SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 6487

Primjena evolucijskih algoritama u bioinformatici

Patrik Okanović
Voditelj: prof. dr. sc. Marin Golub

Zagreb, svibanj 2020.

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

Zagreb, 2. ožujka 2020.

Polje: **2.09 Računarstvo**

ZAVRŠNI ZADATAK br. 6487

Pristupnik: **Patrik Okanović (0036505553)**
Studij: Računarstvo
Modul: Računarska znanost

Zadatak: **Primjena evolucijskih algoritama u bioinformatici**

Opis zadatka:

Navesti podjelu i opisati evolucijske algoritme. Posebnu pažnju posvetiti genetskom algoritmu. Odabrati, opisati i definirati nekoliko optimizacijskih problema iz područja bioinformatike. Razmotriti mogućnost rješavanja odabranih problema evolucijskim algoritmima. Opisati razvojno okruženje za evolucijsko računanje (engl. Evolutionary Computation Framework, ECF) i u tom okruženju programski ostvariti evaluacijsku funkciju koja predstavlja vrednovanje jedinke za odabrani optimizacijski problem. Usporediti dobivene rezultate optimiranja s rezultatima iz literature.

Zadatak uručen pristupniku: 13. ožujka 2020.
Rok za predaju rada: 12. lipnja 2020.

Mentor:

Prof. dr. sc. Marin Golub

Predsjednik odbora za završni rad modula:

Doc. dr. sc. Marko Čupić

Djelovođa:

Izv. prof. dr. sc. Tomislav Hrkać

SADRŽAJ

1. Uvod

Razvojem područja računarstva te unaprijeđenjem procesora, započinju se rješavati sve složeniji problemi koji su dotada bili nerješivi. Koriste se algoritmi pretraživanja cjelokupnog prostora rješenja, to jest tehnika grube sile. Međutim takvi algoritmi pokazali su se poprilično nedjelotvornim i pronalazak globalnog optimuma pokazao se nerješiv u realnom vremenu. Srećom, pokazano je da nema potrebe ispitivati svaku permutaciju rješenja, što je rezultiralo pronalaskom djelotvornih algoritama. Takvi algoritmi koji pronalaze rješenja koja su zadovoljavajuće dobra, ali ne nude garanciju optimalnosti rješenja, te su im složenosti relativno niske nazivaju se heuristike. Jedna od specijalizacija heurističkih algoritama, točnije metaheuristika čiji je zadatak usmjeravanje problemski specifičnih heuristika prema području u prostoru rješenja u kojem se nalaze dobra rješenja [14], jest područje evolucijskog računanja. Što je to evolucijsko računarstvo? Jedan od odgovora bi bio : „evolucijsko računanje je područje računarske znanosti koje razmatra algoritme koji simuliraju evolucijski razvoj vrsta, život i ponašanje jedinke u društvu ili pak simuliraju različite aspekte umjetnog života.“ [12]. Bioinformatika kao izazovno područje obiluje problemima velikih složenosti. Primjena evolucijskog računarstva, posebice genetskog algoritma na neke od tih problema izložena je u nastavku.

Ovaj završni rad sastoji se od 7 poglavlja. Nakon uvodnog poglavlja u 2. poglavlju objašnjen je genetski algoritam.

U 3. poglavlju dan je pregled triju optimizacijskih problema iz područja bioinformatike te je razmotrena implementacija rješenja genetskim algoritmom.

Poglavlje 4. opisuje se razvojno okruženje za evolucijsko računanje korišteno u implementaciji rješenja.

Poglavlje 5. izlaže rješenje optimizacijskog problema iz bioinformatike unutar oda-

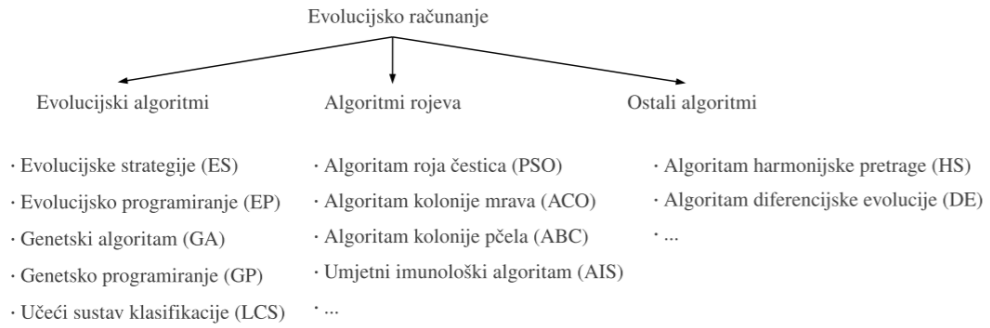
branog razvojnog okruženja za evolucijsko računanje.

Unutar 6. poglavlja demonstrirani su rezultati programske implementacije te je dana usporedba s rezultatima iz literature.

Konačno, 7. poglavlje iznosi zaključak završnog rada.

2. Genetski algoritam

Evolucijsko računanje može se podijeliti u tri grane, kao što je prikazano na slici 2.1: evolucijske algoritme, algoritme rojeva te ostale algoritme. Genetski algoritam pripada prvoj navedenoj grani evolucijskih algoritama.



Slika 2.1: Hijerarhijski prikaz evolucijskog računanja

Kroz milijune godina različite vrste se razvijaju kako bi što bolje preživjele u okolišu oko njih. Razvijene prilagodbe se prenose unutar vrste generacijama, te je proces prilagodbe kontinuirani proces. Prilagodbe su većinom takve da omogućuju jedinki bolje šanse za preživljavanje, iako postoje i nepogodne promjene. Štetne promjene generacijama postepeno iščezavaju, dok promjene koje su korisne imaju veće šanse za očuvanjem unutar populacije jer povećavaju šanse jedinke za preživljavanjem. Opisani mehanizam jest način prirode za razvijanjem sve sposobnijih organizama.

"Genetski algoritam je heuristička metoda optimiranja koja imitira prirodni evolucijski proces." [8]. Genetski algoritmi predloženi su od strane Johna H. Hollanda 1970-ih godina. Primjenjuju se za rješavanje optimizacijskih problema, štoviše koristi se uz dobre rezultate pri učenju neuronskih mreža, traženju najkraćeg puta, problemu trgovačkog putnika, strategiji igara, problemima sličnim transportnom problemu, optimiranjima nad bazom podataka te u mnogim drugim problemima. Genetski algoritam modelira prirodni genetski sustav:

– genetski materijal jedinke zapisan je u strukturi zvanoj kromosom,

– nad populacijom se vrše genetski operatori: križanje i mutacija,

– događa se selekcija.

Pseudokod genetskog algoritma dan je na 2.2.

```
Genetski_algoritam
{
   t = 0
   generiraj početnu populaciju potencijalnih rješenja P(0);
   sve dok nije zadovoljen uvjet završetka evolucijskog procesa
   {
      t = t + 1;
      selektiraj P'(t) iz P(t-1);
      križaj jedinke iz P'(t) i djecu spremi u P(t);
      mutiraj jedinke iz P(t);
   }
   ispiši rješenje;
}
```

Slika 2.2: Pseudokod genetskog algoritma

Za genetski algoritam jedinke su potencijalna rješenja, a okolinu predstavlja funkcija cilja. Ukoliko postoji potreba za implementiranjem genetskog algoritma kao metode optimiranja, potrebno je definirati proces dekodiranja i preslikavanja podataka zapisanih u kromosomu te odrediti funkciju dobrote.

Neke od mogućnosti prikaza rješenja genetskog algoritma su:

– prikaz rješenja s pomoću binarnog koda, to jest binarni prikaz,

– prikaz s Grayevim kodom,

– kromosom kao broj s pomičnom točkom,

– matrični prikaz.

Funkcija dobrote u literaturi zvana *fitness* funkcija ili funkcija ocjene kvalitete jedinke je ključ za proces selekcije. Što je kvaliteta jedinke veća uz pomoć funkcije dobrote ta jedinka će imati veću šansu za preživljavanje.

Selekcijom se čuvaju i prenose dobra svojstva jedinki na sljedeću generaciju u populaciji. Genetske algoritme s obzirom na vrstu selekcije dijelimo na generacijske i eliminacijske. Karakteristične vrste selekcija koje koristi generacijski genetski algoritam su: jednostavna selekcija i turnirska selekcija.

U praktičnom dijelu ovog rada koristi se 3-turnirska eliminacijska selekcija. 3-turnirska eliminacijska selekcija odabire slučajno tri različite jedinke iz populacije i eliminira najlošiju jedinku za razliku od 3-turnirske generacijske selekcije koja kopira najbolju jedinku u novu populaciju. Vjerojatnost eliminacije *t*-te jedinke uporabom 3-turnirske selekcije iznosi

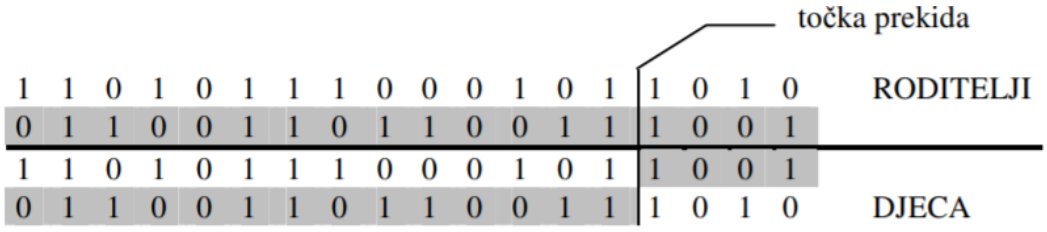
$$p_{3,E}(i) = \frac{\binom{i}{3} - \binom{i-1}{3}}{\binom{N}{3}}$$

Tri jedinke iz skupa od N jedinki moguće je odabrati na $\binom{N}{3}$načina. Najlošija jedinka od tri odabrane ima najveću oznaku. Ukolika je odabrana *i*-ta jedinka sve ostale jedinke moraju imati manji ili jendaki indeks. Vjerojatnost da su slučajno odabrane jedinke u skupu od prvih *i* iznosi $\frac{\binom{i}{3}}{\binom{N}{3}}$. Konačno od te vjerojatnosti potrebno je oduzeti vjerojatnost da sve tri jedinke imaju indeks manji od *i*, a to iznosi $\frac{\binom{i-1}{3}}{\binom{N}{3}}$.

