UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Milena Dukanac

JEZIK ELIXIR SA PRIMENOM U SEKVENCIONIRANJU GENOMA

master rad

Beograd, 2019.

Mentor:

dr Milena Vujošević Janičić, docent
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

dr Jovana Kovačević, docent
Univerzitet u Beogradu, Matematički fakultet

dr Vesna Marinković, docent
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: _____

Mami i tati za nesebičnu ljubav, podršku i razumevanje

Naslov master rada: Jezik Elixir sa primenom u sekvencioniranju genoma

Rezime: Cilj ovog rada je upoznavanje sa osnovama funkcionalnog programskog jezika Elixir i važnošću oblasti sekvencioniranja genoma. Ideja je da kroz implementaciju algoritama, koji se koriste u rešavanju problema iz ove oblasti, uočimo koje su mogućnosti i prednosti ovog jezika, te da uz pomoć istih olakšamo njihovo rešavanje. Rezultati rada pokazuju da je Elixir dobar izbor i da obezbeđuje pouzdanu i brzu obradu velike količine podataka koju je potrebno obraditi kada su u pitanju problemi iz ove oblasti.

Ključne reči: Programski jezik Elixir, konkurentnost, sekvencioniranje genoma, asembliranje, očitavanje, k-mer

Sadržaj

Glava 1

Uvod

U oblasti razvoja programskih jezika stalno se dešavaju promene. Velika pažnja i značaj se pridaje razvoju novih programskih jezika koji omogućavaju brže i lakše programiranje, bržu obradu velike količine podataka i podršku za konkurentne procese. Upravo sa tim ciljem, u toku prethodnih godina nastao je veliki broj novih programskih jezika i razvojnih okruženja za njih. Jedan od novih jezika koji je nastao 2011. godine je ***Elixir***. Za veoma kratko vreme postao je izuzetno popularan i danas ga veliki broj kompanija koristi za svoje projekte i softverska rešenja. Neki od razloga za popularnost Elixir-a su brza obrada velike količine podataka, skalabilnost, podrška za konkurentne procese i veoma visoka tolerancija na greške.

Rekonstrukcija genoma pomoću sekvencioniranja genoma je važan problem. Sekvencioniranje genoma omogućava istraživačima da proučavaju ne samo gene koji kodiraju važne proteine u našem organizmu, već i čitave oblasti DNK koje imaju druge važne uloge. Naši genomi ostaju relativno konstantni tokom čitavog života, tako da je genom dovoljno sekvencionirati jedanput kako bi se mogao proučavati iznova i iznova. U medicini, jedna od oblasti gde je sekvencioniranje genoma od izuzetnog značaja je dijagnostikovanje rizika od naslednih bolesti. Na primer, otkrivanje rizika od razvoja određenog karcinoma će uticati na obavljanje češćih skrininga. Češći skrininzi će pomoći da se bolest na vreme otkrije, što je od presudnog značaja za proces lečenja.

Cilj ovog rada je prikazivanje osnovnih koncepata i osobina programskog jezika Elixir kroz primenu u implementaciji algoritama koji se koriste u oblasti sekvencioniranja genoma. Takođe, cilj je i upoznavanje sa osnovnim pojmovima i problemima ove oblasti, kao i prikazivanje značaja njenog proučavanja i unapređivanja.

Poglavlje 2 je posvećeno Elixir-u. Na početku je prikazano razvojno stablo na

kome se može videti koji su jezici uticali na njegov nastanak i razvoj, kao i godine nastanka tih jezika. Zatim su predstavljeni osnovni tipovi podataka i osnovne osobine Elixir-a, zajedno sa primerima koda radi boljeg razumevanja. Poglavlje 3 je posvećeno oblasti sekvencioniranja genoma. Uvodni deo ovog poglavlja služi upoznavanju osnovnih pojmova iz ove oblasti i njenom istorijskom razvoju i napredovanju. Nakon toga su obrađeni osnovni koraci koji se sprovode u toku sekvencioniranja genoma, kao i algoritmi koji pomažu u ovom procesu. Poglavlje 4 je namenjeno opisu implementacije algoritama koji su obrađeni u okviru ovog rada. U prvom delu dati su pseudokodovi algoritama zajedno sa razmatranjem njihove vremenske i prostorne složenosti. Potom sledi deo koji je posvećen opisu aplikacije i načinu pokretanja algoritama. Na kraju poglavlja prikazani su rezultati izvršavanja algoritama. Poslednje poglavlje 5 sumira rezultate i daje osnovne zaključke rada.

Glava 2

Elixir

Elixir je funkcionalan programski jezik nastao 2011. godine. Njegovim tvorcem se smatra Hosé Valim (engl. *José Valim*). Elixir je dizajniran za izgradnju skalabilnih i lako održivih aplikacija. Poseduje jednostavnu i modernu sintaksu. Zbog svoje funkcionalne prirode, izuzetno dobre podrške za rad u distrubiranim sistemima i tolerancije na greške koja je na jako visokom nivou, u Elixir-u je rađeno mnogo zanimljivih projekata iz oblasti robotike. Takođe se uspešno koristi u razvoju veba i u domenu softvera za uređaje sa ugrađenim računarom (engl. *embedded sotfware*).

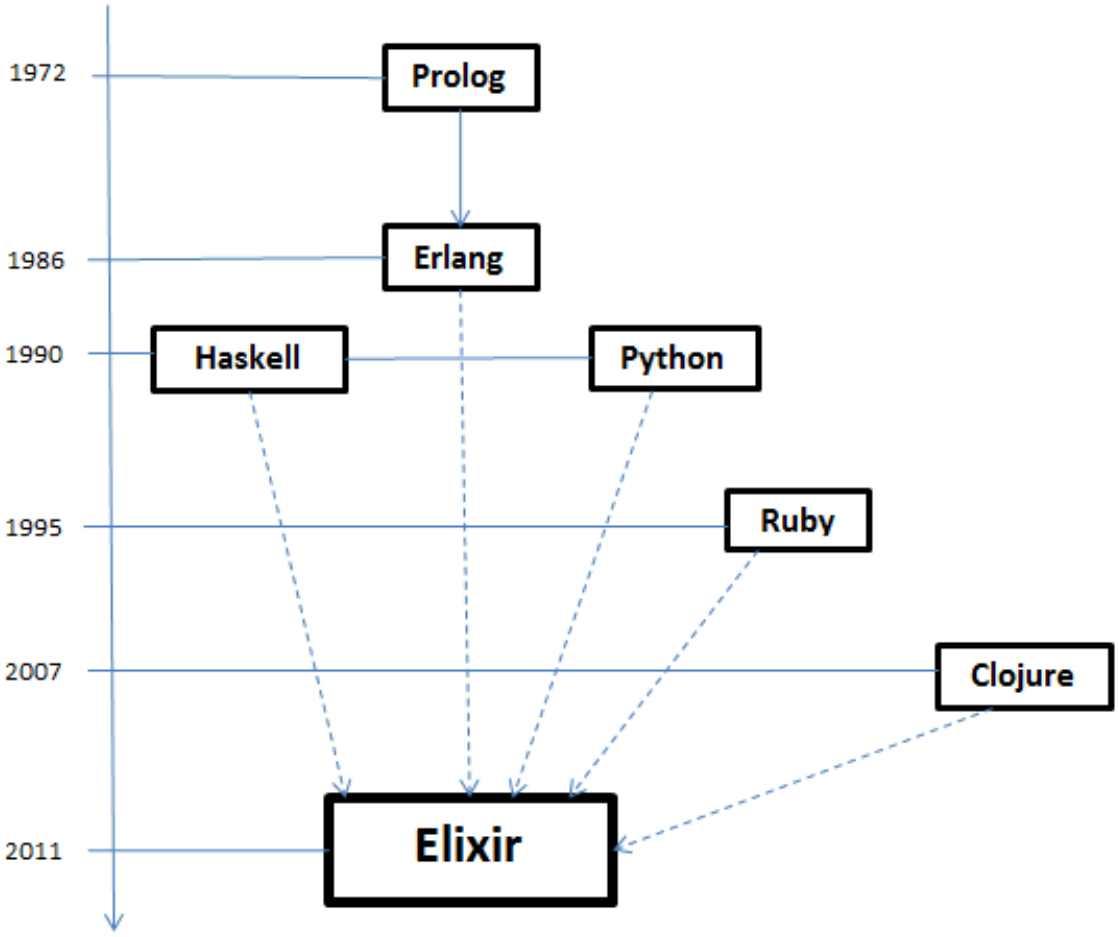
2.1 Razvojno stablo

Na nastanak Elixir-a je uticao programski jezik **Erlang**. Pri njegovom kreiranju značajnu ulogu u smislu sintakse imao je programski jezik **Ruby**, a iz jezika kao što su **Python, Haskell** i **Clojure** je preuzeo mnoge koncepte. Razvojno stablo jezika Elixir može se videti na slici 2.1.

Erlang

Firma Erikson je 1981. godine oformila novu laboratoriju **Erikson CSLab** (engl. *The Ericsson CSLab*) sa ciljem da predlaže i stvara nove arhitekture, koncepte i strukture za buduće softverske sisteme [2]. Jedan od zadataka novonastale laboratorije bio je dodavanje konkurentnih procesa u programski jezik **Prolog**[1] i

[1]Prolog (engl. ***PRO****gramming in* ***LOG****ic*) je deklarativan programski jezik namenjen rešavanju zadataka simboličke prirode. Prolog se temelji na teorijskom modelu logike prvog reda. Početkom 1970-ih godina **Alen Kolmerur** (engl. *Alain Colmerauer*) i **Filip Rusel** (engl. *Philippe Roussel*) na Univerzitetu u Marselju, zajedno sa **Robertom Kovalskim** (engl. *Robert Kowalski*) sa



Slika 2.1: Razvojno stablo jezika Elixir

njegovo unapređivanje. Prolog predstavlja začetak novog programskog jezika koji je 1987. godine nazvan **Erlang**. Ime je nastalo zahvaljujući inicijativi zaposlenih koji su radili na telefonskim prekidačima, a za koje je jezik dizajniran. Naime, oni su predložili da jezik nosi ime Erlang u čast danskom matematičaru i inženjeru Agneru Krarupu Erlangu (engl. *Agner Krarup Erlang*), a što je ujedno odgovaralo i skraćenici od „**Er**icsson **Lang**uage". Erlang se smatrao dijalektom Prologa sve do 1990. godine, kada je postao potpuno samostalan programski jezik sa sopstvenom sintaksom. Međutim, zadržao je neke delove sintakse i koncepte iz Prologa (promenljive počinju velikim slovom, svaka funkcionalna celina se završava tačkom, poklapanje obrazaca (engl. *pattern matching*)).

Nakon mnogo godina rada nastajale su sve brže, bolje i stabilnije verzije jezika,

Departmana veštačke inteligencije na Univerzitetu u Edinburgu , razvili su osnovni dizajn jezika Prolog.

kao i **standardna biblioteka OTP** (engl. *The Open Telecom Platform*) [21]. Od decembra 1998. godine, kada su postali deo slobodnog softvera (engl. *open source software*), Erlang i OTP se mogu slobodno preuzeti sa zvaničnog sajta jezika Erlang [21]. Erlang dobija široko prihvatanje pojavom višejezgarnih procesora i njihovog novog skalabilnog pristupa konkurentnosti. Erlang je funkcionalan jezik idealan za svaku situaciju u kojoj su paralelnost, tolerancija na greške i brz odziv neophodni [4], te se koristi u velikom broju kompanija za razvoj njihovih glavnih softverskih rešenja (npr. Erikson (engl. *Ericsson*), Motorola, Votsap (engl. *WhatsApp*), Jahu (engl. *Yahoo!*), Fejsbuk (engl. *Facebook*)).

Elixir je preuzeo izmenjenu Erlangovu sintaksu i dopunjenu Erlangovu standardnu biblioteku. Pokreće se na vituelnoj mašini jezika Erlang, što znači da je nasledio i sve karakteristike Erlang platforme koja postoji već godinama i koja se pokazala pouzdanim rešenjem za skalabilne aplikacije.

Python

Python je interpretirani jezik opšte namene čiji tvorac je Gido van Rosum (engl. *Guido van Rossum*) [23]. Krajem 1980-ih je koncipiran kao naslednik jezika **ABC** [25], a prvi put je objavljen 1991. godine. Filozofija dizajna jezika Python naglašava čitljivost koda. Njegove jezičke konstrukcije i objektno-orijentisani pristup imaju za cilj da pomognu programerima da napišu jasan i logičan kôd za male i velike projekte. Python je dinamički tipiziran jezik i poseduje sistem za prikupljanje smeća (engl. *garbage collector*). Podržava više paradigmi programiranja uključujući proceduralno, objektno-orijentisano i funkcionalno programiranje. Python interpreteri su dostupni za mnoge operativne sisteme. Globalna zajednica programera razvija i održava referentnu implementaciju otvorenog koda **CPython**. Neprofitna organizacija *The Python Software Foundation* upravlja i usmerava resursima za razvoj jezika Python i CPython. Jedna od osobina koje je Elixir nasledio od Python-a je podrška za dokumentaciju u vidu dokumentacionih stringova (engl. *docstrings*) koji omogućavaju povezivanje dokumentacije sa modulima, funkcijama, klasama, metodama.

