= 1I 9= 1X 84= 20S, što znači da je potrebno obaviti jednu operacije brisanja, dvije operacije umetanja i jednu zamjenu kako bi se dvije sekvence podudarile.

Na uzorku J29_B_CE_IonXpress_005, za kojeg također imamo poznat rezultat, dobivamo tri varijante gena. Rezultati su prikazani u nastavku.

Consensus (296)

GATCCTCTCTCTGCAGCACATTTCCTGCTGTATGCTAAGAGCGAGTGTCATTTCTCCAACGGGAC
GCAGCGGGTGGGGTTCCTGGACAGATACTTCTATAACGGAGAAGAGTTCGTGCGCTTCGACAGC
GACTGGGGCGAGTACCGGGCGGTGACAGAGCTGGGGCGGCCGGTGGCCGAGTACCTGAACAGC
CAGAAGGAGTACATGGAGCAGACGCGGGCCGAGGTGGACACGTACTGCAGACACAACTACGG
CGGCGTTGAGAGTTTCACTGTGCAGCGGCGAGGTGACGCGAA

Consensus (298)

GATCCTCTCTCTGCAGCACATTTCCTGCTGTATACTACGAGCGAGTGTCATTTCTCCAACGGGAC
GCAGCGGGTGGGGTTCCTGGACAGATACTTCTATAACGGAGAAGAGTACGTGCGCTTCGACAG
CGACTGGGGCGAGTACCGGGCGGTGACAGAGCTGGGGCGGCCGTCCGCCAAGTACTGGAACAG
CCAGAAGGAGTACATGGAGCAGACGCGGGCCGAGGTGGACAGGTACTGCAGACACAACTACG
GGGTTCTTGACAGTTTCGCTGGTGCAGCGGTCGAGGTGACGCGAA

Consensus (297)

GATCCTCTCTCTGCAGCACATTTCCTGCTGTATGCTAAGAGCGAGTGTCATTTCTCCAACGGGAC
GCAGCGGGTGGGGTTCCTGGACAGATACTTCTATAACGGAGAAGAGTTCGTGCGCTTCGACAGC
GACTGGGGCGAGTACCGGGCGGTGACAGAGCTGGGGGCGGCCGGTGGCCGAGTACCTGAACAG
CCAGAAGGAGTACATGGAGCAGACGCGGGCCGAGGTGGACACGTACTGCAGACACAACTACG
GGGTCGTTGAGAGTTTCACTGTGCAGCGGCGAGGTGACGCGAA

Za prvu ispisanu sekvencu postoji grupa od 1297 sličnih sekvenci, za drugu 143 sličnih sekvenci, dok za treću sekvencu postoji samo jedan primjer. Prvi primjer razlikuje se od rješenja za samo tri brisanja (CIGAR: 24S 3= 3D 246= 20S). Drugi primjer razlikuje se od rješenja u dvije zamjene (CIGAR: 27S 247= 2X 22S). Zadnji primjer nije nalik poznatome rješenju što je i očekivano pošto je jedini u grupi.

Razvijen program, iako pronalazi dobre varijante gena iz testnih utoraka, ima i probleme. Najveći problem je u podzastupljenosti određenih alela u uzorku, odnosno mali broj varijanti gena u pojedinim grupama. U tom slučaju dolazi do stvaranja grupe koja ne reprezentira stvarni alel, nego je vjerojatno greška kod sekvenciranja.