2.1. Utjecaj operatora

Neovisno o tome koja inačica genetskog algoritma se koristi, operatori mutacije i križanja imaju značajnu ulogu.

Operator križanja na temelju roditelja stvara djecu. Zbog toga što su djeca slična roditeljima, ovaj operator ima ulogu fine pretrage prostora rješenja u okolici rješenja koja reprezentiraju roditelji.

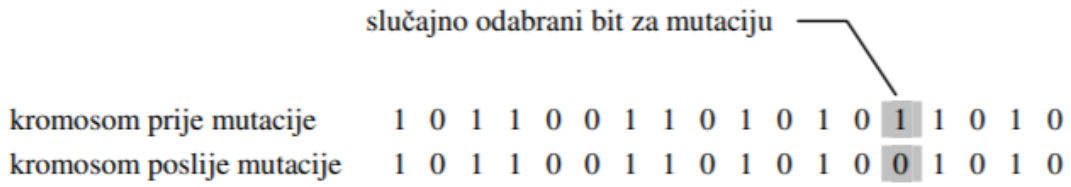


Slika 2.3: Križanje s jednom točkom prekida

Slika 2.3 prikazuje često korišteni operator križanja s jednom točkom prekida prilikom korištenja binarne reprezentacije jedinke. Slučajno se odabire točka prekida unutar jedinki. Kromosomi roditelja prelamaju se na točki prekida i tvore djecu. Prvo dijete prvi dio kromosoma dobiva od prvog roditelja, a drugi dio od drugog roditelja. Drugo dijete prvi dio kromosoma dobiva od drugog roditelja, a drugi dio kromosoma od prvog roditelja.

Operator mutacije ima suprotnu ulogu, on nad jedinkom radi jednu ili više slučajnih izmjena koje jedinku mogu drastično izmijeniti . Ovaj operator povećava volumen

populacije u postupku pretraživanja. Druga bitna uloga operatora mutacije jest izbacivanje populacije iz lokalnih optimuma.



Slika 2.4: Jednostavna mutacija [8]

Slika 2.4 prikazuje jednostavnu mutaciju prilikom korištenja binarne reprezentacije jedinke. Svaki bit kromosoma mutira s odabranom vjerojatnošću p. Postoje brojne mutacije koje je moguće koristiti. Odabir mutacije mora odgovarati problemu koji se rješava. Neki od primjera operatora mutacije nad binarnim kromosomom mogu biti: invertiranje svih bitova ili mutiranje točno određenog broja slučajnih pozicija u kromosomu.

Kako bi genetski algoritam bio djelotvoran potrebno je postići dobar balans između operatora križanja i operatora mutacije. Ukoliko je djelovanje križanja prejako, postupak može vrlo lako zaglaviti u lokalnome optimumu. Ako je djelovanje mutacije prejako, može se dogoditi da mutacija uništava sve pozitivno što je algoritam dotada pronašao, što svodi pretragu na slučajnu. U idealnoj situaciji, ovi operatori u balansu osiguravaju da algoritam pretraživanja napreduje prema kvalitetnijim rješenjima.

3. Optimizacijski problemi iz bioinformatike

Prije predstavljanja tri optimizacijska problema iz bioinformatike dan je uvod u osnove genomike te korišteni format podataka u implementaciji rješenja. Predstavljeni optimizacijski problemi su:

1. Populacijska genetika u kojoj je potrebno simulirati prirodni tijek razmnožavanja i evolucije te pratiti samo najvjerojatnije ishode u kojima se unutar populacije održavaju dominantna svojstva

2. Genetsko kojim je potrebno odrediti položaj kodirajućeg gena u slijedu.

3. Poravnanje više sljedova odjednom kao što naziv kaže potrebno je dane DNK ili RNK sljedove nadopuniti prazninama te poravnati uz najveće moguće preklapanje.

3.1. Osnovni pojmovi u području genomike

Genomika je grana genetike koja primjenjuje metode DNA sekvenciranja te bionformatike kako bi ostvarila sekvenciranje, sastavljanje i analizu funkcija i strukture genoma.

Bipolimeri su presudne makromolekule nukleinskih kiselina, proteina i ugljikohidrata za funkcioniranje živih bića. Temeljne vrste nukleinskih kiselina su: RNA i DNA. Dok DNA služi kao spremište instrukcija za razvoj i funkcioniranje živih organizama, RNA dijelimo na nekoliko vrsta ovisno u funkciji. Tako postoji glasnička RNA (mRNA), transportna RNA (tRNA), ribosomska (rRNA) te regulacijske RNA poput mikro RNA (miRNA), male jezgrene RNA (snRNA) i male interferirajuće RNA (siRNA).

Osnovna jedinica nasljeđivanja živih organizama jest gen. Gen je dio nukelinske kiseline koji kodira informaciju za proizvodnju proteina ili RNA lanaca koji imaju aktivnu funkciju u organizmu.

DNA molekule su dvostruke uzvojnice koje se sastoje od nukleotida. Nukleotidi se sastoje od nukleinskih baza A (adenin), C (citozin), G (gvanin), T (timin), potom od naizmjeničnog niza šećera i fosfatne skupine. Lanci DNA povezani su na način da se adenin spaja s timinom, a gvanin s citozinom.

3.2. Format podataka FASTA

FASTA je najčešći oblik podataka u bioinformatici. FASTA je tekstualni format koji služi za prikazivanje sljedova proteina ili nukleinskih kiselina pri čemu je svaki nukleotid ili aminokiselina prikazana jednim slovom. U tablici 3.1 dan je prikaz svih mogućih kodova unutar FASTA zapisa podataka.

Kod nukleinske kiseline	Značenje
A	Adenin
C	Citozin
G	Gvanin
T	Timin
U	Uracil
R	A ili G
Y	C, T ili U
K	G, T ili U
M	A ili C
S	C ili G
W	A, T ili U
B	ne A (npr. C, G, T ili U)
D	ne C (npr. A, G, T ili U)
H	ne G (npr. A, C, T ili U)
V	niti T niti U (npr. A, C ili G)
N	A C G T U
X	Maskiranje
-	Praznina/procjep

Tablica 3.1: Kodovi za prikaz nukleinskih kiselina u FASTA formatu

U FASTA datoteci može biti više zapisa, a svaki zapis sastoji se od linije zaglavlja te jedne ili više linija samog slijeda. Linije komentara počinju znakom ",". Primjer FASTA datoteke:

```
>Prvi slijed
AGCCTGCT-CGT
>Drugi slijed
AGCTATAGCTAGA
>Treci slijed
AGCTATCCTAAGCG
```

Slika 3.1: Primjer FASTA datoteke

3.3. Populacijska genetika

Populacijska genetika (engl. *Population Genetics*) proučava distribuciju alela. Aleli, homologni geni, elmorfi naziv su za dva alternativna gena koja određuju istu osobinu, a u stanicama uvijek dolaze u paru. Distribucija alela uzrokovana je četirima faktorima koji definiraju evoluciju, a to su:

1. Prirodna selekcija.
2. Genetski drift, označava promjenu u frekvenciji genetske varijacije određenog alela. Događa se uslijed nasumičnog "nestajanja" gena primjerice razmnožavanjem ili bilo kojim drugim uzrokom koji sprječava da se gen za određeno svojstvo prenese na iduću generaciju.
3. Mutacije.
4. Tok gena, prijenos gena iz jedne populacije u drugu.

Za bolje razumijevanje populacijske genetike, u nastavku su iznesene biološke osnove. Genotip je genska osnovica organizma, njena genska struktura odnosno svi geni koji ga čine. Fenotip je svaka vidljiva karakteristika jedinke. U primjeru boja latica, fenotip bi bio stvarna boja latica koju vidi promatrač. Jedan gen određuje boju latica, iako aleli koji tvore gen mogu biti različiti. Diploidni organizmi poput ljudi, posjeduju dvije kopije svakog kromosoma, jednu kopiju od majke, a drugu od oca. Geni imaju jedan alel od prvoga kromosoma, te drugi alel od drugog kromosoma. Aleli mogu biti recesivni ili dominantni. Za gen s dva alela, gdje je R dominatno svojstvo, a r recesivno

svojstvo, moguće su tri kombinacije: Rr, RR, rr. Veza genotipa i fenotipa opisuje se tako da genotip plus utjecaj okoline daje fenotip. Genotip je moguće saznati gledajući DNK, dok je fenotip moguće utvrditi promatranjem jedinke. Lewontin 1974. predlaže sljedeći opis cilja preslikavanja "genetskog i fenotipskog prostora" [13] :
"Izazov teorije populacijske genetike je pružiti skup pravila pomoću kojih je moguće predvidjeti preslikavanje populacije genotipa G1 u domenu fenotipa P1, gdje postoji selekcija, te drugi skup pravila koje preslikavaju P1 u skup fenotipa P2, te ponovno u domenu genotipa G2."

$$\text{G1} \rightarrow \text{P1} \rightarrow \text{P2} \rightarrow \text{G2}$$

Složenost problema je eksponencijalna [13], zbog toga je prikladno primijeniti genetski algoritam.

Genetskim algoritmom moguće je simulirati populacijsku genetiku uz pseudonepredvidljive uvjete. Pitanje na koje se želi odgovoriti eksperimentom: može li promatranje prethodnih tokova gena pomoći u predviđanju budućih generacija?