Haskell

Haskell je čisto funkcionalni jezik koji je statički tipiziran [22]. Nazvan je po Haskelu Bruks Kariju (engl. *Haskell Brooks Curry*), čiji rad u oblasti matematičke logike služi kao osnova za sve funkcionalne jezike. Haskell je zasnovan na lambda računu, pa se stoga lambda koristi kao logo jezika. Nudi kratak, jasan i održiv kôd, mali procenat grešaka i veliku pouzdanost. Stoga je pogodan za pisanje velikih softverskih sistema, jer njihovo održavanje čini lakšim i jeftinijim. Jedna od karakteristika koje je Elixir preuzeo od ovog jezika je lenjo izračunavanje.

Ruby

Ruby je dinamički tipiziran programski jezik otvorenog koda nastao 1995. godine. Kod ovog programskog jezika fokus je na jednostavnosti i produktivnosti. Ruby ima elegantnu sintaksu koja je prirodna za čitanje i lako pisanje. Ruby je interpretirani programski jezik, što znači da se izvorni kôd prevodi u kôd razumljiv računaru prilikom svakog izvršavanja programa. Interpretirani programski jezici su sporiji od kompajliranih, ali su fleksibilniji i potrebno je kraće vreme za izradu programa. Međutim, sve više iskusnih Ruby programera se okreće Elixir-u. Zapravo, Elixir je prvi jezik nakon Ruby-ja koji zaista brine o lepoti koda i korisničkom iskustvu vezanom za jezik, biblioteke i ekosistem.

Ruby je imao veliki uticaj na sintaksu programskog jezika Elixir. Na slici 2.2 se nalaze delovi koda napisani u jeziku Ruby i jeziku Elixir, koji imaju dosta sličnosti, a čiji je rezultat izvršavanja isti - dva stringa su nadovezana. U jeziku Ruby se definiše klasa *Concat* koja ima polje *value*, funkciju *initialize* koja se poziva pri kreiranju objekta klase radi inicijalizacije polja *value* i funkciju *join* koja vrši nadovezivanje dva stringa. U Elixir-u se umesto klase definiše modul *Concat* koji sadrži samo funkciju *join* koja vrši nadovezivanje dva stringa. Uočavaju se sličnosti u sintaksi koje su ilustrovane ovim primerom pri definisanju klasa/modula, funkcija (ključne reči *def* i *end*), zatim pri nadovezivanju stringova (operatori + i <>) i pozivanju funkcija (ime klase/modula za kojim sledi tačka).

Clojure

Clojure je dinamički tipiziran programski jezik opšte namene nastao 2007. godine. Njegov tvorac je Rič Hiki (engl. *Rich Hickey*). Clojure kombinuje pristupačnost i interaktivni razvoj skriptnog jezika sa efikasnom i robusnom infrastrukturom

Ruby

```ruby
class Concat
  attr_reader :value

  def initialize(value)
    @value = value
  end

  def join(another_value)
    value + another_value
  end
end

irb> Concat.new("Yo").join("!!!")
irb> => "Yo!!!"
```

Elixir

```elixir
defmodule Concat do
  def join(value, another_value) do
    value <> another_value
  end
end

iex> Concat.join("Yo", "!!!")
iex> "Yo!!!"
```

Slika 2.2: Kôd za nadovezivanje dva stringa u jezicima Ruby i Elixir

za višenitno programiranje. On je kompajlirani jezik, ali je i dalje potpuno dinamički tipiziran - svaka funkcija koju podržava Clojure je podržana u toku izvršavanja. Predstavlja dijalekt Lisp-a[2] i deli njegovu filozofiju *code-as-data* (program je funkcija koja se izvršava nad podacima) i moćan makro sistem. Clojure je pretežno funkcionalni programski jezik i sadrži bogat skup nepromenljivih i postojanih struktura podataka. Elixir je preuzeo neke od najboljih Clojure karakteristika - efikasne, nepromenljive strukture podataka, opcionalno lenjo izračunavanje i protokole.

2.2 Uputstvo za instalaciju i pokretanje Elixir-a

Za instaliranje Elixir-a je potrebno posetiti zvanični sajt jezika Elixir [15] i pratiti uputstva za odgovarajući operativni sistem. Kada se instalacija završi, može se proveriti verzija Elixir-a pomoću komande *elixir –version* izvršene u komandnoj liniji.

Elixir programi imaju ekstenziju *.ex* ili *.exs* i mogu se pisati i izvršavati u okviru odgovarajućih integrisanih razvojnih okruženja. Neki od najpoznatijih su *Spacemacs*, *Visual Studio Code*, *Emacs* i *Atom*, a čitav spisak, kao i link ka njihovim zvaničnim stranicama, može se videti na adresi [17].

[2]Lisp je programski jezik zasnovan na matematičkoj teoriji rekurzivnih funkcija (u kojoj se funkcija pojavljuje u sopstvenoj definiciji), a Lisp program je funkcija koja se primenjuje na podatke. Ime LISP je nastalo od „LISt Processor", a povezane liste su jedan od glavnih tipova podataka. Osnova Lisp-a je funkcionalno programiranje, ali se Lisp zbog raznih drugih svojstava smatra multiparadigmatskim programskim jezikom.

Pojedinačni izrazi, ali i Elixir programi, mogu se takođe izvršavati u komandnoj liniji nakon izvršavanja komande *iex* i pokretanja interaktivnog Elixir-a. Primer pokretanja interaktivnog Elixir-a i izvršavanja pojedinačnog izraza može se videti na listingu 2.1.

```
1 Microsoft Windows [Version 6.1.7601]
2 Copyright (c) 2009 Microsoft Corporation. All rights reserved.
3
4 C:\Windows\system32>iex
5 Interactive Elixir (1.8.0) - press Ctrl+C to exit (type h() for
     help)
6 iex(1)> 2 + 2
7 4
```

Listing 2.1: Pokretanje interaktivnog Elixir-a

U Elixir-u se sve funkcije definišu u okviru modula pomoću ključne reči *defmodule*. Za kompajliranje Elixir fajla u kome je definisan modul sa funkcijama i pokretanje interaktivnog Elixir-a nakon toga, najpre se poziva komanda *iex imeFajla.ex*. Potom se može pokrenuti bilo koja funkcija unutar definisanog modula tako što se navede ime modula i ime funkcije razdvojeni tačkom, za kojima slede argumenti u okviru zagrada (*imeModula.imePrograma(argument1, argument2)*). Primer definisanja modula i funkcije u okviru njega može se videti na listingu 2.2, a primer kompajliranja fajla i pokretanja funkcije dat je na listingu 2.3.

```
1 defmodule Primer do
2   def odstampaj_argumente(arg1, arg2) do
3     IO.puts(arg1 <> " " <> arg2)
4   end
5 end
```

Listing 2.2: Primer definisanja modula

```
1 C:\Windows\system32>iex odstampaj_argumente.ex
2 Interactive Elixir (1.8.0) - press Ctrl+C to exit (type h() for
     help)
3 iex(1)>Primer.odstampaj_argumente("Zdravo", "svete")
4 Zdravo svete
5 :ok
```

Listing 2.3: Primer kompajliranja fajla i pokretanja programa

Elixir programi koji imaju ekstenziju *.exs* se mogu kompajlirati i izvršavati iz komandne linije pomoću komande *elixir imePrograma.exs*. Nakon ove komande neće

biti pokrenut interaktivni Elixir, već će samo biti prikazani rezultati izvršavanja programa.

2.3 Osnovne karakteristike

Hosé Valim je tokom 2010. godine bio zaposlen u kompaniji *Platformatec* [20] i radio je na poboljšanju performansi okruženja *Ruby on Rails* na višejezgarnim sistemima. Shvatio je da Ruby nije bio dovoljno dobro dizajniran da reši problem konkurentnosti, pa je započeo istraživanje drugih tehnologija koje bi bile prihvatljivije. Tako je otkrio Erlang i upravo ga je interesovanje prema virtuelnoj mašini jezika Erlang podstaklo da započne pisanje jezika Elixir. Uticaj projekta na kome je do tada radio odrazio se na to da Elixir ima sintaksu koja je nalik na sintaksu jezika Ruby. Ovaj jezik se pokazao veoma dobro pri upravljanju milionima simultanih konekcija: u 2015. je zabeleženo upravljanje nad dva miliona *WebSocket* konekcija, dok je u 2017. za skalirani Elixir zabeležena obrada pet miliona istovremenih korisnika. Elixir se danas koristi u velikim kompanijama, kao što su *Discord* i *Pinterest* [9].

Elixir je dinamički tipiziran, funkcionalni programski jezik koji se pokreće na vituelnoj mašini jezika Erlang, pa samim tim i nasleđuje pogodna svojstva koje dolaze sa ovim okruženjem kao što su **konkurentnost** i **tolerisanje grešaka** [13]. Elixir je nadomestio mnoge koncepte koji su nedostajali jeziku Erlang. Neki od njih su **metaprogramiranje**[3], **polimorfizam**, **makroi** i **podrška za alate**. Elixir poseduje podrazumevano okruženje, takozvani **Kernel**, koji obezbeđuje podršku za osnovne tipove i funkcionalnosti jezika.

U ovom delu će biti opisani osnovni tipovi jezika Elixir, njegove osobine, osnove njegove sintakse, semantike, kao i podrška za osnovne koncepte funkcionalnih jezika poput poklapanja obrazaca i nepromenljivosti podataka.

[3]Tehnika koja omogućava da programi posmatraju druge programe kao svoje podatke i na taj način čitaju i modifikuju i njihov i svoj kôd u vreme izvršavanja.

2.4 Osnovni tipovi podataka

Elixir ima svoje ugrađene (primitivne) tipove. To su:

1. Atomi
2. Celi brojevi
3. Brojevi u pokretnom zarezu
4. Portovi
5. Ugrađene torke
6. Liste
7. Mape
8. Funkcije
9. Niske bitova
10. Reference

Svaki od ovih tipova, osim poslednja dva, ima odgovarajuće module koji sadrže funkcije koje se koriste za operacije nad tim tipom. Oni predstavljaju omotač oko primitivnog tipa koji nam omogućava korišćenje dodatnih funkcionalnosti nad njim. U nastavku će biti opisani neki od osnovnih tipova.

Atomi

Atomi su konstante ili simboli, pri čemu njihovo ime predstavlja njihovu vrednost. Počinju dvotačkom (:) i mogu sadržati slova, cifre, simbole _, @. Mogu se završavati sa ! i ?. Atomi se mogu naći svuda u Elixir-u. U *listama ključnih reči* koje će biti opisane u odeljku 2.4, predstavljaju prvu vrednost elementa liste i često se koriste da označe uspeh (*:ok*) ili grešku (*:error*).

Celi brojevi

Celi brojevi u Elixir-u su predstavljeni slično kao i u većini programskih jezika i mogu biti dekadni, heksadekadni, oktalni i binarni. Karakter _ se može koristiti za odvajanje blokova cifara. Veoma značajna stvar je da ne postoji fiksna veličina za čuvanje celih brojeva u memoriji, već interna reprezentacija raste kako bi broj mogao biti smešten u potpunosti.

Brojevi u pokretnom zarezu

Brojevi u pokretnom zarezu se zapisuju uz pomoć decimalne tačke po standardu *IEEE 754*. Pre i posle decimalne tačke mora biti najmanje jedna cifra (1.0, 0.2456). Može se koristiti i notacija koja obuhvata navođenje eksponenata ($0.314159e1$, $314159.0e-5$).

Liste

Liste se čuvaju u memoriji kao povezane liste, što znači da svaki element u listi čuva svoju vrednost i ukazuje na sledeći element sve dok se ne dostigne kraj liste. To znači da je pristup proizvoljnom elementu liste kao i određivanje dužine liste linearna operacija, jer je potrebno da prođemo celu listu da bismo odredili njenu dužinu. Slično, performanse spajanja dve liste zavise od dužine one koja se nalazi sa leve strane, jer se spajanje vrši nadovezivanjem liste sa desne strane na kraj liste koja se nalazi sa leve strane. Da bi se izvršilo nadovezivanje, mora se odrediti dužina liste sa leve strane.

Elixir koristi uglaste zagrade ([ ]) da označi listu vrednosti. Vrednosti mogu biti bilo kog tipa, a primer liste sa vrednostima različitih tipova prikazan je na listingu 2.4.

```
1 iex(1)>[1, 2, true, 3]
2 [1, 2, true, 3]
3 iex(2)>length([1, 2, true, 3])
4 4
```

Listing 2.4: Primer liste

Nadovezivanje ili oduzimanje dve liste korišćenjem operatora $++/2$ i $--/2$ prikazano je na listingu 2.5. Rezultat oduzimanja dve liste je nova lista koja nastaje tako što se iz liste sa leve strane operatora $--$ eliminišu elementi koji se nalaze u listi sa desne strane.

```
1 iex(1)>[1, 2, 3] ++ [4, 5, 6]
2 [1, 2, 3, 4, 5, 6]
3 iex(2)>[1, false, 2, true, 3, false]  - - [true, false]
4 [1, 2, 3, false]
```

Listing 2.5: Nadovezivanje i oduzimanje dve liste

Operatori za rad nad listama nikada ne menjaju postojeću listu. Rezultat povezivanja listi ili uklanjanja elemenata iz liste je uvek nova lista, jer su strukture

podataka u Elixir-u nepromenljive. Jedna od prednosti nepromenljivosti je jasniji kôd. Omogućeno je slobodno prosleđivanje podatka sa garancijom da neće biti izmenjeni u memoriji.