6. Upute za korištenje

Za pokretanje programa prvo potrebno je klonirati projekt s GitHub repozitorija. Na samom računalu za pokretanje potrebno je instalirati Python 3.6+ i C prevoditelj gcc. Nakon početnih koraka pokreću se skripte koje prevode C kod te generiraju Python modul za daljnje korištenje i instaliraju sve potrebne module za Python. Prije pokretanja skripti moguće je promijeniti parametre algoritma u datoteci alignment.c. Na Linux operacijskom sustavu unutar komandne linije pokreće se skripta build.sh, dok na Windowsima build.bat. Tada se može pokrenuti sam program za pronalaženje varijanti gena. Pokretanje se vrši u naredbenom retku s naredbom: python main.py <putanja_do_FASTA/FASTQ_datoteke>. Nakon završetka programa varijante gena će se spremiti u output.fasta te će se također ispisati na zaslon unutar konzole.

```
(bio) PS D:\bio projekt tulek\Bioinf> python .\main.py .\data\fastq\J30_B_CE_IonXpress_006.fastq
Data length: 3818
Data length(after): 2167
100%|████████████████████████████████████████| 4695889/4695889 [10:00<00:00, 7815.19it/s]
Cluster 2, length: 14
Cluster 0, length: 1168
Cluster 1, length: 910
Number of clusters: 3
Consensus (295)
GATCCTCTCTCTGCAGCACATTTCCTGATGTATACTAAGAAAGAGTGTCATTTCTCCAACGGGACGCAGCGGGTGGGGCTCCTGGACAGATACTTCTATAACGGA
GAAGAGTTCGTGCGCTTCGACAGCGACTGGGGCGAGTTCCGGGCGGTGACCGAGCTGGGGCGGCCGGACGCCGAGGCTGGAACAGACAGAAGGAGCTCCTGGAGC
AGAGGCGGGCCGCGGTGGACACGTACTGCAGACACAACTACGGGGTTATTGAGAGTTTCACTGTGCAGCGGCGAGGTGACGCGAA

Consensus (296)
GATCCTCTCTCTGCAGCACATTTCCTGGAGTATGCTAAGAGCGAGTGTCATTTCTCCAACGGGACGCAGCGGGTGCGGTTCCTGGACAGATACTTCTATAACCGG
GAAGAGTACGTGCGCTTCGACAGCGACTGGGGCGAGTTCCGGGCGGTGACCGAGCTGGGGCGGCCGTCCGCCAAGTACTGGAACAGCCAGAAGGATTTCATGGAG
CAGAAGCGGGCCGAGGTGGACACGGTGTGCAGACACAACTACGGGGTTATTGAGAGTTTCACTGTGCAGCGGCGAGGTGACGCGAA

Consensus (296)
GATCCTCTCTCTGCAGCACATTTCCTGGAGCATCTTAAGGCCGAGTGTCATTTCTTCAACGGGACGGAGCGGATGCAGTTCCTGGCGAGATACTTCTATAACGGA
GAAGAGTACGCGCGCTTCGACAGCGACGTGGGCGAGTTCCGGGCGGTGACCGAGCTGGGGCGGCCGGACGCCAAGTACTGGAACAGCCAGAAGGAGATCCTGGAG
CAGCACGGGGCAGAGGTGGACAGGTACTGCAGACACAACTACGGGGTCGGTGAGAGTTTCACTGTGCAGCGGCGAGGTGACGCGAA
```

Slika 5. Ispis programa

7. Zaključak

Radeći na ovom projektu upoznali smo se s principima bioinformatike te kako te iste principe iskoristiti u datom projektu. Projekt je dosta računarski nastrojen s klasteriranjem, ali sama obrada podataka nas je naučila algoritmima poravnavanja sekvenci. Naše rješenje radi dobro na datim primjerima te testnim podacima, no uvijek ima mjesta za nadogradnju i poboljšanje.

Također smo pokušali iskoristi dosadašnje znanje s fakulteta da napravimo, po nama, čim točnije i bolje rješenje predstavljenog problema u projektu. To se može i vidjeti u spajanju dvaju jezika za postizanje boljih performansi.

Nakon ovog imamo bolji uvid u svijet bioinformatike i kako bi se ona mogla koristiti za unapređenje analize gena te samim time dovesti do poboljšanja općenitog zdravlja i života na Zemlji.

Literatura

[1] »Needleman–Wunsch algorithm,« Available: https://en.wikipedia.org/wiki/Needleman%E2%80%93Wunsch_algorithm.

[2] »Smith–Waterman algorithm,« Available: https://en.wikipedia.org/wiki/Smith%E2%80%93Waterman_algorithm.

[3] »DBSCAN,« scikit-learn.org, Available: https://scikit-learn.org/stable/modules/clustering.html#dbscan.

[4] M. D.-L. Mile Šikić, Skripta iz bioinformatike.

[5] D. K. Nikica Hlupić, »Dinamičko programiranje - prezentacija iz predmeta NASP.«.

[6] S. Kosier, »Pronalaženje varijanti gena iz podataka dobivenih sekvenciranjem,« 2019.