Moguća implementacija rješenja od korisnika bi tražila unos broja alela koje je potrebno pratiti. Dopuštene kombinacije alela bilo bi moguće zadati konfiguracijskom datotekom uz ostale parametre koji simuliraju uvjete u prirodi, poput svojstva dominacije, recesivnosti, pojedine vjerojatnosti i mnoge druge. Za svaku populaciju bilo bi potrebno pratiti frekvencije alela. Praćenjem frekvencije alela ostvario bi se uvid u tok gena te bi bila dana sposobnost predikcije svojstava budućih naraštaja. Primjerice bez dodatnih ograničenja ukoliko bi korisnik zadao alele : {0, 1, 2, 3}. Postoje 4^2 kombinacije alela. Izradom različitih funkcija dobrote koje bi bile naklonjenije pojedinim alelima približno bi se modelirali uvjeti prirode. U radu [13] korišten je generacijski genetski algoritam s jednostavnom selekcijom. Jedinka predstavlja par alela te je u implementaciji reprezentirana binarnim prikazom. Korišteni operator križanja je križanje s jednom točkom prekida, a korišteni operator mutacije je jednostavna mutacija kojom se mijenjaju slučajno odabrani bitovi unutar reprezentacije jedinke. Iz slika 3.2 i 3.3 rezultata rada [13] moguće je zaključiti promatranjem frekvencija koji geni su dominantni (0 i 2), a koji recesivni (1 i 3). Kroz sve veći broj iteracija stvaranjem novih generacija, dominanta svojstva postaju zastupljenija, a recesivnim se frekvencija pojavljivanja smanjuje.

```
------------------------------------------------------------------------
Generation 0:
------------------------------------------------------------------------
Trait Frequencies:
[00]=5.480999946594238
[01]=5.546999931335449
[02]=5.572000026702881
[03]=5.427999973297119
[10]=9.914999961853027
[11]=9.888999938964844
[12]=10.104999542236328
[13]=9.98799991607666
[20]=5.51200008392334
[21]=5.629000186920166
[22]=5.684000015258789
[23]=5.585000038146973
[30]=3.9230000972747803
[31]=3.867999792098999
[32]=3.926999807357788
[33]=3.9469997882843018

Allele Frequencies:
[0]=23.42949867248535
[1]=32.415000915527344
[2]=23.849000930786133
[3]=20.306499481201172

Statistics:
Random Allele Alterations: {3=0.0, 2=0.0, 1=0.0, 0=0.0}
Random Allele Increase Flags: {}
Fitness Sum: 1.0E7
Fitness Average: 100.0
Fitness Max: 100.0
------------------------------------------------------------------------
```

Slika 3.2: Početna populacija

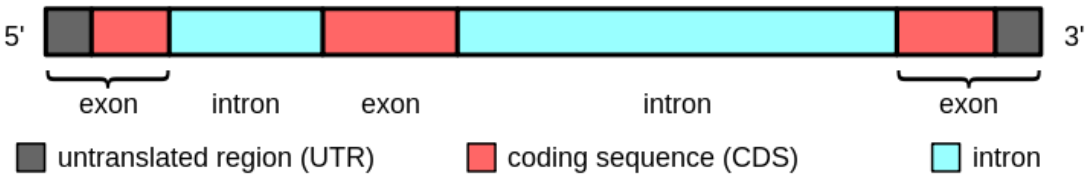
```
------------------------------------------------------------------------
Final Generation:
------------------------------------------------------------------------
Trait Frequencies:
[00]=31.865999221801758
[01]=9.92300033569336
[02]=13.26300048828125
[03]=2.267000198364258
[10]=9.173999786376953
[11]=2.758999824523926
[12]=3.929999828338623
[13]=0.6599999666213989
[20]=12.645999908447266
[21]=3.9099998474121094
[22]=5.269000053405762
[23]=0.9490000009536743
[30]=1.840000033378601
[31]=0.5950000286102295
[32]=0.7980000376701355
[33]=0.1509999930858612

Allele Frequencies:
[0]=56.42250061035156
[1]=16.854999542236328
[2]=23.017000198364258
[3]=3.7055001258850098

Statistics:
Random Allele Alterations: {3=0.18, 2=0.0, 1=0.0, 0=0.0}
Random Allele Increase Flags: {3=false, 2=false, 1=false, 0=true}
Fitness Sum: 9867091.240000026
Fitness Average: 98.67091240000026
Fitness Max: 100.0
------------------------------------------------------------------------
```

Slika 3.3: Završna populacija

3.4. Genetsko predviđanje

Genetsko predviđanje(engl *gene prediction, gene finding*) jedan je od najizazovnijih problema bioinformatike. Cilj je genetskog predviđanja identificirati područja u lancu koja će kodirati proteine. Kodiranje slijeda dušičnih baza jedno je od temeljnih spoznaja u biologiji. Niz od tri dušične baze tvori kodon. Postoje 64 moguće kombinacije kodona, neke kombinacije predstavljaju početak proteina i takvi kodoni nazivaju se start kodoni, sukladno tome neke kombinacije označavaju kraj te ih nazivamo stop kodonima. Različite permutacije kodona počevši sa start kodonima, a završavajući sa stop kodonima tvore različite bjelančevine. Zbog toga što je iz dana u dan sekvencirano sve više genoma, potreba za djelotvornom analizom tih lanaca postaje sve veća.

Prvi korak analize jest locirati kodirajuća i nekodirajuća područja sekvenci RNK, odnosno egzone i introne. Egzoni su uz introne dijelovi koji tvore RNK slijed, razlika između egzona i introna jest u tome što egzoni kodiraju proteine. Prema tome u stanici se mRNK prelama na dodiru introna i egzona i nakon toga samo egzoni kodiraju proteine. Metode je moguće podijeliti u dvije kategorije: pretraživanja po sličnosti te *ab initio* predviđanja. Slika 3.4 prikazuje strukturu gena.

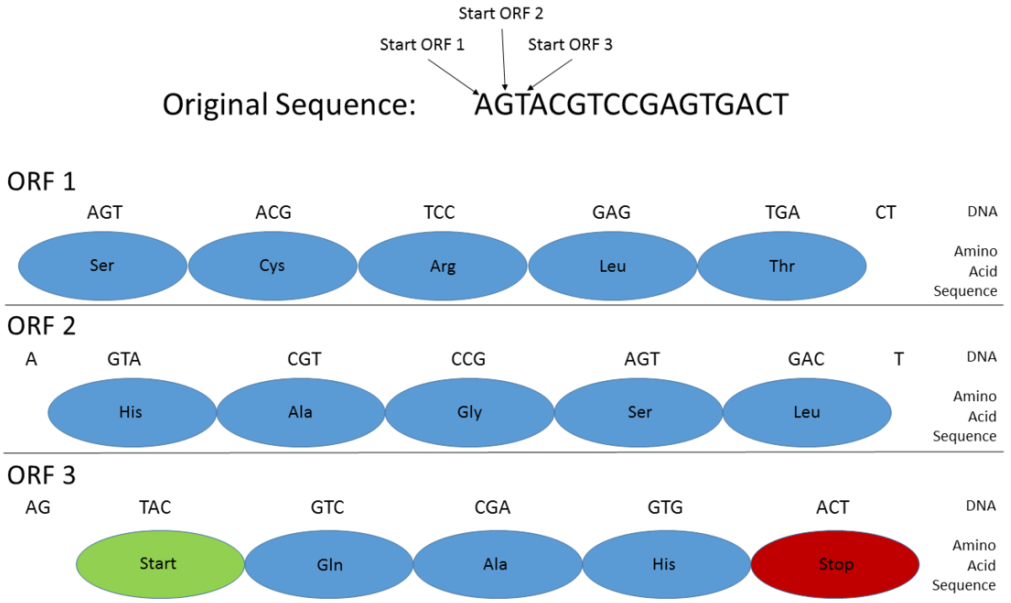


Slika 3.4: Struktura gena [1]

Empirijskim metodama pronalaska gena potrebno je pronaći genom sekvenci kojim se može identificirati područje uz pomoć mRNK, tvorevinama proteina, homolognim ili ortogonalnim nizovima. Uz mRNK sekvencu, trivijalno je otkriti otkuda se dogodila transkripcija [13]. Pomoću danog proteinskog slijeda, moguće je pronaći familiju kodirajućih DNK sekvenci reverznim prevođenjem genetskog koda. Nakon sekvenciranja, algoritmi lokalnog poravnjanja poput algoritma BLASTA i algoritma Smith-Waterman pretražuju područja sličnosti između dijela DNK i potencijslnih kandidata. Uspješnost ovakvog načina ograničena je ispravnošću baze poznatih sekvenci.

Ab initio (lat. *od početka*) genetsko predviđanje je intrinzična metoda zasnovana na detekciji signala i sadržaja. Primjeri signala su: start kodoni, stop kodoni, točke grananja. Algoritmi poput dinamičkog programiranja, linearne analize diskrimanti, Linguist metode, skrivenih Markovljevih modela te neuronske mreže koriste se u *ab initio* metodi pronalaženja gena. Općenito u algoritmima za predviđanje gena prvo je

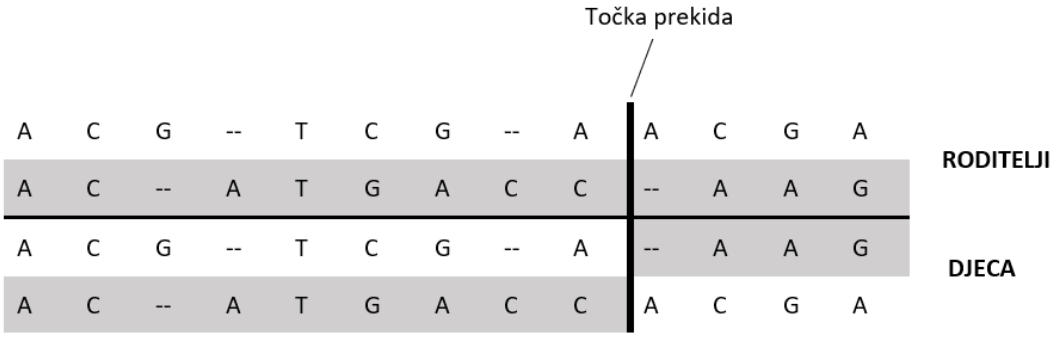
potrebno identificirati ORF (engl. *Open Reading Format*). ORF se može objasniti kao dio slijeda kojeg je moguće prevesti. Prevođenje u biologiji odgovara postupku u kojem stanični ribosomi sintetiziraju proteine nakon transkripcije RNK u DNK. Jedan od mogućih načina kako prepoznati ORF jest sljedeći: za pretpostavljene ORFove na promatranome slijedu napraviti prevođenje danoga slijeda u aminokiselinu pomoću baze podataka ORFova te nakon toga tu aminokiselinu potražiti u drugoj bazi podataka [13]. Postojeći programi koji služe za prepoznavanje ORFa su BioEdit, DNA STRIDER, PLOTORF, GETORF... Slika 3.5 prikazuje postupak predviđanja gena prepoznava-



Slika 3.5: Prepoznavanje ORFa [2]

njem ORFova, određuju se moguće granice početka kodiranja slijeda. Funkcionalni proteini moraju započeti sa start kodonom (gdje započinje transkripcija DNA), a završavaju sa stop kodonom (gdje se transkripcija završava). Gledajući gdje ti kodoni mogu pasti u slijedu DNK, može se vidjeti gdje se može nalaziti funkcionalni protein. Ovo je važno u predviđanju gena, jer može otkriti gdje se kodiraju geni u čitavom genomskom slijedu. U ovom primjeru, funkcionalni protein može se otkriti pomoću ORF3, jer započinje s početnim kodonom, ima više aminokiselina, a zatim završava stop kodonom, a sve unutar istog okvira za čitanje, to jest ORFa. Moguće rješenje uporabom genetskog algoritma nalazi se u [13]. Inicijalna populacija generirana je di-

jeljenjem dane DNK sekvence u slučajni broj dijelova. Križanje jedinki ostvareno je križanjem s jednom točkom prekida. Prilikom križanja implementacija provjerava da se točka prekida ne nalazi usred kodirajućeg područja DNK.