Lista može biti prazna ili se može sastojati od **glave** i **repa**. Glava je prvi element liste, a rep je ostatak liste. Oni se mogu izdvojiti pomoću funkcija *hd/1* i *tl/1*. Dodeljivanje liste promenljivoj, dohvatanje njene glave i repa prikazano je na listingu 2.6. Izdvajanje glave ili repa prazne liste rezultuje greškom.

```
iex(1)> lista = [1, 2, 3, 4]
[1, 2, 3, 4]
iex(2)>hd(lista)
1
iex(3)>tl(lista)
[2, 3, 4]
```

Listing 2.6: Izdvajanje glave i repa liste

Prilikom kreiranja liste, ukoliko Elixir vidi listu *ASCII* brojeva, ispisaće listu znakova. Liste znakova su uobičajene kada se povezuju sa postojećim Erlang kodom. Primer koda koji ilustruje ovo prikazan je na listingu 2.7.

```
iex(1)>[11, 12, 13]
'\v\f\r'
iex(2)>[104, 101, 108, 108, 111]
'hello'
```

Listing 2.7: Lista vrednosti pod jednostrukim navodnicima

Preuzimanje informacija o tipu neke vrednosti može se izvršiti pomoću funkcije *i/1* i može se videti na listingu 2.8.

```
iex(1)>i 'hello'
Term
  'hello'
Data type
  List
Description
  This is a list of integers that is printed as sequence of
   characters delimited by single quotes because all the integers
   in it represent valid ASCII characters. Conventionally, such
   lists of integers are referred to as "charlists" (more precisely
   , a charlist is a list of Unicode codepoints, and ASCII is a
   subset of Unicode).
Raw representation
```

```
  [104, 101, 108, 108, 111]
Reference modules
  List
Implemented protocols
  Collectable, Enumerable, IEx.Info, Inspect, List.Chars, String.
   Chars
```

Listing 2.8: Preuzimanje informacija o tipu vrednosti

Reprezentacije sa jednostrukim i dvostrukim navodnicima u Elixir-u nisu ekvivalentne i predstavljaju različite tipove. Primer se može videti na listingu 2.9.

```
iex(1)>'zdravo' = = "zdravo"
false
```

Listing 2.9: Dva različita tipa

Torke

Torke se u Elixir-u definišu pomoću vitičastih zagrada {}. Kao i liste, mogu sadržati vrednosti bilo kog tipa. Primer torke sa vrednostima različitih tipova i određivanjem njene dužine prikazan je na listingu 2.10.

```
iex(1)>{:ok, "zdravo", 1}
{:ok, "zdravo", 1}
iex(2)>tuple_size({:ok, "zdravo", 1})
3
```

Listing 2.10: Primer torke i određivanje njene dužine

Torke su strukture fiksne dužine koje bi trebalo da sadrže svega nekoliko elemenata koji su zapisani u memoriji jedan za drugim. To znači da se pristup elementu torke ili određivanje dužine torke izvršava u konstantnom vremenu. Razlika u odnosu na liste je u semantici upotrebe. Liste se koriste kada se manipuliše kolekcijom, dok se torke, zbog brzine pristupa njihovim elementima, uglavnom koriste za smeštanje povratne vrednosti funkcije. Na primer, *File.read/1* je funkcija koja se može koristiti za čitanje sadržaja fajla. Ako putanja do fajla postoji, povratna vrednost funkcije je torka sa prvim elementom koji je atom *:ok* i drugim elementom koji je sadržaj datog fajla. U suprotnom, povratna vrednost funkcije će biti torka gde je prvi element atom *:error*, a drugi element opis greške. Primer upotrebe ove funkcije može se videti na listingu 2.11.

```
iex(1)>File.read("C:\elixir\tekstualni_dokument.txt")
```

```
2 {:ok, "Zdravo, svete!"}
```

Listing 2.11: Primer upotrebe funkcije *File.read/1*

Indeksi torke počinju od nule, a primer izdvajanja elementa sa indeksom 1 može se videti na listingu 2.12.

```
1 iex(1)>torka = {:ok, "zdravo", 1}
2 {:ok, "zdravo", 1}
3 iex(2)>elem(torka, 1)
4 "zdravo"
```

Listing 2.12: Izdvajanje elementa torke sa indeksom 1

Umetanje novog elementa na određeno mesto u torki vrši se pomoću funkcije *put_elem*/3. Ona vraća novu torku, u kojoj je tekući element na zadatoj poziciji zamenjen novim elementom, dok originalna torka ostaje neizmenjena. Primer koda koji ilustruje upotrebu ove funkcije prikazan je na listingu 2.13.

```
1 iex(1)>torka = {:ok, "zdravo", 1}
2 {:ok, "zdravo", 1}
3 iex(2)>put_elem(torka, 1, "svete")
4 {:ok, "svete", 1}
5 iex(3)>torka
6 {:ok, "zdravo", 1}
```

Listing 2.13: Umetanje novog elementa u torku

Kao i liste, torke su takođe nepromenljive. Svaka operacija nad torkom vraća novu torku i nikada ne menja postojeću. Ova operacija, kao i operacija ažuriranja torke je skupa, jer zahteva kreiranje nove torke u memoriji. Ovo se odnosi samo na samu torku, a ne na njen sadržaj. Na primer, prilikom ažuriranja torke, svi unosi se dele između stare i nove torke, osim unosa koji je izmenjen. Drugim rečima, torke i liste u Elixir-u mogu da dele svoj sadržaj, što smanjuje količinu memorije koju jezik treba da zauzme. Ove karakteristike performansi diktiraju upotrebu struktura podataka.

Liste ključnih reči i mape

Elixir podržava asocijativne strukture podataka. Asocijativne strukture podataka su one koje su u stanju da pridruže određenu vrednost ili više vrednosti ključu. Dve glavne strukture među njima su **liste ključnih reči** i **mape**.

Liste ključnih reči

U mnogim funkcionalnim programskim jezicima uobičajeno je da se koristi lista dvočlanih torki za predstavljanje strukture podataka ključ - vrednost. Elixir takođe podržava ovakav način predstavljanja. Dodatno, lista torki gde je prvi element torke atom (tj. ključ) u Elixir-u se naziva **lista ključnih reči**. Obezbeđena je i posebna sintaksa za definisanje takvih lista: $[key : value]$. Primer oba načina definisanja prikazan je na listingu 2.14.

```
iex(1)> lista = [{:a, 1}, {:b, 2}, {:c, 3}]
[a: 1, b: 2, c: 3]
iex(2)> lista = = [a: 1, b: 2, c: 3]
true
```

Listing 2.14: Primer liste ključnih reči

Kako su liste ključnih reči liste, nad njima možemo primenjivati sve operacije dostupne nad listama. Na primer, korišćenjem operatora ++ može se izvršiti dodavanje nove vrednosti listi ključnih reči. Primer koda koji ilustruje ovo dodavanje dat je na listingu 2.15.

```
iex(1)> lista = [{:a, 1}, {:b, 2}, {:c, 3}]
[a: 1, b: 2, c: 3]
iex(2)>lista ++ [d: 4]
[a: 1, b: 2, c: 3, d: 4]
iex(3)>[a: 0] ++ lista
[a: 0, a: 1, b: 2, c: 3]
```

Listing 2.15: Dodavanje nove vrednosti listi ključnih reči

Elementima liste ključnih reči se pristupa na način prikazan na listingu 2.16.

```
iex(1)>lista = [a: 0, a: 1, b: 2, c: 3]
[a: 0, a: 1, b: 2, c: 3]
iex(2)>lista[:a]
0
```

Listing 2.16: Pristup elementu liste ključnih reči

Liste ključnih reči su važne, jer imaju tri posebne karakteristike:

1. Ključevi moraju biti atomi.
2. Ključevi su uređeni, onako kako je navedeno od strane programera.
3. Ključevi se mogu ponavljati.

Elixir obezbeđuje modul koji omogućava manipulisanje listama ključnih reči. Liste ključnih reči su jednostavno liste, i kao takve pružaju iste karakteristike linearnih performansi kao i liste. Što je lista duža, više vremena će biti potrebno za pronalaženje ključa, prebrojavanje elemenata i tako dalje. Iz tog razloga, liste ključnih reči se u Elixir-u koriste uglavnom za prosleđivanje opcionih vrednosti. Za čuvanje mnogo elemenata ili garantovanje pojavljivanja jednog ključa sa maksimalno jednom vrednošću treba koristiti mape.

Mape

Mapa je kolekcija koja sadrži parove ključ : vrednost. Glavne razlike između liste parova ključ-vrednost i mape su u tome što mape ne dozvoljavaju ponavljanje ključeva (jer su to asocijativne strukture podataka) i što ključevi mogu biti proizvoljnog tipa. Mapa je veoma efikasna struktura podataka, naročito kada količina podataka raste. Ukoliko želimo da podaci u kolekciji ostanu baš u onom redosledu u kom smo ih naveli inicijalno, onda je bolje koristiti liste parova ključ : vrednost, jer mape ne prate nikakvo uređenje.

Mapa se definiše pomoću sintakse %{} na način prikazan na listingu 2.17.

```
iex(1)> mapa = %{:a => 1, 2 => :b}
%{:a => 1, 2 => :b}
iex(2)>mapa[:a]
1
iex(3)>mapa[2]
:b
iex(4)>mapa[:c]
nil
```

Listing 2.17: Primer mape i pristupa njenim elementima

Modul **Map** obezbeđuje razne funkcije za manipulaciju mapama, a neke od njih mogu se videti na listingu 2.18.

```
iex(1)>Map.get(%{:a => 1, 2 => :b}, :a)
1
iex(2)>Map.put(%{:a => 1, 2 => :b}, :c, 3)
%{2 => :b, :a => 1, :c => 3}
iex(3)>Map.to_list(%{:a => 1, 2 => :b})
[{2 => :b}, {:a => 1}]
```

Listing 2.18: Neke od funkcija modula Map

Mape imaju sintaksu za ažuriranje vrednosti ključa prikazanu na listingu 2.19

```
iex(1)>mapa = %{:a => 1, 2 => :b}
%{2 => :b, :a => 1}
iex(2)>%{mapa | 2 => "dva"}
%{2 => "dva", :a => 1}
iex(3)>%{mapa | :c => 3}
** (KeyError) key :c not found in: %{2 => :b, :a => 1}
    (stdlib) :maps.update(:c, 3, %{2 => :b, :a => 1})
    (stdlib)  erl_eval.erl:259: anonymous fn/2 in :erl_eval.expr/5
    (stdlib) lists.erl:1263: :lists.foldl/3
```

Listing 2.19: Ažuriranje vrednosti ključa

Prethodno prikazana sintaksa zahteva da dati ključ postoji u mapi i ne može se koristiti za dodavanje novih ključeva. Na primer, korišćenje ove sintakse za ključ :c nije uspelo, jer ključ :c ne postoji u mapi.

Ukoliko su svi ključevi u mapi atomi, onda se radi pogodnosti može koristiti sintaksa ključnih reči data listingom 2.20.

```
iex(1)> mapa = %{a: 1, b: 2}
%{a: 1, b: 2}
```

Listing 2.20: Sintaksa ključnih reči

Još jedno zanimljivo svojstvo mapa je to što obezbeđuju sopstvenu sintaksu za pristup atomskim ključevima. Primer ove sintakse možemo videti na listingu 2.21.

```
iex(1)>mapa = %{:a => 1, 2 => :b}
%{2 => :b, :a => 1}
iex(2)>mapa.a
1
iex(3)>mapa.c
** (KeyError) key :c not found in: %{2 => :b, :a => 1}
```

Listing 2.21: Sintaksa za pristup atomskim ključevima

Programeri koji programiraju u Elixir-u pri radu sa mapama češće koriste *map.field* sintaksu i poklapanje obrazaca nego funkcije iz modula Map, jer dovode do asertivnog stila programiranja.

Često se koriste mape unutar mapa ili čak liste ključnih reči unutar mapa. Elixir obezbeđuje pogodnosti za manipulisanje ugnježdenim strukturama podataka poput *put_in*/2, *update_in*/2 i drugih naredbi koje daju iste pogodnosti koje se mogu pronaći u imperativnim jezicima, a da pritom zadrže svojstvo nepromenljivosti podataka (svojstvo koje će detaljnije biti objašnjeno u poglavlju 2.7).