Slika 3.6: Križanje s jednom točkom prekida

Na slici 3.6 prikazano je križanje s jednom točkom prekida. Mutacijom jedinke pokušava se prepraviti ukoliko je DNK sekvence prekinuta usred egzona. Najbitniji dio implementacije genetskog algoritma je funkcija dobrote. Funkcija dobrote uzima u obzir frekvenciju kodona te frekvenciju područja G+C nasuprot frekvenciji područja A+T, te potom kreira nehomogene Markovljeve modele za kodirana područja [13]. Algoritam izračunava vjerojatnost da je kodirajuće područje odgovarajuće pretpostavljenoj aminokiselini. Inicijalno je u implementaciji Navedena vjerojatnost koristi se kao vrijednost funkcije dobrote jedinke. Jedinka s najvećom pronađenom funkcijom dobrote predstavljat će rješenje, to jest određuje koji dijelovi ulaznog lanca kodiraju protein.

3.5. Poravnanje sljedova

"Poravnanje bioloških sljedova je najčešći prvi korak u bioinformatičkog analizi – bilo da je u pitanju pronalazak evolucijski očuvanih regija među vrstama, analiza genetske bolesti ili kreiranje rodovskog stabla" [15]. Poravnanje dva ili više slijeda predstavlja jedan od najstarijih te najviše istraživanih problema u bioinformatici. U svrhu kvalitetnijeg razumijevanja problema poravnanja više sljedova prvo je predstavljen problem poravnanja dva slijeda. Vladimir Levenshtein je 1965. godine poopćio problem koji je definirao Richard Hamming 1960. godine. Dok je Hamming definirao mjeru udaljenosti dva niza znakova jednake duljine, Levenshtein je poopćio problem na dva niza proizvoljnih duljina koji je prikladniji za bioinformatičku analizu. Prema Levenshte-

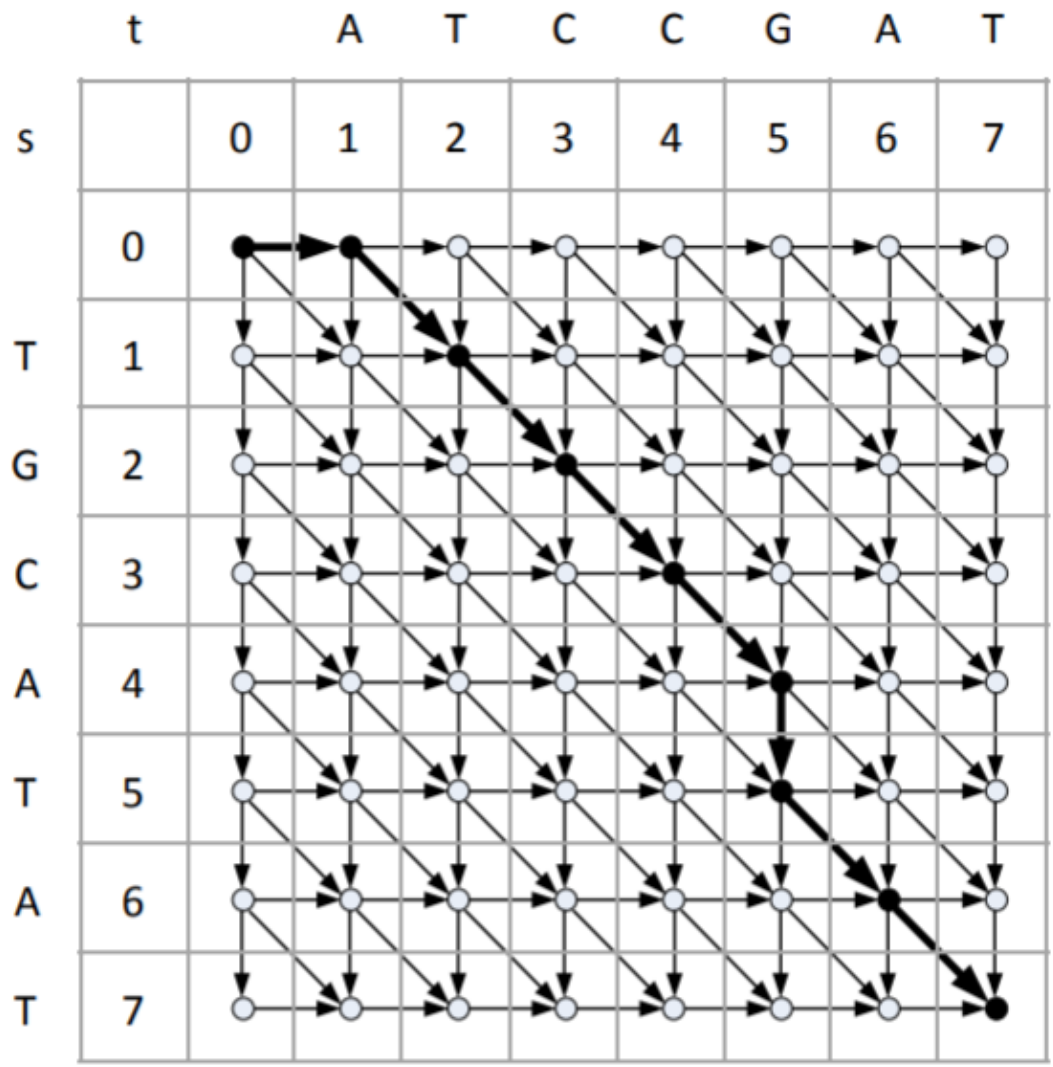
inu udaljenost dva niza jest broj potrebnih modifikacija nad jednim znakom potrebnih za pretvaranje jednog niza u drugi. Dozvoljene operacije nad jednim znakom su:

- Zamjena - promjena jednog znaka u drugi, primjerice iz l**o**v u l**a**v
- Umetanje - umetanje jednog znaka u cilju podudaranja, primjerice kas u k**l**as
- Brisanje - brisanje jednog znaka iz niza u svrhu podudaranja

Međutim Levenshtein nije predložio algoritam za određivanje udaljenosti uređivanja dva slijeda.

Neka je potrebno poravnati dva DNK niza *s=TGCATAT* i *t=ATCCGAT*. Nizove možemo predstaviti s matricom od dva retka. Kako niz *s* ne počinje s *A*, potrebno je na početak niza *t* dodati prazninu. Nemoguće je dobiti poravnjanje gdje se u oba retka za isti stupac nalazi praznina -. Iz toga slijedi da je najveća moguća duljina poravnanja jednaka zbroju duljina sljedova *t i s*. Slučaj u kojem u stupcu su retcima jednake slova, naziva se slaganje (engl. *match*), dok slučaj kada se razlikuju neslaganje (engl. *mismatch*). Rješenje računanja udaljenosti moguće je predstaviti mrežom poravnanja koja tvori graf. Kretanje između vrhova odvija se preko bridova koji predstavljaju operacije uređivanja. Iz svakog vrha moguće je kretati se u 3 smjera: desno, dolje i dijagonalno dolje-lijevo. Kretanjem dolje ili desno pomičemo se po jednome nizu, a da u drugome ostajemo na istome mjestu. Primjerice ako želimo niz *s* čiji indeksi predstavljaju redove pretvoriti u niz *t* čiji indeksi predstavljaju stupce, onde kretanjem udesno umećemo u niz *s*, dok se pomičemo za jedno mjesto u nizu *t*. Kretanjem prema dolje brišemo jedan znak iz *s*. Dijagonalno kretanje predstavlja slaganje ili neslaganje između dva niza. Svakome kretanju moguće je pridijeliti određenu težinu, čime reguliramo želimo li više penalizirati neslaganje ili nagraditi slaganje. Pronalazak optimalnog poravnanja odgovara pronalasku puta najmanje cijene iz vrha (0,0) do vrha u donjem desnom uglu. Na slici 3.7 prikazan je optimalni put za nizove *s* i *t*. Do vrha (*n, m*) u donjem desnom uglu mogli smo doći iz jednog od vrhova (*n-1, m*), (*n, m-1*), (*n-1, m-1*). Prema tome cijena puta u vrhu (*n, m*) jednaka je minimumu cijena do tih vrhova plus troškovi puta od tih vrhova do vrha u donjem desnom uglu. Postupak se rekurzivno ponavlja. Takav algoritam u kojem rješavamo manje podprobleme i progresivno rješavamo koristeći rezultate podproblema naziva se dinamičko programiranje.

Međutim bitno je istaknuti kako algoritam računanja udaljenosti nije bio prvi efikasni algoritam koji rješava problem poravnanja dva niza. Zbog potreba biologije, biologe je zanimala maksimalna sličnost bioloških sljedova. Needleman i Wunsch prvi su koji su formulirali algoritam za njegovo rješavanje temeljen na dinamičkom programiranju.