Na listingu 2.22 je prikazana lista ključnih reči korisnika, gde je svaka vrednost mapa koja sadrži ime, starost i listu programskih jezika koje svaki korisnik voli.

```
iex(1)>korisnici = [dzoni: %{ime: "Dzoni", godine: 27, jezici:
["Erlang", "Ruby", "Elixir"]}, marija: %{ime: "Marija", godine: 29,
jezici: ["Elixir", "F#", "Clojure"]}]
[
  dzoni:%{godine:27,jezici:["Erlang","Ruby","Elixir"],ime:"Dzoni"},
  marija:%{godine:29,jezici:["Elixir","F#","Clojure"],ime:"Marija"}
]
```

Listing 2.22: Struktura koja predstavlja listu korisnika

Pristup Džonijevim godinama mogao bi se izvršiti na način prikazan na listingu 2.23.

```
iex(2)>korisnici[:dzoni].godine
27
```

Listing 2.23: Pristup broju godina korisnika sa imenom Dzoni

Ista sintaksa se može koristiti i za ažuriranje vrednosti i prikazana je na listingu 2.24.

```
iex(3)> korisnici = put_in users[:dzoni].godine, 31
[
  dzoni:%{godine:31,jezici:["Erlang","Ruby","Elixir"],ime:"John"},
  marija:%{godine:29,jezici:["Elixir","F#","Clojure"],ime:"Mary"}
]
```

Listing 2.24: Ažuriranje vrednosti

Makro *update_in*/2 je sličan, ali daje mogućnost prosleđivanja funkcije koja kontroliše kako se vrednost menja. Na primer, uklanjanje programskog jezika Clojure sa Marijinog spiska jezika može se uraditi na način prikazan listingom 2.25.

```
iex(4)> korisnici = update_in korisnici[:marija].jezici,
fn jezici -> List.delete(jezici, "Clojure") end
[
  john:%{godine:31,jezici:["Erlang","Ruby","Elixir"],ime:"Dzoni"},
  marija:%{godine:29,jezici:["Elixir","F#"],ime:"Marija"}
]
```

Listing 2.25: Brisanje jezika iz liste

Funkcija *get_and_update_in* omogućava pristup vrednosti i ažuriranje strukture podataka odjednom, a funkcije *put_in*/3, *update_in*/3 i *get_and_update_in*/3 omogućavaju dinamički pristup strukturama podataka.

2.5 Osnovni operatori

Pored osnovnih artimetičkih operatora $+$, $-$, $*$, $/$, kao i funkcija *div/2* i *rem/2* za celobrojno deljenje i ostatak pri celobrojnom deljenju, Elixir podržava i već pomenute operatore $++$ i $--$ za nadovezivanje i oduzimanje listi, kao i operator $<>$ koji se koristi za nadovezivanje stringova.

Elixir obezbeđuje tri logička operatora: **and**, **or** i **not**. Oni su striktni u smislu da očekuju nešto što ima vrednost *true* ili *false* kao svoj prvi operand. Primer koda koji ilustruje ovu osobinu prikazan je na listingu 2.26.

```
iex(1)true and true
true
iex(2)>false or is_atom(:primer)
true
```

Listing 2.26: Primer upotrebe logičkih operatora

Ukoliko kao prvi operand prosledimo nešto čija vrednost nije logička, rezultat je greška kao na listingu 2.27.

```
iex(1)>1 and true
** (BadBooleanError) expected a boolean on left-side of "and",
 got: 1
```

Listing 2.27: Greška pri upotrebi logičkog operatora

And i *or* su lenji operatori, jer desni operand izračunavaju samo u slučaju da levi nije dovoljan za određivanje rezultata.

Pored ovih logičkih operatora, Elixir takođe obezbeđuje operatore ||, && i ! koji prihvataju argumente bilo kog tipa. Povratna vrednost ovih operatora ne mora biti logička, već može biti i brojevna. Sve vrednosti osim **false** i **nil** će biti procenjene na *true*, što se može videti na primeru prikazanom listingom 2.28.

```
iex(1)>1 || true
1
iex(2)>false || 11
11
iex(3)>nil && 13
nil
iex(4)true && 17
17
iex(5)>!true
false
iex(6)!1
```

```
12 false
13 iex(7)>!nil
14 true
```

Listing 2.28: Operatori koji prihvataju argumente bilo kog tipa

Pravilo je da kada se očekuju logičke vrednosti, treba koristiti operatore *and* i *or*, a ako bilo koji od operanada ima vrednost koja nije logička, onda treba koristi ||, && i !.

Elixir takođe obezbeđuje ==, !=, ===, !==, <=, >=, < i > kao operatore poređenja, pri čemu se operator === od operatora == razlikuje po tome što pored vrednosti poredi i tip.

Moguće je i poređenje tipova među sobom. Razlog zbog kojeg se mogu uporediti različiti tipovi podataka je pragmatizam. Algoritmi sortiranja ne moraju da brinu o različitim tipovima podataka da bi sortirali. Ukupan redosled sortiranja je definisan na način prikazan na listingu 2.29.

```
1 number < atom < reference < function < port < pid < tuple
2 < map < list < bitstring
```

Listing 2.29: Poređenje tipova

2.6 Poklapanje obrazaca

Poklapanje obrazaca je proveravanje da li se u datoj sekvenci tokena može prepoznati neki obrazac. Ovaj koncept će biti jasniji na praktičnom primeru operatora =. U većini programskih jezika, operator = je operator dodele koji levoj strani dodeljuje vrednost izraza na desnoj. U Elixir-u se ovaj operator naziva **operator uparivanja** (engl. *matching*). On se uspešno izvršava, ako pronađe način da izjednači levu stanu (svoj prvi operand) sa desnom (drugi operand).

Na primer, izraz $5 = 2 + 2$ bi rezultirao greškom datom na listingu 2.30.

```
1 iex(1)5 = 2 + 2
2 **(MatchError) no match of right hand side value: 4
```

Listing 2.30: Operator uparivanja

Na osnovu greške se može zaključiti da $2 + 2$ zaista nije 5. U Elixir-u leva strana mora da ima istu vrednost kao i desna strana. Vrednost izraza dat listingom 2.31 nije greška, već uspešno poklapanje obrazaca:

```
iex(1)4 = 2 + 2
4
```

Listing 2.31: Uspešno poklapanje obrazaca

Slično, dva identična stringa sa obe strane znaka jednakosti će dati rezultat prikazan listingom 2.32.

```
iex(1)>"pas" = "pas"
"pas"
```

Listing 2.32: Uspešno poklapanje obrazaca sa stringovima

Poklapanje obrazaca se može prikazati i na primeru sa listama. Neka je data lista osoba koja je prikazana listingom 2.33.

```
iex(1)>lista = ["Milan Stamenkovic","Petar Jovanovic","Milica
Lazarevic","Lena Markovic"]
["Milan Stamenkovic","Petar Jovanovic","Milica Lazarevic","Lena
Markovic"]
```

Listing 2.33: Lista osoba

Neka prve tri osobe treba da budu zapamćene. U te svrhe se može iskoristiti poklapanje obrazaca dato na listingu 2.34.

```
iex(2)>[prvi, drugi, treci | ostali] = lista
["Milan Stamenkovic", "Petar Jovanovic", "Milica Lazarevic",
 "Lena Markovic"]
```

Listing 2.34: Poklapanje obrazaca sa listama

Izvršeno je dodeljivanje prve, druge i treće stavke iz liste promenljivama *prvi*, *drugi* i *treći*. Ostatak liste je dodeljen pomenljivoj *ostali* pomoću **pipe operatora** (|). Vrednost svake od ovih promenljivih može se iščitati na način prikazan na listingu 2.35.

```
iex(3)>prvi
"Milan Stamenkovic"
iex(2)>drugi
"Petar Jovanovic"
iex(3)>treci
"Milica Lazarevic"
iex(4)>ostali
["Lena Markovic"]
```

Listing 2.35: Iščitavanje sadržaja promenljivih

Mape su vrlo korisne kod poklapanja obrazaca. Kada se koristi u poklapanju obrazaca, mapa će se uvek podudarati sa podskupom date vrednosti kao što se može videti na listingu 2.36.

```
iex(1)> %{} = %{:a => 1, 2 => :b}
%{:a => 1, 2 => :b}
iex(2)>%{:a => a} = %{:a => 1, 2 => :b}
%{2 => :b, :a => 1}
iex(3)>a
1
iex(4)>%{:c => c} = %{:a => 1, 2 => :b}
** (MatchError) no match of right hand side value: %{1 => :b, :a =>
    1}
    (stdlib) erl_eval.erl:453: :erl_eval.expr/5
    (iex) lib/iex/evaluator.ex:257: IEx.Evaluator.handle_eval/5
    (iex) lib/iex/evaluator.ex:237: IEx.Evaluator.do_eval/3
    (iex) lib/iex/evaluator.ex:215: IEx.Evaluator.eval/3
    (iex) lib/iex/evaluator.ex:103: IEx.Evaluator.loop/1
    (iex) lib/iex/evaluator.ex:27: IEx.Evaluator.init/4
```

Listing 2.36: Mape pri poklapanju obrazaca

Mapa se podudara sve dok ključevi u obrascu postoje u datoj mapi. Tako, prazna mapa odgovara svim mapama.

Promenljive se mogu koristiti prilikom pristupa, podudaranja i dodavanja ključeva mape, što je dato listingom 2.37.

```
iex(1)>mapa = %{a => :jedan}
%{a => :jedan}
iex(2)>mapa[a]
:jedan
iex(3)>%{^a => :jedan} = %{1 => :jedan, 2 => :dva, 3 => :tri}
%{1 => :jedan, 2 => :dva, 3 => :tri}
```

Listing 2.37: Upotreba promenljivih u mapama

2.7 Nepromenljivost podataka

U mnogim programskim jezicima je dozvoljeno dodeljivanje vrednosti promenljivoj, a zatim njeno menjanje tokom izvršavanja programa. Mogućnost zamene vrednost na određenoj memorijskoj lokaciji drugom vredošću čini se legitimna i čini se da povećava čitljivost našeg programa. Tokom izvršavanja programa obično se

ne zna tačno vreme izvršavanja ove promene i obično se i ne vodi računa o tome pri pisanju programa. Ali šta se dešava kada se vrednost u memoriji, ili čak i tip vrednosti, promeni u trenutku kada je koristi više instanci programa? Ovakvo ponašanje je poznato kao **promenljivost (mutabilnost)**. U konkurentnim okruženjima je izvor grešaka koje je veoma teško pratiti i reprodukovati. Promenljivost takođe vodi veoma komplikovanom kodu, koji se često piše ad-hoc kako bi se rešili problemi sinhronizacije.

Umesto toga, drugi jezici, kao što je Erlang, a samim tim i Elixir imaju osobinu **nepromenljivosti (imutabilnosti)**. Oni jednostavno ne dozvoljavaju promenu vrednosti na određenoj memorijskoj lokaciji. Na ovaj način, ako se promenljiva *a* poklopila sa vrednošću 1, onda se njena vrednost sigurno neće menjati tokom izvršavanja programa i ne mora se voditi računa o problemima sinhronizacije u konkurentnom okruženju.

2.8 Konkurentnost

Jedna od glavnih karakteristika Elixir-a je ideja o organizovanju koda u male komande koje se mogu izvršavati nezavisno i istovremeno. Zahvaljujući arhitekturi virtuelne mašine Erlang-a na kojoj se Elixir pokreće, stvaranje velikog broja procesa ne utiče na smanjivanje performansi. Elixir koristi model konkurentnosti ***actor***. *Actor* je nezavisan proces koji ne deli ništa sa bilo kojim drugim procesom. Mogu se stvarati novi procesi, slati poruke procesima i primati poruke od njih. Procesi u Elixir-u nemaju nikakve veze sa izvornim procesima operativnog sistema koji su spori i glomazni. Elixir koristi podršku za procese iz Erlang-a. Ovi procesi se odvijaju širom svih centralnih procesorskih jedinica, baš kao i izvorni procesi operativnog sistema, ali koriste vrlo malo resursa. Moguće je stvoriti stotine hiljade procesa čak i na skromnom računaru. Na listingu 2.38 se nalazi kôd koji definiše modul *KreiranjeProcesa* i u okviru njega funkciju *pozdrav* koja ispisuje *Zdravo*.

```
1 defmodule KreiranjeProcesa do
2   def pozdravdrav do
3     IO.puts "Zdravo"
4   end
5 end
```

Listing 2.38: Modul *KreiranjeProcesa*

Nakon kompajliranja fajla u kome je definisan modul *KreiranjeProcesa* pomoću komande *iex ime_fajla.ex*, na listingu 2.39 se može videti regularno pokretanje funkcije *pozdrav* i pokretanje u okviru posebnog procesa. Pokretanje u okviru posebnog procesa se vrši pomoću funkcije *spawn*.

```
1 iex> KreiranjeProcesa.pozdrav
2 Zdravo
3 :ok
4 iex> spawn(KreiranjeProcesa, :pozdrav, [])
5 Zdravo
6 #PID<0.42.0>
```

Listing 2.39: Pokretanje funkcije *greet*

Funkcija *spawn* kreira novi proces. Pojavljuje se u mnogim oblicima, ali dva najjednostavnjija su ona koji omogućavaju pokretanje anonimne funkcije i pokretanje imenovane funkcije u modulu, prosleđujući listu argumenata. Drugi oblik je iskorišćen u primeru sa listinga 2.39. Ova funkcija vraća identifikator procesa koji se obično naziva ***PID*** i koji jedinstveno identifikuje proces koji je kreiran. Kada se funkcija *spawn* pozove, kreira se novi proces u okviru koga će se funkcija izvršiti. Ne zna se kada će tačno biti izvršena, ali se zna da je kvalifikovana za izvršavanje [14].

2.9 Odlučivanje

Strukture odlučivanja zahtevaju da programer odredi jedan ili više uslova koje će program proceniti ili testirati zajedno sa naredbom ili naredbama koje treba izvršiti, ako je uslov određen ili tačan, i opciono, druge naredbe koje treba izvršiti, ako je utvrđeno da je uslov netačan.

Elixir obezbeđuje ***if/else*** uslovne konstrukte kao i mnogi drugi programski jezici. On takođe poseduje naredbu ***cond*** koja poziva prvu tačnu vrednost koju pronađe. ***Case*** je još jedan kontrolni tok koji koristi poklapanje obrazaca za kontrolu toka programa.

Elixir ima sledeće vrste naredbi za odlučivanje:

1. *if* - Naredba *if* se sastoji od logičkog izraza praćenog ključnom reči *do*, jedne ili više izvršnih naredbi i na kraju ključne reči *end*
2. *if..else* - Naredba *if* može biti praćena naredbom *else* (unutar *do..end* bloka), koja se izvršava, ako je logički izraz netačan.

3. *unless* - Naredba *unless* ima isto telo kao i naredba *if*. Kôd unutar *unless* naredbe se izvršava samo kada je navedeni uslov netačan.
4. *unless..else* - Naredba *unless...else* ima isto telo kao i naredba *if..else*. Kôd unutar *unless..else* naredbe se izvršava samo kada je navedeni uslov netačan.
5. *cond* - Naredba *cond* se koristi ukoliko treba izvršiti neki kôd na osnovu nekoliko uslova. Radi kao *if..else if..else* kod drugih programskih jezika.
6. *case* - Naredba *case* se može smatrati zamenom za naredbu ***switch*** u imperativnim programskim jezicima. Naredba *case* uzima promenljivu ili literal i primenjuje odgovarajući obrazac poklapanja u različitim slučajevima. Ako se bilo koji slučaj poklapa, Elixir izvršava kôd povezan sa tim slučajem i izlazi iz naredbe *case*.

2.10 Moduli

U Elixir-u se može vršiti grupisanje nekoliko funkcija u module. Već su pomenuti različiti moduli u prethodnim odeljcima (*Map*, *Enum*, *List*, *String*,...). Za kreiranje sopstvenih modula u Elixir-u, koristi se makro ***defmodule***, a za definisanje svojih funkcija, koristimo makro ***def***. Primer koda koji ilustruje kreiranje modula i funkcija dat je listingom 2.40.

```
1 defmodule Matematika do
2   def saberi(a, b) do
3     a + b
4   end
5 end
```

Listing 2.40: Kreiranje modula i funkcija

Moduli mogu biti ugnježdeni u Elixir-u. Ova osobina jezika omogućava bolje organizovanje koda. Na listingu 2.41 se može videti definisanje dva modula: **Matematika** i **Matematika.Sabiranje**, pri čemu je drugi ugnježden unutar prvog. Drugom se može pristupati samo pomoću *Sabiranje* unutar modula *Matematika* sve dok su u istom leksičkom opsegu. Ako se kasnije modul *Sabiranje* premesti izvan definicije modula *Matematika*, onda se mora referencirati njegovim punim imenom *Matematika.Sabiranje* ili pseudonim mora biti postavljen pomoću direktive aliasa.

```
defmodule Matematika do
  defmodule Sabiranje do
    def saberi(a, b) do
      a + b
    end
  end
end
```

Listing 2.41: Ugnježdavanje modula

U Elixir-u nema potrebe za definisanjem modula *Matematika*, kako bi se definisao modul *Matematika.Sabiranje*, pošto jezik prevodi sva imena modula u atome. Mogu se definisati i proizvoljno ugnježdeni moduli bez definisanja bilo kog modula u lancu. Na primer, može se definisati modul *Matematika.Sabiranje.Saberi*, iako prethodno nije definisan modul *Matematika* i *Matematika.Sabiranje*.

2.11 Direktive

Kako bi se olakšala ponovna upotreba koda, Elixir obezbeđuje tri direktive - **alias, require i import**, kao i makro **use**. Primer njihove upotrebe može se videti na listingu 2.42.

```
#Alias modula tako da se moze pozvati sa Sabiranje umesto sa
#Matematika.Sabiranje
alias Matematika.Sabiranje, as: Sabiranje

#Obezbedjuje da je modul kompajliran i dostupan (obicno za makroe)
require Matematika

#Ukljucuje prilagodjen kod definisan u modulu Matematika kao
   prosirenje
use Matematika
```

Listing 2.42: Primer upotrebe direktiva

Direktiva *alias*

Direktiva aliasa služi za podešavanje pseudonima za bilo koje ime modula. Aliasi moraju uvek počinjati velikim slovom. Validni su samo unutar leksičkog opsega u kome su pozvani.

Direktiva *require*

Elixir obezbeđuje makroe kao mehanizam za meta-programiranje (pisanje koda koji generiše kôd). Makroi su delovi koda koji se izvršavaju i proširuju tokom kompilacije. To znači da bi se mogao koristiti makro, mora se garantovati da su njegovi moduli i implementacija dostupni tokom kompilacije. Ovo se čini pomoću **require** direktive. Uopšteno, moduli nisu potrebni pre upotrebe, osim ako želimo da koristimo makroe koji su dostupni u njemu. Require direktiva je takođe leksički određena.

Direktiva *import*

Direktiva **import** se koristi kako bi se lakše pristupalo funkcijama i makroima iz drugih modula bez upotrebe potpuno kvalifikovanog imena. Import direktiva je takođe leksički određena.

Makro *use*

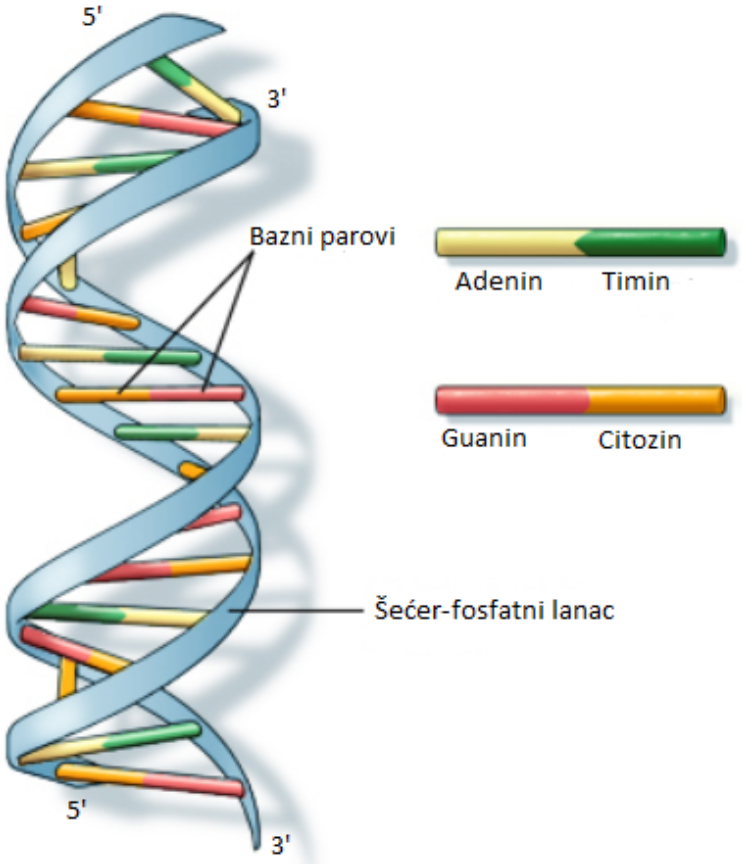
Iako nije direktiva, **use** je makro koji je usko povezan sa zahtevom koji omogućava korišćenje modula u trenutnom kontekstu. Makro use se često koristi za unos spoljne funkcionalnosti u trenutni leksički opseg, često modula.

Glava 3

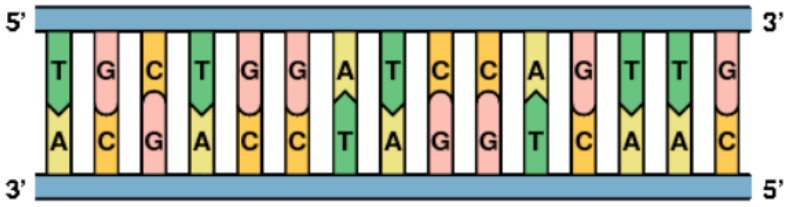
Sekvencioniranje genoma

DNK (dezoksiribonukleinska kiselina) je nukleinska kiselina koja sadrži uputstva za razvoj i pravilno funkcionisanje svih živih organizama. Informacije u DNK se čuvaju kao kôd koji čine **četiri azotne baze**: **adenin** (A), **guanin** (G), **citozin** (C) i **timin** (T). Ljudski DNK se sastoji od oko tri milijarde baza, a više od 99 procenata tih baza je isto kod svih ljudi. Redosled ili sekvenca ovih baza određuje informacije dostupne za izgradnju i održavanje organizma, slično načinu na koji se slova abecede pojavljuju određenim redosledom kako bi se formirale reči i rečenice. DNK baze se spajaju jedna sa drugom, adenin sa timinom i citozin sa guaninom, da bi formirale jedinice koje se nazivaju **bazni parovi**. Svaka baza je takođe vezana za molekul šećera dezoksiriboze i molekul fosfata. Zajedno, baza, šecer i fosfat nazivaju se **nukleotidima**. Nukleotidi su raspoređeni u dva dugačka lanca koji koja čine spiralu koja se naziva **dvostruka spirala** [18]. Struktura dvostruke spirale pomalo je nalik na merdevine, pri čemu bazni parovi formiraju lestvičaste trake, a molekuli šecera i fosfata formiraju vertikalne bočne delove merdevina (slika 3.1). Molekul DNK se može zamisliti odmotan i rotiran, tako da su trake merdevina orijentisane vertikalno, a bazni parovi se mogu čitati sa leva na desno (slika 3.2).

Krajevi šećer-fosfatnih lanaca se međusobno razlikuju po prirodi nevezanog atoma ugljenika - jedan kraj ima nevezani $5'$ (pet prim) atom ugljenika, dok drugi kraj ima nevezani $3'$ (tri prim) atom ugljenika. $5'$ ugljenik ima fosfatnu grupu koja je vezana za njega, a $3'$ ugljenik-hidroksilnu (-OH) grupu. Ova asimetrija daje DNK pravac $5' \rightarrow 3'$. Kada predstavljamo molekul DNK pomoću dijagrama, kakav je prikazan na slici 3.1, podrazumeva se da gornji lanac kreće od $5'$ kraja sa leve strane do $3'$ kraja sa desne strane. Donji lanac je obrnuto orijentisan, od $3'$ kraja sa leve strane do $5'$ kraja sa desne strane [3].



Slika 3.1: Struktura DNK [18]



Slika 3.2: Odmotan DNK [3]

Sva živa bića svoj genetički materijal nose u obliku DNK, sa izuzetkom nekih virusa koji imaju ribonukleinsku kiselinu (RNK). DNK ima veoma važnu ulogu ne samo u prenosu genetskih informacija sa jedne na drugu generaciju, već sadrži i uputstva za građenje neophodnih ćelijskih organela, proteina i RNK molekula. **Geni** su delovi DNK sekvence unutar kojih je kodirano pravilo za sintezu jednog proteina u ćeliji, koji je neophodan za njeno pravilno funkcionisanje. **Genom** je skup gena jednog organizma i sastoji od niza uparenih baza. Sa računarske strane

genom predstavlja nisku nad azbukom $\{A, C, G, T\}$. Nukleotidi su raspoređeni u dva dugačka lanca koji imaju antiparalelnu[1] orijentaciju.

Sekvencioniranje genoma podrazumeva otkrivanje sastava genoma. U pitanju je veoma kompleksan proces koji uključuje i eksperimentalan deo – da bi se saznalo šta se nalazi u sastavu jednog genoma, potreban je uzorak tkiva odgovarajuće vrste [11]. **Sekvencioniranje DNK** je proces određivanja preciznog redosleda nukleotida unutar molekula DNK, a mašine u kojima taj proces započinje se nazivaju **sekvenceri**. Sekvenceri za dati uzorak bilo kog tkiva bilo kog organizma (npr. krvi kod čoveka, dlake kod miša,...) mogu da očitaju podsekvence DNK koje se nazivaju **očitavanja** (engl. *reads*), a koje je nakon toga neophodno sastaviti u polaznu DNK sekvencu pomoću posebnih alata za sklapanje, takozvanih **asemblera**. Podniske očitavanja dužine k nazivaju se **k-meri**. Dužina genoma i dužina očitavanja DNK se mere u **baznim parovima (bp)**. Očitavanja mogu biti **kratka** i **duga**. Kratka očitavanja imaju dužinu od 50 bp do 400 bp, a duga očitavanja dužinu veću od 400 bp.