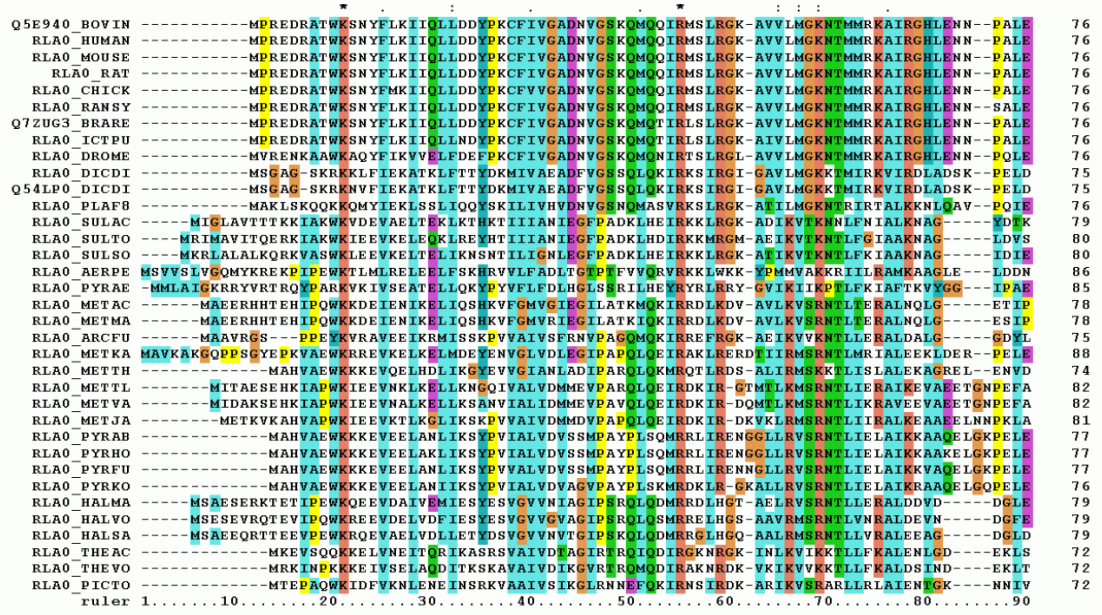


Slika 3.7: Mreža poravnanja za nizove *s=TGCATAT* i *t=ATCCGAT*

"Ako želimo s tim algoritmima poravnati genome čovjeka i miša duge nekoliko milijardi baza trebat će nam minimalno $9 \cdot 10^{18}$ operacija. Uz radni takt od 1GHz za računanje ovoga poravnanja trebat će nam preko 200 godina."[Šikić, Domazet-Lošo, Bionformatike, 2013.] Nažalost, spomenuti algoritmi nisu učinkoviti kada je potrebno usporediti cijele genome, a kamoli više genoma odjednom. Prikladniji pristup rješavanju toga problema su heuristički algoritmi.

"Poravnavanje više sljedova odjednom (engl. multiple sequence alignment; MSA) ima smisla kada su sljedovi ili neki njihovi dijelovi homologni."[Šikić, Domazet-Lošo, Bioinformatika, 2013.]. Za razliku od uspredbe ulaznog slijeda s bazom podataka, gdje je homologiju potrebno tek odrediti, kod rješavanja poravnanja više sljedova odjednom polazi se od pretpostavke da su sljedovi koje želimo poravnati homologni. Kada su

sljedovi homologni cijelom svojom duljinom i podjednake duljine, radi se o globalnom poravnanju više sljedova odjednom, poput poravnanja genoma više sojeva bakterija. Nasuprot tome, kada su sljedovi homologni samo po dijelovima, mogu se poravnati po sačuvanim dijelovima (engl. *conserved regions*) i tada se radi o lokalnom poravnanju, primjerice poravnanje homolognih proteina između različitih vrsta. Na slici 3.8 prikazan je primjer poravnjanja.



Slika 3.8: Poravnanje prvih 90 stupaca proteina L10E iz raznih organizama

Kod poravnanja tri ili više sljedova odjednom ideja je smjestiti homologna mjesta u promatranim sljedovima u iste stupce. Slika 3.9 prikazuje jednostavan slučaj poravna-

Matrica	1	2	3	4	5
1	A	A	-	G	C
2	A	C	G	G	C
3	A	C	G	-	C

Slika 3.9: Smještanje homolognih mjesta u iste stupce

nja tri slijeda. Sljedovi su prikazani u matrici, a homologna mjesta smještena su u iste stupce. Formalna definicija problema jest sljedeća: neka je zadan skup od n sljedova:

$S_1, S_2, ..., S_n$, abeceda sljedova mogu biti nukleotidi ili aminokiseline. Poravnanje više sljedova odjednom je skup sljedova $S'_1, ..., S'_n$ za koje vrijedi:

1. Duljine poravnatih sljedova su jednake, $|S'_1| = ... = |S'_n|$
2. Uklanjanje praznina "-" iz S'_i dobiva se S_i za *i=1,...,n*

Moguće je da duljine poravnatih sljedova budu dulje ukoliko su dodavane praznine u početne sljedove.

Heuristički pristupi poravnanju više sljedova odjednom su:

1. progresivno poravnanje (engl. *progressive alignment*)
2. metode temeljene na skrivenim Markovljevim modelima (engl. *Hidden Markov Model;* HMM)
3. iterativne metode
4. metode temeljene na filogenetskom predznanju
5. metode temeljene na genetskom algoritmu

Primjer progresivne metode jest alat ClustalW/ClustalX (Thompson et al., 1994; Larkin et al., 2007) te višeprocesorska verzija Clustal Omega(Sievers et al., 2011). Pimjer korištenja iterativne metode jest program MUSCLE (Edgar, 2004.). Primjer korištenja genetskog algoritma za poravnanje više sljedova odjednom izložen je u poglavlju koje opisuje implementaciju ovoga rada.

4. Razvojno okruženje za evolucijsko računanje

ECF (engl. *Evolutionary Computation Framework*) skup je gotovih programa napisanih u programskome jeziku C++ razvijenih na Zavodu za elektroniku, mikroelektroniku, računalne i inteligentne sustave namijenjenih za izradu evolucijskih algoritama. Neki od ponuđenih algoritama u ECFu su:

- genetski algoritam, genetsko programiranje
- algoritam roja čestica
- algoritam genetsko kaljenje
- algoritam kolonije pčela
- evolucijska strategija
- diferencijska evolucija

Za korištenje ECFa prethodno je potrebno preuzeti *boost C++(www.boost.org)*. ECF pruža podršku za Visual Studio i CLion razvojna okruženja. Više detalja oko instalacije moguće je pronaći na *http://ecf.zemris.fer.hr/install.html*. Za uspješno pokretanje ECFa potrebno je kao argument predati konfiguracijsku datoteku kojom se parametrizira izvođenje. Moguće je parametrizirati genotip, odnosno reprezentaciju jedinke. Trenutna verzija ECFa za genotip nudi `BitString, FloatingPoint, Binary genotype, Tree genotype i Permutation genotype`. Uz genotip moguće je odabrati algoritam izvođenja.

Primjer korištene konfiguracijske datoteke `parameters.txt` koju je prilikom pokretanja programa potrebno predati kao argument dan je na 4.1. Konfiguracijska datoteka sastoji se od tri glavne sekcije `Algorithm, Genotype, Registry`. Jedino je `Genotype` sekciju nužno definirati u svakoj konfiguracijskoj datoteci, za sve ostale parametre postoje predpostavljene vrijednosti. Sekcija `Algorithm` služi za odabir željenog algoritma te postavljanje dodatnih parametara poput veličine turnirkse

```
<ECF>
  <Algorithm>
    <SteadyStateTournament>
      <Entry key="tsize">3</Entry>
    </SteadyStateTournament>
  </Algorithm>

  <Genotype>
    <MsaGenotype>
      <Entry key="mut.insert">0.33</Entry>
      <Entry key="mut.delete">0.33</Entry>
      <Entry key="mut.simple">0.34</Entry>
    </MsaGenotype>
  </Genotype>

  <Registry>
    <Entry key="input">FASTA.txt</Entry>
  </Registry>
</ECF>
```

Slika 4.1: `parameters.txt` datoteka

selekcije, odabira elitizma ili maksimalnog broja iteracija. Unutar sekcije `Genotype` odabire se željena reprezentacija jedinke uz željene parametre specifične za pojedini algoritam. Vrlo često se postavljaju parametri vjerojatnosti pojedinih operatora mutacije. Sekcija `Registry` sadrži sve parametre nevezane za `Algorithm` ili `Genotype`, poput ulazne datoteke, definiranje maksimalnog vremena izvođenja i brojnog drugog.

Primjer evaluacijske funkcije koju je potrebno evaluirati prilikom svakog korištenja razvojnog okruženja ECF nalazi se na 4.2. Prikazana evaluacijska funkcija vrednuje bolje rješenje koje ima više jedinica u `Bitstring` genotipu.

```
FitnessP OneMaxEvalOp::evaluate(IndividualP individual)
{
  // evaluation creates a new fitness object (using a smart pointer
  // in our case, we try to maximize the number of ones
  FitnessP fitness (new FitnessMax);

  // Each individual is a vector of genotypes .
  // We'll use BitString, and retrieve it as the first and
  only genotype
  BitString::BitString* bitstr = (BitString::BitString*) individual
  getGenotype(0).get();

  // count the ones; where are they?
  // BitString genotype contains a std::vector of bool's named 'bit
  uint ones = 0;
  for(uint i = 0; i<bitstr->bits.size(); i++){
    if(bitstr->bits[i] == true)
      ones++ ;
  }
  fitness->setValue(ones);

  // return the new fitness
  return fitness;
}
```

Slika 4.2: Primjer evaluacijske funkcije iz [9]

5. Implementacija genetskog algoritma za odabrani problem

Koristeći razvojno okruženje za evolucijsko računanje Evolutionary Computation Framework - ECF riješen je problem poravnanja tri ili više DNK/RNK slijeda odjednom. Za rješavanje poravnanja aminokiselina potrebno je nadograditi algoritam na način objašnjen u nastavku.