Rekonstrukcija genoma kroz sekvencioniranje DNK je veoma važan problem. Ona se može uporediti sa kompletiranjem slagalice, gde su očitavanja delovi slagalice. Za razliku od kompletiranja slagalice, gde je poznata slika koju treba na kraju dobiti, pri rekonstrukciji genoma nije pozanto kako treba da izgleda krajnji rezultat. Što su delovi veći, slagalicu je lakše sastaviti. Zadatak asemblera je da na osnovu očitavanja, za koje ne postoji informacija sa kog dela genoma potiču, složi genomsku slagalicu.

3.1 Istorija sekvencioniranja genoma

Sekvencioniranje Sanger predstavlja prvu tehniku sekvencioniranja. Nastala je 1977. godine, a njeni tvorci su Frederik Sanger (engl. *Frederick Sanger*) i njegove kolege. Razvijena su dva asemblera za asembliranje očitavanja sekvencioniranja Sanger: *OLC*[2] asembler **Celera** i asembler **Ojler** zasnovan na De Brojnovim grafovima. **Humani referentni genom** sastavljen je korišćenjem ova dva pristupa. Humani referentni genom predstavlja reprezentativni primer skupa gena čoveka koje

[1] U biohemiji, dva molekula su antiparalelna, ako se prostiru paralelno, ali u suprotnim smerovima.

[2] Skraćenica od *overlap layout consensus* (*overlap* - izgradnja grafa preklapanja, *layout* - spajanje putanja u grafu u kontige, *consensus* - određivanje najverovatnije sekvence nukleotida za svaku kontigu.

su prikupili naučnici. Kako su često sastavljeni sekvencioniranjem DNK većeg broja davalaca, referentni genomi ne predstavljaju skup gena nijedne pojedinačne osobe. Humani referentni genom ***GRCh37*** *(The Genome Reference Consortium human genome (build 37))* je izveden 2009. godine iz DNK trinaest anonimnih dobrovoljaca iz države Bafalo u SAD. Referentni genomi se obično koriste kao uputstvo na osnovu koga se grade novi genomi, što im omogućava da se sastave mnogo brže i ekonomičnije. Međutim, kako je sekvencioniranje Sanger niskopropusno (omogućava sekvencioniranje malog broja očitavanja odjednom) i neekonomično, samo nekoliko genoma je sastavljeno pomoću njega.

Razvoj tehnika za sekvencioniranje **druge generacije** značajno je doprineo efikasnijem i ekonomičnijem sekvencioniraranju stotine miliona očitavanja. Međutim, očitavanja druge generacije sekvencioniranja su kratka. Njihova pojava je dovela do većeg broja uspešnih ***de novo*** asemblerskih[3] projekata, uključujući rekonstrukciju genoma **Džejmsa Votsona** (engl. *James Watson*) i genoma **panda**. Iako je ovaj pristup ekonomičan, rezultat su bili fragmentisani genomi, jer su očitavanja kratka i ponavljajući regioni, takozvani **ponovci**, dugi. Ponovci su fenomen kada se šablon od nekoliko nukleotida pojavljuje više puta u nizu.

Od nedavno su na raspolaganju tehnike za sekvencioniranje **treće generacije**, koje proizvode duga očitavanja (dužine od oko 10000 bp). Duga očitavanja mogu obuhvatiti složene genomske karakteristike, omogućavajući njihovo ispravnije postavljanje u rekonstruisanom genomu, tj. mogu rešiti problem ponavljajućih regiona. Međutim, duga očitavanja imaju visoku stopu greške ($15\% - 18\%$). U cilju rešavanja ovog problema razvijen je veliki broj računskih metoda za korekciju grešaka u očitavanjima treće generacije sekvencioniranja.

3.2 Sekvencioniranje *shotgun* celokupnog genoma

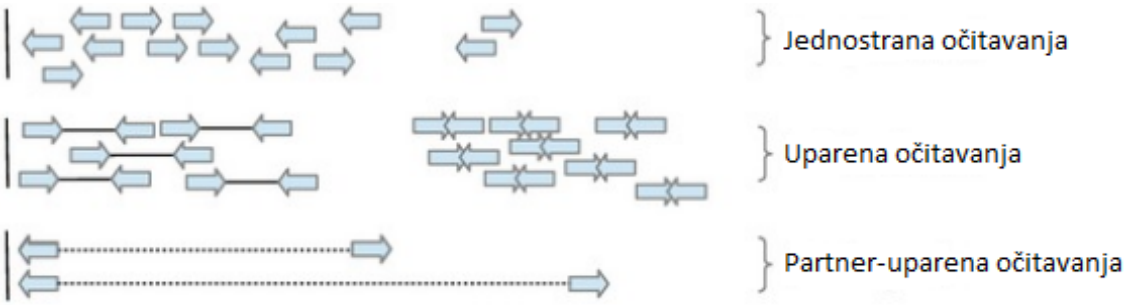
Jedna od tehnika druge generacije sekvencioniranja je sekvencioniranje *shotgun* celokupnog genoma. Prvi korak u ovom procesu je razbijanje genoma na skup očitavanja, a na osnovu njegovog uzorka. Postoje tri vrste očitavanja:

- jednostrana očitavanja (engl. *single-end reads*)
- uparena očitavanja (engl. *paired-end reads*)

[3] *De novo* asembleri su programi koji vrše sklapanje tako što proširuju kratka očitavanja spajanjem susednih očitavanja u dužu sekvencu, bez korišćenja referentne sekvence.

- partner-uparena očitavanja (engl. *mate-pair reads*)

Jednostrana očitavanja su očitavanja koja se generišu tako što sekvencer čita DNK fragment samo u jednom smeru, ili sa $5'$, ili sa $3'$ kraja. U slučaju uparenih očitavanja, sekvencer čita DNK fragment sa oba kraja. Partner-uparena očitavanja su $5'$ ili $3'$ očitavanja DNK fragmenta oba lanca. Prikaz ovih vrsta očitavanja može se videti na slici 3.3. Strelice označavaju smer u kome je sekvencer vršio očitavanje DNK fragmenta.



Slika 3.3: Vrste očitavanja [6]

Sekvenceri čitaju deo DNK fragmenta neke unapred zadate dužine koja se naziva **dužina očitavanja**. **Dubina pokrivanja** neke pozicije u DNK sekvenci je velika, ako je nukleotid na toj poziciji pročitan veliki broj puta u jedinstvenim očitavanjima. Tačnost sekvencioniranja za svaki pojedinačni nukleotid je veoma visoka, ali ukoliko se pojedinačni genom sekvencira samo jednom, zbog veoma velikog broja nukleotida u genomu, doći će do značajnog broja grešaka u sekvencioniranju. Pored toga, neke pozicije u genomu sadrže retke **jednonukleotidne polimorfizme**[4] (engl. *single-nucleotide polymorphisms - SNPs*). Stoga, kako bi se napravila razlika između grešaka u sekvencioniranju i pravih *SNP*-ova, potrebno je još više povećati tačnost sekvencioniranjem pojedinačnih genoma veći broj puta [10].

Za potrebe sekvencioniranja *shotgun* celokupnog genoma postoje dva protokola:

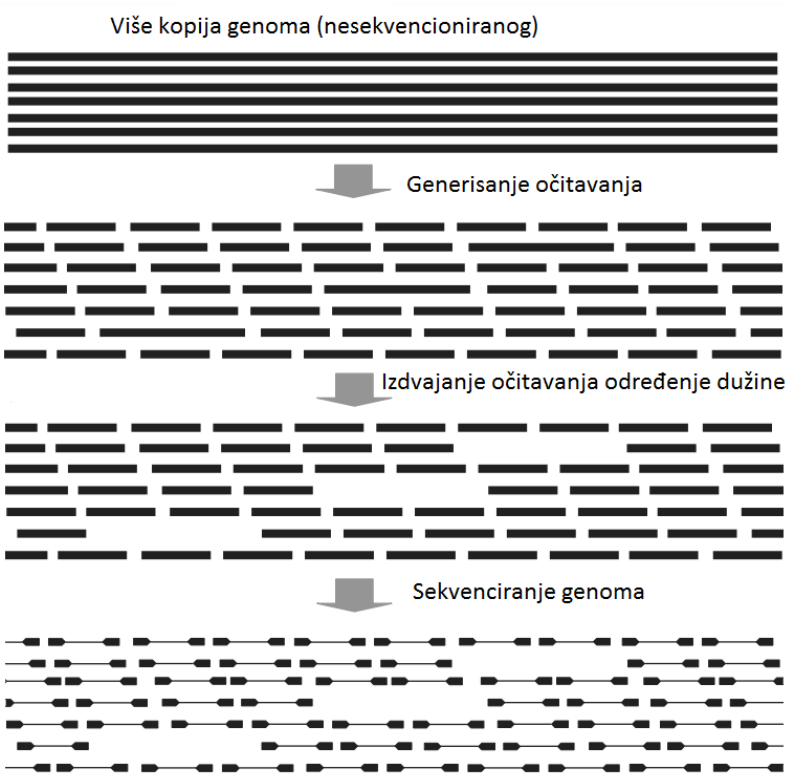
- sekvencioniranje celokupnog genoma
- sekvencioniranje partner-uparenih očitavanja

U okviru oba protokola prvo se uzorak genoma na slučajan način razbija na DNK fragmente, a potom se vrši odabir dužine očitavanja i izdvajanje fragmenata te dužine.

[4]Pojava zamene mesta jednog nukleotida nekim drugim nukleotidom.

Sekvencioniranje celokupnog genoma

Nakon izdvajanja fragmenata DNK određene dužine, vrši se sekvencioniranje jednostranih očitavanja ili sekvencioniranje uparenih očitavanja. Pri sekvencioniranju jednostranih očitavanja, sekvencer čita DNK fragment u jednom smeru, dok u slučaju uparenih očitavanja, fragment biva pročitan u oba smera. Pomenuti koraci su prikazani na slici 3.4.

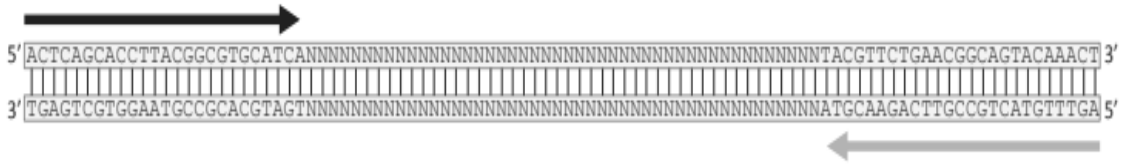


Slika 3.4: Sekvencioniranje *shotgun* celokupnog genoma [12]

Na slici 3.5 prikazan je odmotan i rotiran DNK fragment. Delovi koji se nalaze na poziciji koja premašuje dužinu očitavanja i ne ulaze u sastav očitavanja označeni su sa N.

Prilikom sekvencioniranja jednostranih očitavanja, dobija se očitavanje sa $5'$ kraja gornjeg lanca, tj. $ACTCAGCACCTTACGGCGTGCATCA$. Prilikom sekvencioniranja uparenih očitavanja, dobijaju se $5'$ očitavanja i gornjeg i donjeg lanca:

- $ACTCAGCACCTTACGGCGTGCATCA$
- $AGTTTGTACTGCCGTTCAGAACGTA$



Slika 3.5: DNK fragment ekstrahovan iz uzorka genoma [12]

Drugo očitavanje $AGTTTGTACTGCCGTTCAGAACGTA$ je obrnuti komplement očitavanja $TACGTTCTGAACGGCAGTACAAACT$, jer je drugo očitavanje dobijeno iz donjeg lanca koji je komplementaran sa gornjim.

Za asembliranje genoma, dužina očitavanja je veoma važan parametar. Različite tehnologije sekvencioniranja imaju različita ograničenja na dužinu očitavanja. Na primer, *Illumina Hi-seq*[5] sekvencioniranje može izvršiti samo sekvencioniranje uparenih očitavanja dužine manje od 1000 bp. Za treću generaciju sekvencioniranja, ograničenje dužine očitavanja može biti veće od 10000 bp.

Sekvencioniranje partner-uparenih očitavanja

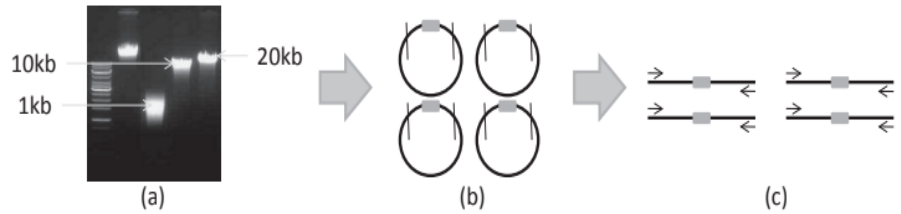
Sekvenceri druge generacije mogu izdvojiti partner-uparena očitavanja sa oba kraja kratkih fragmenata DNK (dužine očitavanja manje od 1000 bp). Za izdvajanje ovakvih očitavanja može se koristiti **sekvencioniranje partner-uparenih očitavanja**.

Sekvenceri prvo izdvajaju duge DNK fragmente neke fiksirane dužine očitavanja (npr. 10000 bp). Potom dodaju takozvane **adapter sekvence**[6] na krajeve svakog fragmenta (slika 3.6(a)). Zatim se vrši sečenje fragmenta levo i desno od adapter sekvence (slika 3.6(b)). Na kraju se uparena očitavanja ekstrahuju sekvencioniranjem uparenih očitavanja (slika 3.6(c)).

Orijentacije uparenih očitavanja očitanih od strane sekvencera partner-uparenog i uparenog očitavanja se razlikuju. Sekvenceri partner-uparenih očitavanja daju dva očitavanja sa oba kraja svakog fragmenta DNK u spoljašnjoj orijentaciji umesto u unutrašnjoj. Npr. za DNK fragment sa slike 3.5 sekvencioniranje partner-uparenih očitavanja će dati:

[5]Illumina je američka kompanija koja je proizvodila visokopropusne sekvencere (omogućavaju sekvencioniranje velikog broja očitavanja odjednom (engl. *high-throughput sequencers*)).

[6]Adapter sekvence su kratke hemijski sintetizovane sekvence nukleotida koje se mogu vezati za krajeve nepoznatih DNK sekvenci i neophodne su u nekim koracima sekvenciranja.



Slika 3.6: Sekvencioniranje partner-uparenih očitavanja [12]

- TGATGCACGCCGTAAGGTGCTGAGT
- TACGTTCTGAACGGCAGTACAAACT

Iako protokol za sekvencioniranje partner-uparenih očitavanja može izdvojiti uparena očitavanja velike dužine očitavanja, on zahteva veći broj ulaznih kopija genoma za pripremu sekvencerskih biblioteka i sklon je greškama pri preklapanju očitavanja (engl. *ligation errors*).

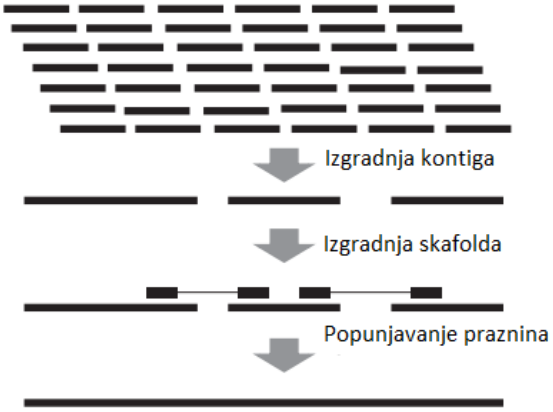
3.3 Asembliranje *de novo* genoma za kratka očitavanja

Druga generacija tehnika sekvencioniranja omogućava dobijanje skupa jednostranih ili uparenih kratkih očitavanja celokupnog genoma. Asembliranje *de novo* ima za cilj da izvrši preklapanje očitavanja u ispravnom redosledu i rekonstruiše genom.

Problem asembliranja genoma je računski težak. Čak i kada ne postoji greška sekvencioniranja, ovaj problem je ekvivalentan **problemu superstringa** za koji se zna da je NP-kompletan [8]. Problem superstringa predstavlja problem pronalaženja superstringa na osnovu skupa stringova S, gde je superstring najkraći string P takav da je svaki string s iz skupa S podstring stringa P. Na primer, ako je $S = \{ACATGC, ATGCGTGT, GTGTACGT\}$, onda je odgovarajući superstring $ACATGCGTGTACGT$.

Postoji veliki broj *de novo* asemblera za asembliranje kratkih očitavanja. Opšte rešenje uključuje četiri koraka. U prvom koraku se koriguju greške u očitavanjima (opisano u poglavlju 3.4). Na osnovu korigovanih očitavanja, u drugom koraku se vrši spajanje očitavanja preklapanjem (opisano u poglavlju 3.5). U idealnom slučaju, teži se spajanju svih očitavanja tako da se formira kompletan genom, ali kako se zbog ponovaka javljaju dvosmislenosti, to nije moguće. Postojeće metode,

radi razrešavanja dvosmislenosti, preklapanjem očitavanja prvo grade kontinuirane sekvence, takozvane **kontige** (engl. *contigs*). Kontige obično predstavljaju jednu **konsenzus nisku**[7]. Algoritmi za konstrukciju kontiga biće obrađeni u poglavlju 3.5. Zatim, koristeći uparena očitavanja, vrši se rekonstrukcija redosleda kontiga tako da se formiraju **skafoldi** (engl. *scaffolds*). Svaki skafold je niz kontiga, a još se naziva i **superkontig** ili **metakontig**. Na kraju se vrši preuređivanje očitavanja u skafoldima kako bi se popunile praznine između susednih kontiga. Opisani koraci su prikazani na slici 3.7.



Slika 3.7: Sekvencioniranje *de novo* genoma za kratka očitavanja [12]

3.4 Korekcija grešaka

Ukoliko se neki k-mer pojavljuje u ulaznim očitavanjima jednom ili veoma mali broj puta, velika je verovatnoća da je on nastao kao posledica grešaka prilikom očitavanja i da ne treba da se nađe u rekonstruisanom genomu. Sa druge strane, za k-mer koji se pojavljuje veliki broj puta u ulaznim očitavanjima sa velikom sigurnošću se može tvrditi da nije posledica grešaka pri očitavanju. Ove razlike se mogu iskoristiti za filtriranje grešaka u k-merima i selektivno uklanjanje iz skupa podataka očitavanja koja sadrže greške [12]. Greške u očitavanjima mogu zbuniti *de novo* asemblere. Da bi se to izbeglo, vrši se korigovanje tih grešaka pre početka asembliranja genoma.

[7]Niska sastavljena od najfrekventnijih nukleotida na pozicijama poravnatih sekvenci.

Brojanje k-mera

Jedan konceptualno jednostavan, ali osnovni problem je **brojanje k-mera**. Brojanje k-mera se prvenstveno koristi u korekciji grešaka u očitavanjima, ali može biti korišćeno i u koraku asembliranja, detekciji ponovaka i kompresiji genomskih podataka. Ima značajnu ulogu u utvrđivanju da li je došlo do grešaka u očitavanju ili je u pitanju jednonukleotidni polimorfizam.

Problem brojanja k-mera se definiše nad ulaznim skupom očitavanja i parametrom k. Iz ovog skupa, potrebno je izdvojiti podskup Z svih mogućih k-mera i odrediti njihove frekvencije. Iako jednostavan problem, zbog velike količine podataka, on se ne može rešavati naivnim pristupima, već su potrebni napredni algoritmi i strukture podataka kako bi problem bio efikasno rešen. U nastavku će biti razmatrana četiri rešenja:

- Algoritam jednostavno heširanje
- Algoritam JellyFish
- Algoritam BFCounter
- Algoritam DSK

Jednostavno heširanje - Problem brojanja k-mera može biti rešen implementacijom asocijativnog niza koristeći **heširanje**. Heširanje je tehnika kojom se vrši preslikavanje skupa ključeva na tabelu značajno manjih dimenzija. Idealno bi bilo da funkcija za svaki ključ daje jedinstvenu poziciju. Ta funkcija naziva se **heš funkcija**, a tabela koja se koristi u tom postupku zove se **heš tabela**. Kada je k malo (npr. manje od 10), u procesu brojanja k-mera koristi se **savršeno heširanje**. Savršeno heširanje garantuje da neće doći do **kolizije**[8]. To je moguće kada se tačno zna koji skup ključeva će biti heširan prilikom dizajniranja heš funkcije. Ukoliko je k veliko, algoritam jednostavnog heširanja će zahtevati previše prostora.

Algoritam JellyFish - Moguće je smanjiti prostornu složenost algoritma jednostavnog heširanja koristeći mehanizam **otvorenog adresiranja**. Otvoreno adresiranje je način rešavanja kolizije u heš tabelama. Kada se desi kolizija, traži se sledeća slobodna lokacija u heš tabeli za smeštanje vrednosti. Postoje tri metode otvorenog adresiranja: **linearno popunjavanje**, **kvadratno popunjavanje** i **du-**

[8]Izraz koji potiče od latinske reči *collisio* i znači sudar, sukob. U ovom kontekstu, moguće je da heš funkcija za dva različita k-mera da istu vrednost, te kažemo da su ta dva k-mera u koliziji, tj. sukobu.

plo heširanje. Mehanizam otvorenog adresiranja se koristi u algoritmu *JellyFish* čija će implementacija detaljnije biti objašnjena u poglavlju 4.1.

Algoritam BFCounter - U mnogim aplikacijama, od značaja su samo k-meri koji se pojavljuju najmanje q puta. Kada bi moglo da se izbegne čuvanje k-mera koji se pojavljuju manje od q puta, sačuvalo bi se mnogo memorije. Pol Melsted (engl. *Páll Melsted*) je predložio algoritam **BFCounter** koji broji samo k-mere koji se pojavljuju najmanje q puta. On koristi *counting Bloom* filter[9] da odredi da li se k-mer pojavljuje najmanje q puta [12].

Algoritam DSK - Iako je $BFCounter$ prostorno efikasan, njegova prostorna složenost i dalje zavisi od broja k-mera u skupu očitavanja. Gijom Rizk (u originalu *Guillaume Rizk*) predlaže algoritam koji se naziva **DSK**. Ideja ovog algoritma je da se skup k-mera podeli u različite liste tako da svaka lista bude smeštena na disk koristeći D bitova. Zatim, za svaku listu, k-meri iz liste se dalje dele u podliste tako da svaka podlista može biti sačuvana u memoriji koristeći M bitova. Na kraju, frekvencije k-mera u svakoj podlisti se izračunavaju algoritmom $JellyFish$. Algoritam DSK će biti detaljno objašnjen u poglavlju 4.2.

3.5 Konstrukcija kontiga

Nakon što su sva očitavanja korigovana, može se vršiti njihovo spajanje radi formiranja kontiga. Postoje dva pristupa za konstrukciju kontiga:

- pristup baznog proširenja (*base-by-base* pristup)
- De Brojnov grafovski pristup

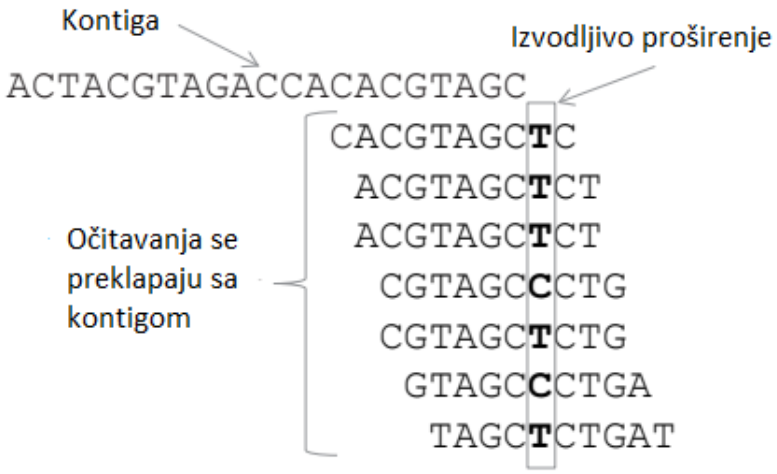
Pristup baznog proširenja

Pristup baznog proširenja rekonstruiše svaku kontigu tako što je proširuje bazu po bazu. Metod počinje tako što se nasumično bira očitavanje koje će služiti kao šablon. Zatim se očitavanja poravnavaju na oba kraja šablona ($5'$ kraj i $3'$ kraj). Na osnovu poravnanja se dobija **nukleotidna baza**[10] i šablon se njome proširuje.

Na slici 3.8 je prikazan jedan korak baznog proširenja. Na vrhu je data sekvenca koja predstavlja šablon, a ispod nje sedam očitavanja poravnatih na $3'$ kraju šablona.

[9] Prostorno efikasna probabilistička struktura podataka koja dozvoljava dodavanje bilo kojih k-mera u nju i ispitivanje da li se k-mer pojavljuje najmanje q puta.

[10] Baza koja se pojavljuje najveći broj puta na određenoj poziciji u poravnanju.



Slika 3.8: Jedan korak baznog proširenja [12]

Pravougaonik pokazuje da su baze C i T izvodljivo proširenje šablona. Kako je T konsenzusna baza, metod baznog proširenja će proširiti kontigu bazom T. Bazno proširenje se ponavlja sve dok ima konsenzusa. Zatim se prestaje sa proširenjem i dobija se kontiga. Proširenje se izvodi i na $3'$ kraju i na $5'$ kraju šablona.

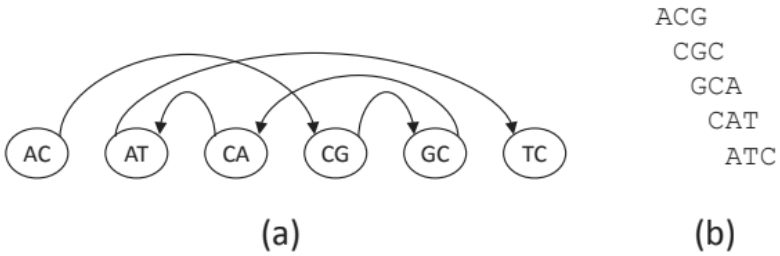
Iako je bazno proširenje jednostavno, ono često daje kratke kontige zbog dva problema. Prvo, početni šablon je nasumično izabrano očitavanje. Ukoliko ono sadrži greške sekvencioniranja ili se nalazi u ponovku, to će uticati na proširenje. Drugi problem je što se može desiti da se šablon proširi u neki od ponovaka. Ponovak stvara grane koje gore navedeni pristup ne može da razreši.