5.1. Populacija

Kako bi unutar ECFa rješili neki optimizacijski problem potrebno je unutar konfiguracijske datoteke zadati i odabrati genotip jedinke. S obzirom da ECFova trenutna implementacija sadrži razrede `BitString, FloatingPoint, Binary genotype, Tree genotype i Permutation genotype` te nijedan genotip nije prikladan za reprezentaciju jedinke osmišljen je novi genotip. Novi genotip `MsaGenotype` nasljeđuje osnovni razred `Genotype` te sadrži atribut `vector<string> alignment` čime je reprezentirana jedinka. Virtualna metoda koju je potrebno naslijediti kako bi se napravio novi genotip jest metoda `initialize(StateP state)`. U toj metodi implementirano je generiranje inicijalne populacije. Prvim pokretanjem metode `initialize` `FastaParser.cpp` pročita datoteku zadanu kao parametar u konfiguracijskoj datoteci `<Entry key="input">FASTA.txt</Entry>` pod sekcijom `Registry` zapisanu u FASTA formatu podataka te odredeđuje najdulji niz. Nakon toga početna duljina svakog slijeda generira se na način da se u slijed umeću praznine na slučajno odabranom mjestu dok duljina slijeda ne postane 10% dulja od duljine najduljeg niza. Sljedeći primjer daje mogući izgled FASTA.txt datoteke te generirane jedinke.

Opisani način generiranja inicijalne populacije odgovara autorima Horng et al. (2005), Shyu et al. (2004) i Wang & Lefkowitz (2005), dok Hernandez et al. (2004) već prilikom inicijalnog generiranja jedinki radi djelomično poravnanje.

```
>first
ATTGCCGACT
>second
AC
>third
GACCCTAG
```

Slika 5.1: *FASTA.txt* datoteka

```
ATTGC-CGACT
--A----C---
G--A-CCCTAG
```

Slika 5.2: Generirana jedinka

5.2. Reprodukcija

Autori (Hernandez et al., 2004; Horng et al., 2005; Shyu et al.,2006; Wang & Lefkowitz, 2005) koriste jednstavnu turnirsku selekciju (engl. *roulette wheel parent seelction*). Autori (Horng et al., 2006; Wang &Lefkowitz, 2005) također koriste neki oblik elitizma kako bi sačuvali najbolju jedinku. Čiji je pseudokod dan u nastavku

```
single generation {
   select individuals to form the new generation /
    (fitness proportional selection operator);
   create new generation (make copies);
   noCrx = (deme size) * <crxRate_> / 2;
   repeat(<noCrx> times) {
      randomly select two parents;
      perform crossover, _replace_ parents with their children;
   }
   perform mutation on new generation;
}
```

Slika 5.3: Pseudokod genetskog algoritma s jednostavnom selekcijom(`roulette wheel`) u razvojnom okruženju ECF

U implementaciji rješenja u razvojnom okruženju ECF korištena je 3-turnirsku eliminacijsku selekciju čiji je pseudokod dan u nastavku:

Oba navedena algoritma implementirana su u ECFu, a odabir algoritma naveden je

```
single generation {
   repeat(deme size times) {
      randomly add <nTournament_> individuals to the tournament;
       select the worst one in the tournament;
      randomly select two parents from /
       remaining ones in the tournament;
      replace the worst with crossover child;
      perform mutation on child;
   }
}
```

Slika 5.4: Pseudokod eliminacijskog genetskog algoritma s jednostavnom troturnirskom selekcijom(`steady state tournament`) algoritma u razvojnom okruženju ECF

```
<SteadyStateTournament>
 <Entry key="tsize">3</Entry>
</SteadyStateTournament>
```

Slika 5.5: `Algorithm` sekcija u ECFu za odabir željenog algoritma

`Algorithm` sekciji na sljedeći način prikazan na 5.5

5.3. Križanje

Kako bi se u ECFu implementirao operator križanja (engl. *crossover*) potrebno je naslijediti klasu `CrossoverOp` te implementirati zadanu metodu `mate(GenotypeP gen1, GenotypeP gen2, GenotypeP child)`. Implementacija križanja s jednom točkom prekida dana je u `MsaGenotypeCrsOnePoint`. Budući da dvije jedinke koje se križaju ne moraju imati sljedove jednake duljine, slučajno se odabire točka prekida na jedinci s kraćim slijedovima. Točka prekida slučajno se generira sve dok nije zadovoljen uvjet da jedinka sadrži nizove veće ili jednake duljine niza od najveće pronađene duljine niza prilikom čitanja datoteke FASTA.txt. Time se sprječava konvergencija genetskog algoritma prema rješenju koje biološki nema smisla kod poravnanja sljedova, iako ga evaluacijska funkcija može vrednovati kao bolje rješenje. Primjer križanja dan je na 5.6. Oznaka "|" predstavlja odabranu točku prekida. Dijete

```
Roditelj 1  Roditelj 2     Dijete
  AT|CG-A     GG|GC-TA     AT|GC-TA
  CG|T--G  +  TA|-GACG  =  CG|-GACG
  GA|TC--     CG|--GGA     GA|--GGA
```

Slika 5.6: Primjer križanja

u ovome slučaju sadrži genetskog materijal prvog roditelja do točke prekida, a od točke prekida sadrži genetski materijal drugog roditelja.

Kod funkcije `mate` dan je u nastavku:

Isječak koda 5.1: Funkcija `mate`

```
1   bool MsaGenotypeCrsOnePoint::mate(GenotypeP gen1, GenotypeP gen2, GenotypeP child
2   {
3       // dohvat roditelja i djeteta
4       MsaGenotype *p1 = (MsaGenotype *) (gen1.get());
5       MsaGenotype *p2 = (MsaGenotype *) (gen2.get());
6       MsaGenotype *ch = (MsaGenotype *) (child.get());
7
8       // generiraj tocku prekida na kracem slijedu
9       int cross_point = state_->getRandomizer()->
10        getRandomInteger(std::min(p1->alignment[0].size(),
11          p2->alignment[0].size()));
12
13      // slucajno odaberi raspored prepisivanja
14      // genetskog materijala roditelja
15      switch (state_->getRandomizer()->getRandomInteger(0, 1)) {
16      case 0: // prvo zapisi prvog roditelja
17
18      for(int i = 0; i < p1->alignment.size(); i++)
19      {
20        ch->alignment[i] = p1->alignment[i].substr(0, cross_point)
21          + p2->alignment[i].substr(cross_point);
22        std::cout << p1->alignment[i].substr(0, cross_point) +
23          p2->alignment[i].substr(cross_point) << "\n";
24      }
25      break;
26      case 1: // prvo zapisi drugog roditelja
27      for (int i = 0; i < p1->alignment.size(); i++)
28      {
```

```
29          ch->alignment[i] = p2->alignment[i].substr(0, cross_point)
30            + p1->alignment[i].substr(cross_point);
31        }
32        }
33
34    }
```

5.4. Mutacija

Operatori mutacije u ECFu ostvaruju se nasljeđivanjem razreda `MutationOp`. U implementaciji su ponuđene tri različita operatora mutacije: `MsaGenotypeMut`, `MsaGenotypeMutDeleteGap`, `MsaGenotypeMutInsertGap`. Svi autori se slažu kako su ključne mutacije brisanje i umetanje praznina, međutim implementiraju to na različite načine.

`MsaGenotypeMut` po uzoru na Shyu et al. (2004) slučajno odabire dva stupca te u svakome sljedu zamjenjuje sadržaj tih stupaca.

Isječak koda 5.2: Jednostavna mutacija izmjene stupaca

```
1   bool MsaGenotypeMut::mutate(GenotypeP gene) {
2    // dohvati jedinku
3    MsaGenotype* msa_alignment = (MsaGenotype*) (gene.get());
4
5    // odaberi prvu poziciju
6    int first_position = state_->getRandomizer()->
7    getRandomInteger(msa_alignment->
8    getAlignment().size());
9    // odaberi drugu poziciju
10   int second_position = state_->getRandomizer()->
11   getRandomInteger(msa_alignment->
12   getAlignment().size());
13
14     std::vector<std::string> &alignment = msa_alignment->
15     getAlignment();
16  // zamijeni stupce na pozicijama jedan i dva
17   for (int i = 0; i < alignment.size(); i++)
18   {
19     std::swap(alignment[i][first_position],
20      alignment[i][second_position]);
21   }
22
```

```
23    return true;
24  }
```

```
ACG-GTA  Mutacija  ACG-GTA
AGCGGTA ----------> ACGGGTA
-CAG--A             -CAG--A
```

Slika 5.7: Primjer djelovanja operatora jednostavne mutacije

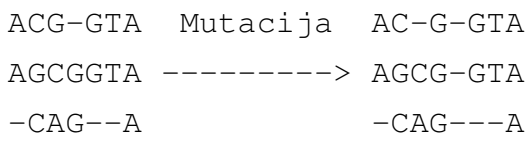
Primjer 5.7 ilustrira jednostavnu mutaciju. U prvome retku slučajno su odabrane dvije jednake pozicije te se mutacija nije dogodila. U drugome retku odabrane su pozicije 1 i 2, dok su utrećem retku odabrane pozicije 0 i 4. U prvome retku i trećem retku efekt je kao da se nije ni dogodila mutacija, dok mutacija u drugome retku osigurava bolje poravnanje.

Operator mutacije `MsaGenotypeMutInsertGap` za svaki slijed odabire slučajnu poziciju te umeće prazninu.

Isječak koda 5.3: Mutacija umetanja praznina

```
1   bool MsaGenotypeMutInsertGap::mutate(GenotypeP gene)
2   {
3       // dohvati jedinku
4       MsaGenotype* msa_alignment = (MsaGenotype*) (gene.get());
5       std::vector<std::string> &alignment = msa_alignment->
6       getAlignment();
7       // za svaki redak slucajno odaberi poziciju
8       // na odabranu poziciju umenti prazninu
9       for (int i = 0; i < alignment.size(); i++)
10      {
11        int pos = state_->getRandomizer()->
12        getRandomInteger(alignment[i].size() + 1);
13        alignment[i].insert(pos, "-");
14      }
15      return true;
16  }
```

Primjer 5.8 prikazuje djelovanje operatora umetanja praznina, koje za ovaj slučaj povećava vrijednost evaluacijske funkcije. U prvome retku praznina se umeće na poziciju 2, u drugome retku na poziciju 4, dok u trećem retku se umeće na poziciju 5. Operator `MsaGenotypeMutDeleteGap` prvo provjerava postoji li u svakome slijedu praznina. Ukoliko je taj uvjet zadovoljen, za svaki slijed se određuju indeksi



Slika 5.8: Primjer djelovanja operatora umetanja praznina

praznina, te nakon toga slučajno odabire jedan od pronađenih indeksa na kojem će se izbrisati praznina.

Isječak koda 5.4: Mutacija brisanja praznina

```
1  bool MsaGenotypeMutDeleteGap::mutate(GenotypeP gene)
2  {
3    // dohvati jedinku
4     MsaGenotype* msa_alignment = (MsaGenotype*) (gene.get());
5     std::vector<std::string> &alignment = msa_alignment->
6     getAlignment();
7     // prvjeri postoji li praznina u svakom retku
8     bool no_gap = false;
9     for (int i = 0; i < alignment.size(); i++)
10    {
11       if (alignment[i].find('-') == std::string::npos)
12       {
13          no_gap = true;
14          break;
15       }
16    }
17    // ako je pronadjen redak bez praznine
18    // ne obavljaj mutaciju
19    if (no_gap) return true;
20    for (int i = 0; i < alignment.size(); i++)
21    {
22      // pronadji indekse praznina
23       std::vector<int> indexes;
24       int br = 0;
25       for(auto& c : alignment[i])
26       {
27          if(c == '-')
28          {
29            indexes.push_back(br);
30          }
31          br++;
```

```
32          }
33          int pos = state_->getRandomizer()->
34          getRandomInteger(indexes.size());
35          // obrisi prazninu na slucajnoj poziciji
36          alignment[i].erase(alignment[i].begin() + indexes[pos]);
37          indexes.clear();
38      }
39      return true;
40  }
```

```
ACGG-GTA  Mutacija   ACGGGTA
AG-CGGTA ---------->  AGCGGTA
-CAG--A-              -CAG--A
```

Slika 5.9: Primjer djelovanja operatora brisanja praznina

Slika 5.9 prikazuje primjer djelovanja operatora brisanja praznina, zadovoljen je uvjet da svaki redak sadrži prazninu. U prvome retku briše se jedina moguća praznina, u drugome retku briše se praznina na poziciji 2, dok se u trećem retku briše ravnina na poziciji 7.

```
ACG-GTA  Mutacija  ACG-GTA
AGCGGTA ---------> AGCGGTA
-CAG--A            -CAG--A
```

Slika 5.10: Primjer djelovanja operatora brisanja praznina bez promjene

U primjeru na slici 5.10 nije zadovoljen uvjet da se u svakome retku nalazi praznina te stoga nema promjene nad jedinkom. Kada se nebi ispitivao uvjet da se u svakome retku nalazi praznina dobili bismo jedinku s retcima različitih duljina koju ne možemo vrednovati evaluacijskom funkcijom, niti bi takvo rješenje imalo smisla.

5.5. Evaluacijska funkcija

Evaluacijska funkcija određuje koliko je pojedino rješenje, to jest jedinka kvalitetno. Implementacija funkcije dobrote predstavlja najveći izazov prilikom rješavanja porav-

nanja više sljedova odjednom genetskim algoritmom. U literaturi sve funkcije dobrote svode se na metodu sume parova. Navedenu metodu moguće je ostvariti na dva načina:

1. Uspoređivanjem svakog para elemenata u određenom stupcu. Za preklapajuće dušične baze ukupnoj sumi se pridodaje jedan. Za neodgovarajuće baze sumi se dodaje -1. Ukoliko je bilo koji od uspoređivanih parova jednak praznini "-" sumi se dodaje -2. Ova jednostavna implementacija proširiva je korištenjem BLOSUM i PAM matrica za podršku poravnanja aminokiselina, a ne samo DNA i RNA sljedova.

2. Određivanjem elementa s najvećim brojem ponavljanja za svaki stupac te uspoređivanjem ostalih elemenata stupca s tim elementom.

U implementaciji rješenja korištena je jednostavna usporedba navedena pod točkom 1. bez podrške za aminokiseline. Slika 5.11 prikazuje primjer poravnanja reprezentiran matricom kako bi se opisao postupak izračuna funkcije dobrote. Primjer izračuna za 5.11 dan je u nastavku.

A	1	2	3	4
1	A	-	G	C
2	A	G	G	G
3	A	T	G	C

Slika 5.11: Primjer poravnanja za objašnjenje izračuna evaluacijske funkcije

$$SP(A_1) = s(A_{1,1}, A_{2,1}) + s(A_{1,1}, A_{3,1}) + s(A_{2,1}, A_{3,1}) = 1 + 1 + 1 = 3$$

$$SP(A_2) = s(A_{1,2}, A_{2,2}) + s(A_{1,2}, A_{3,2}) + s(A_{2,2}, A_{3,2}) = -5$$

$$SP(A_3) = s(A_{1,3}, A_{2,3}) + s(A_{1,3}, A_{3,3}) + s(A_{2,3}, A_{3,3}) = 3$$

$$SP(A_4) = s(A_{1,4}, A_{2,4}) + s(A_{1,4}, A_{3,4}) + s(A_{2,4}, A_{3,4}) = -1$$

$$SP(A) = SP(A_1) + SP(A_2) + SP(A_3) + SP(A_4) = 0$$

Kako bi implementirali funkciju dobrote u ECFu potrebno je naslijediti klasu `EvaluateOp` i implementirati metodu `FitnessP evaluate(IndividualP individual)` Implementacija metode dana je u nastavku:

Isječak koda 5.5: Implementacija evaluacijske funkcije

```
1  FitnessP MsaEvalOp::evaluate(IndividualP individual)
2  {
3    FitnessP fitness(new FitnessMax);
4    MsaGenotype::MsaGenotype* msagen = (MsaGenotype::MsaGenotype*) individual->
5      getGenotype().get();
6    std::vector<std::string> alignment = msagen->getAlignment();
7    int sum = 0;
8    int num_of_columns = alignment[0].size();
9    int num_of_sequences = alignment.size();
10   for (int column = 0; column < num_of_columns; column++)
11   {
12     for (int i = 0; i < alignment.size() - 1; i++)
13     {
14       std::string current_sequence = alignment[i];
15       if (current_sequence[column] == '-')
16       {
17         sum += (num_of_sequences - i - 1) * gaps;
18         continue;
19       }
20       for (int j = i+1; j < alignment.size(); j++)
21       {
22         if (current_sequence[column] == alignment[j][column])
23           sum += match;
24         else if (alignment[j][column] == '-')
25           sum += gaps;
26         else
27           sum += mis_match;
28       }
29   }
30   fitness->setValue(sum);
31 }
```

Kako bi se postignula što kvalitetnija implementacija koja bi se čim više približila alatim poput Clustal Omega potrebno je implementirati funkciju dobrote koristeći biološka saznanja.

6. Usporedba ostvarenih rezultata

Na sljedećoj slici 6.1 vidljiv je rezultat pokretanja implementacije genetskog algoritma za prethodno navedenu FASTA.txt datoteku. Operatori mutacije biraju se jednakim vjerojatnostima svakog puta, to se moglo postići i ne navođenjem nijednog operatora mutacije u konfiguracijskoj datoteci.

```
Best of run:
<HallOfFame size="1">
  <Individual size="1" gen="0">
    <FitnessMax value="-48"/>
    <MsaGenotype size="3">
ATTGCCG-ACT-
--------AC--
-GACCCT-AG--
    </MsaGenotype>
  </Individual>
</HallOfFame>
```

Slika 6.1: Poravnanje FASTA.txt koristeći sva tri operatora mutacije

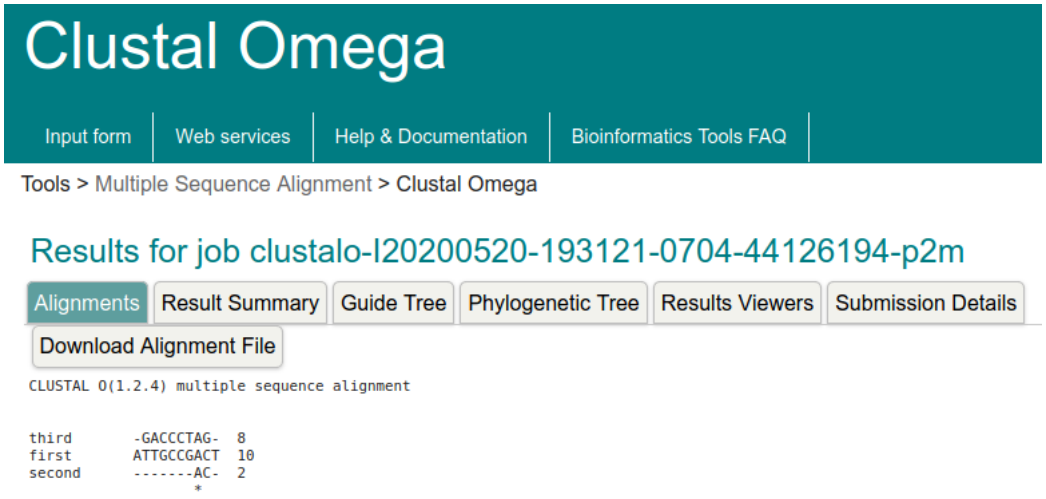
Sljedeća slika prikazuje poravnanje iste ulazne datoteke koristeći alat Clustal Omega

Vidljivo je kako implementacija genetskog algoritma ne pronalazi globalni maksimum, već ostane na pronađenome lokalnome maksimumu.

Prethodni rezultati dobiveni su genetskim algoritmom koristeći operatore mutacije s jednakom vjerojatnošću.

Sljedeće slike prikazuju rezultate izvođenja kada se koristi samo operator mutacije brisanja praznina.

Iz slike 6.3 vidljivo je kako u malenome primjeru korištenjem samo operatora mutacije, postiže se rezultat s većom vrijednošću funkcije dobrote. Naravno, u primjerima gdje nije velika razlika u početnim duljinama sljedova, operator mutacije brisanja neće



Slika 6.2: Poravnanje FASTA.txt pomoću Clustal Omege

znatno doprinijeti rješenju. Shodno tome bolje rješenje na velikim primjerima postići će se korištenjem svih triju operatora mutacije.

Testiranje korištenjem samo operatora mutacije umetanja praznina ili samo operatora zamjene dva stupca rezultiraju na jednaki način. Jedinke dobivene operatorima mutacije imaju manje vrijednosti evaluacijske funkcije u usporedbi s inicijalno generiranom populacijom. Zbog toga što umetanje praznina na bilo koje mjesto samo umanjuje vrijednost funkcije dobrote, iako uz kombinaciju s drugim operatorima ta jedinka s umetnutim prazninama može pridonjeti pronalasku novog najboljeg pronađenog rješenja. Ostale kombinacije svih triju operatora mutacije, gdje je vjerojatnost odabira pojedinog dvostruko veća od sume vjerojatnosti preostala dva operatora rezultiraju na relativno jednaki način, te ne pridonose značajno u pronalasku boljeg rješenja.

Slika 6.4 prikazuje poravnanje DNA MYH16 sljedova genetskim algoritmom.

Slika 6.5 prikazuje poravnanje DNA MYH16 s Clustal Omegom.

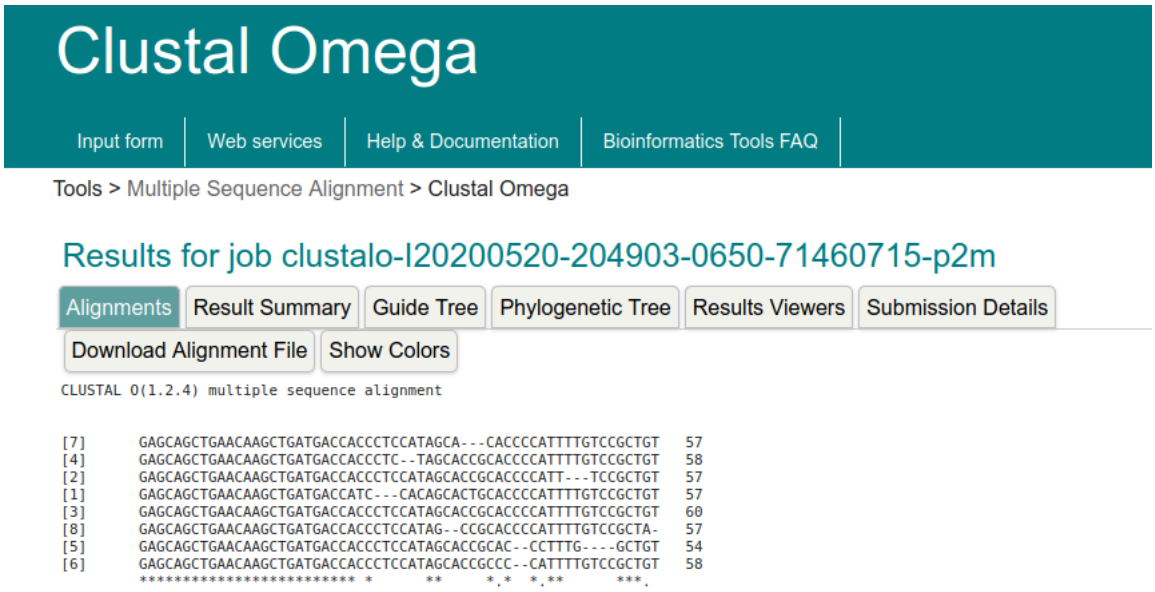
Usporedbom 6.5 s 6.4 vidljivo je kako je implementacija Clustal Omege tim više bolja za veće primjere koje je ptrebno rješavati u praksi. Genetski algoritam mogao bi se unaprijediti korištenjem još većeg broja operatora mutacije i dodatnih operatora križanja te ono najbitnije korištenjem naprednije funkcije dobrote koja bi koristila biološko predznanje.

```
Best of run:
<HallOfFame size="1">
  <Individual size="1" gen="1">
    <FitnessMax value="-37"/>
      <MsaGenotype size="3">
ATTGCCGACT
--AC------
GGACCCT-AG
</MsaGenotype>
  </Individual>
</HallOfFame>
```

Slika 6.3: Poravnanje FASTA.txt koristeći samo operator mutacije brisanja praznina

```
Best of run:
<HallOfFame size="1">
  <Individual size="1" gen="19">
    <FitnessMax value="-875"/>
      <MsaGenotype size="8">
-GAGCAGCTGA-ACAAGCTGATGACCACCCTCCATAG—CCGCACCCCATTTTGTCCGCTATTATCCCCAATGAG-TTAATG-TCG--G
GAG-CAGCTGAAC-AAGC--TGATGACCACCCTCTAGCACCGCACCCCATTTTGTCCGCTGTATTATCCCCAATGAGTTTGCAATCGG
-GAGCAGCTGAACA-A-GCTGATGACCACCCTCCATAGCACCGCACCCTTTGGCTGTATT-ATCCCCAATGAGTTT-AAGCAAT-CGG
GAGCAG---C--TGAACAAGCTGATGACCATCCACAGCACTGCACCCCATTTTGTCCGCTGTATTGTGCCCAATGAGTTTAGTCA--G
GAGCAGCTGAA-CAAGCTGATGACC-ACCCT-CCATAGCAC-CGCACCCCATTTCCGCTGTATTGTCCCCAATGAGTTTTAAT--CGG
G-AG-CAGCTGAACAAG-CTGATGACCACCCTCCATAGCACACCCCATTTTGTCCGCTGTATTATCCCCAATGAGTTTTGCAATC-GG
GAGCAGCTGAA-C-AAGC-TGATGACCACC-CTCCATAGCACCGCCCCATTTTGTCCGCTGTATTATCCCCAATGAGTTTCAATCG-G
GAGCAGCTGAACAAGCTGATGACCACCCTCCATAGCACCGCACCCCATTTTGTCCGCTGTATTGTCCCCAATGAGTTTAAGCAATCGG
</MsaGenotype>
  </Individual>
</HallOfFame>
```

Slika 6.4: Poravnanje DNA MYH16 genetskim algoritmom



Slika 6.5: Poravnanje DNA MYH16 koristeći Clustal Omega

7. Zaključak

U ovom radu dan je kratak uvod u potrebu razvijanja evolucijskog računarstva te time i genetskog algoritma. Predstavljeni su optimizacijski problemi iz područja bioinformatike te je predloženo rješenje genetskim algoritmom.

Opisan je i implementiran problem poravnanja više sljedova odjednom genetskim algoritmom koristeći razvojno okruženje za evolucijsko računanje ECF razvijeno na Fakultetu elektrotehnike i računarstva u Zagrebu.

Iako genetski algoritam postiže dobre rezultate za poravnanje manjeg broja i manje duljine DNA lanaca, razlika između poravnanja uz pomoć alata Clustal Omega koji se temelji na progresivnoj metodi poravnanja postaje veća s većim primjerima .

LITERATURA

[1] Wikipedia gene prediction. . URL https://en.wikipedia.org/wiki/File:Gene-structure.svg.

[2] Wikipedia orf. . URL https://en.wikipedia.org/wiki/Multiple_sequence_alignment.

[3] Shyu C., Sheneman L., i Foster J.A. Multiple sequence alignment with evolutionary computation. 2004.

[4] Wang C. i Lefkowitz E.J. Genomic multiple sequence alignments: Refinement using a genetic algorithm. 2005.

[5] Hernandez D., Grass R., i Appel R. Model: an efficient strategy for ungapped local multiple alignment. 2004.

[6] Gary B. Fogel i David W. Corne. *Evolutionary Computing in Bioinformatics*. Morgan Kufmann Publishers, 2003.

[7] Marin Golub. *Genetski algoritam Drugi dio*. Fakultet elektroethnike i računarstva, Zagreb, 2004. URL http://www.zemris.fer.hr/~golub/ga/ga_skripta2.pdf.

[8] Marin Golub. *Genetski algoritam Prvi dio*. Fakultet elektroethnike i računarstva, Zagreb, 2010. URL http://www.zemris.fer.hr/~golub/ga/ga_skripta1.pdf.

[9] Domagoj Jakobović. *Evolutionary Computation Framework*. URL http://ecf.zemris.fer.hr/.

[10] Thompson J.D. Clustal w: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. 1994.

[11] Horng J.T., Wu L.C., i Lin C.M. A genetic algorithm for multiple sequence alignment. 2005.

[12] Bojana Dalbelo-Bašić Marin Golub, Marko Čupić. *Neizrazito, evolucijsko i neuroraƒunarstvo*. Fakultet elektroethnike i računarstva, Zagreb, 2013. URL http://java.zemris.fer.hr/nastava/nenr/knjiga-0.1.2013-08-12.pdf.

[13] Amie Judith Radenbaugh. Applications of genetic algorithms in bioinformatics. 2008. URL https://scholarworks.sjsu.edu/cgi/viewcontent.cgi?article=4491&context=etd_theses.

[14] Marko Čupić. *Prirodom inspirirani optimizacijski algoritmi. Metaheuristike.* Fakultet elektroethnike i računarstva, Zagreb, 2013. URL http://java.zemris.fer.hr/nastava/pioa/knjiga-0.1.2013-12-30.pdf.

[15] Mile Šikić i Mirjana Domazet-Lošo. *Bioinformatika*. Fakultet elektroethnike i računarstva, Zagreb, 2013. URL https://www.fer.unizg.hr/_download/repository/bioinformatika_skripta_v1.2.pdf.

Primjena evolucijskih algoritama u bioinformatici

Sažetak

U ovom radu dan je uvod u evolucijsko računarstvo i genetski algoritam. Opisani su optimizacijski problemi iz bioinformatike. Koristeći Evolutionary Computation Framework genetskim algoritmom pristupilo se problemu poravnanja više sljedova odjednom. Pokazani su nedostaci primjene genetskog algoritma na danome problemu te je ispitana ovisnost o vjerojatnostima operatora mutacije.

Ključne riječi: genetski algoritam, optimizacijski problemi, bioinformatika, poravnanje više sljedova odjednom, Evolutionary Computation Framework

Applications of evolutionary algorithms in bioinformatics

Abstract

In this paper an intro to evolutionary computing and genetic algorithm is given. Optimization problems from bionformatics are described. The problem of multiple esquence alignment is solved with genetic algorithm in Evolutionary Computation Framework. Flaws of genetic algorithm are stated on the given problem and the relationship between the probabilities of mutation operators and the quality of solution is estimated..

Keywords: genetic algorithm, optimization problems, bioinformatics, multiple sequence alignment, Evolutionary Computation Framework