De Brojnov grafovski pristup

De Brojnov grafovski pristup je zasnovan na De Brojnovim grafovima. Ovaj pristup su uveli Ramana Idjuri (engl. *Ramana Idury*) i Majkl Voterman (engl. *Michael Waterman*) u njihovoj knjizi [7]. Ovaj pristup je danas glavni za asembliranje kratkih očitavanja.

Neka je dat skup očitavanja R i parametar k. De Brojnov graf je graf $H_k = (V, E)$, gde je V skup svih k-mera skupa R, a E skup svih grana u grafu. Ako su u i v prefiks dužine k i sufiks dužine k neke podniske dužine $k+1$ iz R, respektivno, k-meri u i v formiraju granu $(u, v) \in E$. Pod pretpostavkom da je N ukupna dužina svih očitavanja iz R, De Brojnov graf može biti konstruisan u $O(N)$ vremenu [12]. Algoritam za konstrukciju De Brojnovog grafa na osnovu očitavanja i parametra k je implementiran u okviru ovog rada i biće detaljno opisan u poglavlju 4.3.

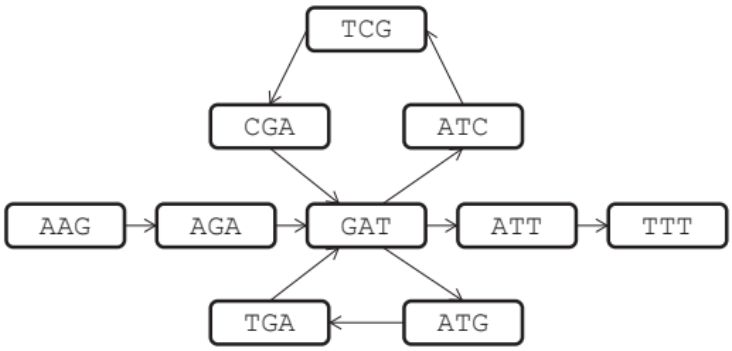
Na slici 3.9(a) je prikazan De Brojnov graf H_2 izgrađen na osnovu skupa k-mera $R = \{ACGC, CATC, GCA\}$, pri čemu je $\{AC, AT, CA, CG, GC, TC\}$ skup čvorova. Preklapanje k-mera koji odgovaraju granama grafa H_2 je prikazano na slici 3.9(b).



Slika 3.9: Konstrukcija De Brojnovog grafa [12]

Genomska sekvenca se može rekonstruisati identifikovanjem **Ojlerove putanje** grafa H_k. Ojlerova putanja je putanja koja obilazi svaku granu grafa H_k tačno jednom. Ojlerova putanja grafa H_k može biti izračunata u $O(n)$ vremenu, ako H_k ima n grana. Na primer, na slici 3.9(a) postoji jedinstvena putanja od čvora AC do čvora TC. Preklapanjem svih ivica 3-mera u redosledu putanje (slika 3.9(b)) dobija se sekvenca $ACGCATC$. Sekvenca $ACGCATC$ je zapravo superstring formiran preklapanjem očitavanja i dobijen na osnovu De Brojnovog grafa za skup R. Međutim, Ojlerova putanja ne mora biti jedinstvena u H_k.

Neka je skup očitavanja $R = \{AAGATC, GATCGAT, CGATGA, ATGATT, GATTT\}$ i neka je $k = 3$. U De Brojnovom grafu H_3 za dati skup R postoje dva ciklusa, te će postojati i dve Ojlerove putanje. Na slici 3.10 je prikazan graf H_3.



Slika 3.10: De Brojnov graf sa dva ciklusa [12]

Ukoliko se prvo obiđe gornji ciklus, dobija se $AAGATCGATGATTT$, a ukoliko se prvo obiđe donji ciklus, dobija se $AAGATGATCGATTT$. Primer sa slike 3.10

pokazuje da Ojlerova putanja možda neće uvek dati ispravnu sekvencu ili možda neće uvek postojati u nekom grafu H_k.

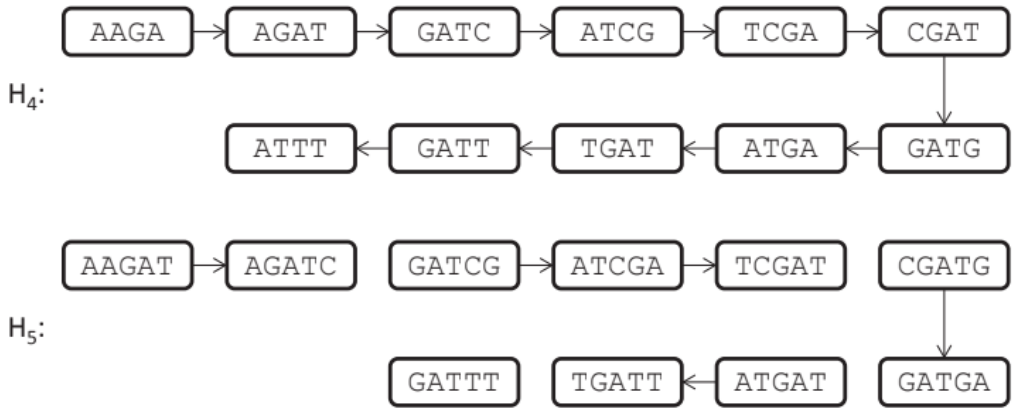
U nastavku će se govoriti o De Brojnovom grafovskom asembleru u slučaju kada:

- ne postoji greška sekvencioniranja
- postoji greška sekvencioniranja

De Brojnov asembler bez greške sekvencioniranja

Kako Ojlerova putanja nije jedinstvena i možda i ne postoji, ne teži se dobijanju kompletnog genoma. Umesto toga, teži se dobijanju skupa kontiga. U grafu H_k, kontiga predstavlja najdužu putanju čiji su svi čvorovi (osim početnog i krajnjeg) unutrašnjeg i spoljašnjeg stepena 1. Takva putanja se naziva maksimalnom prostom putanjom.

Za De Brojnov graf H_3 sa slike 3.10 mogu se konstruisati četiri kontige: $AAGAT$, $GATCGAT$, $GATGAT$, $GATTT$. Parametar k je veoma važan. Za različito k, mogu se dobiti različiti skupovi kontiga. Slika 3.11 ilustruje ovaj problem na De Brojnovom grafu za skup $R = \{AAGATC, GATCGAT, CGATGA, ATGATT, GATTT\}$ u slučajevima kada je $k = 4$ i $k = 5$.



Slika 3.11: De Brojnovi grafovi za $k = 4$ i $k = 5$ [12]

Kada je $k = 4$, H_4 je prosta putanja i može se konstruisati jedna kontiga $AAGATCGATGATTT$. U slučaju da je $k = 5$, H_5 sadrži pet povezanih komponenti, svaka predstavlja prostu putanju i na osnovu toga može se konstruisati pet kontiga: $AAGATC$, $GATCGAT$, $CGATGA$, $ATGATT$, $GATTT$.

Prethodna razmatranja pokazuju da je izbor parametra k veoma vazan. Kada je k malo (pogledati H_3 na slici 3.10), postoji veliki broj grana zbog ponovaka i rezultat je veliki broj kratkih kontiga. Kada je k veliko (pogledati H_5 na slici 3.11), neki k-meri nedostaju, što rezultuje nepovezanim komponentama, koje takođe generišu veliki broj kratkih kontiga. Zato je potrebno identifikovati adekvatno k kako bi se pronašla ravnoteža između ova dva problema. Identifikacija adekvatnog k biće opisana u nastavku teksta.

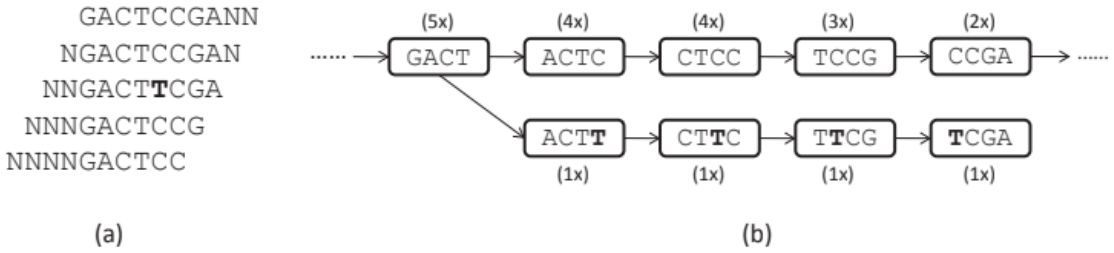
De Brojnov asembler sa greškama sekvencioniranja

Kod De Brojnovog asemblera bez grešaka sekvencioniranja pretpostavka je bila da ne postoje greške sekvencioniranja u očitavanjima, što je nerealno. Kada postoje greške, pokušava se njihovo uklanjanje redukovanjem šuma u De Brojnovom grafu. Rešenje koje će se razmatrati su predložili Danijel Zerbino (engl. *Daniel Zerbino*) i Ivan Birni (enlg. *Ewan Birney*). Kratka očitavanja imaju nisku stopu greške (sadrže jednu grešku na svakih 100 baza), a većina k-mera sadrži najviše jednu grešku. Ovakvi k-meri sa greškom mogu kreirati dve vrste anomalija u De Brojnovom grafu: **vrh (špic)** i **mehurić**.

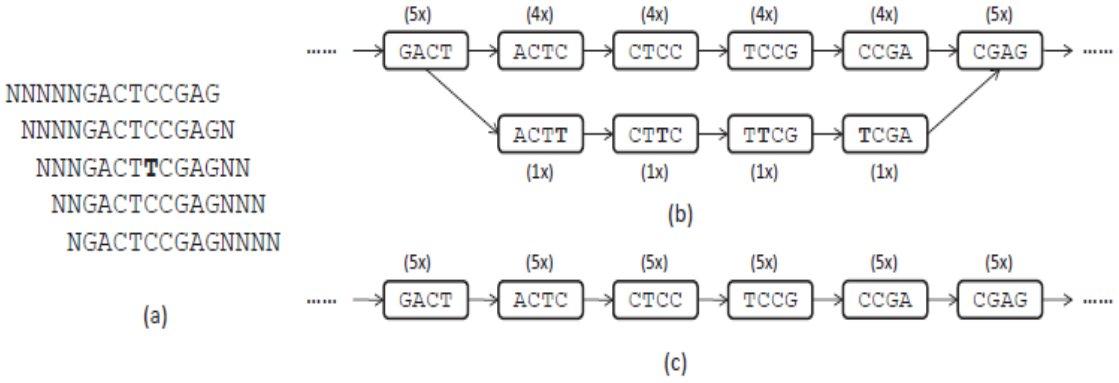
Vrh je putanja dužine najviše k gde svi unutrašnji čvorovi imaju ulazni i izlazni stepen 1, dok jedan od njihovih krajeva ima ulazni ili izlazni stepen 0. To može proizvesti potencijalnu kontigu dužine najviše $2k$.

Slika 3.12(a) predstavlja višestruko poravnanje skupa od pet očitavanja, gde treće očitavanje ima grešku sekvencioniranja (prikazana podebljanim fontom). Slika 3.12(b) predstavlja De Brojnov graf koji odgovara skupu očitavanja sa prvog dela slike i vrh koji je formiran usled jednog nepoklapanja u jednom očitavanju, tj. usled pomenute greške. Brojevi u zagradama označavaju mnogostrukost 4-mera (broj pojavljivanja 4-mera u očitavanjima). Ovaj vrh se može ukloniti iz De Brojnovog grafa, ali uklanjanje jednog vrha može generisati nove vrhove. Zbog toga, procedura uklanjanja mora uklanjati vrhove rekurzivno.

Mehurić se sastoji od dve različite kratke putanje sa istim početnim i krajnim čvorovima u De Brojnovom grafu, pri čemu su te dve putanje zapravo kontige koje se razlikuju u samo jednom nukleotidu. Na primer, slika 3.13(a) sadrži skup očitavanja gde treće očitavanje ima jedno nepoklapanje. Slika 3.13(b) prikazuje odgovarajući De Brojnov graf i jedan mehurić koji se formira.
Gornja putanja mehurića predstavlja $GACTCCGAG$, dok donja putanja predstavlja $GACTTCGAG$. Kada su dve putanje u mehuriću veoma slične, putanja sa



Slika 3.12: Formiranje vrha [12]



Slika 3.13: Formiranje mehurića [12]

nižom mnogostukošću će verovatno biti ona koja će biti odbačena. Može se pokušati sa spajanjem mehurića. U ovom primeru, dve putanje imaju samo jedno nepoklapanje. Budući da su čvorovi u donjoj putanji niže mnogostrukosti, vrši se spajanje mehurića i dobija se graf na slici 3.12(c). Preciznije, može se definisati **težina putanje** $w_1 \rightarrow w_2 \rightarrow ... \rightarrow w_p$ kao $\sum_{i=1}^{p} f(w_i)$, gde je $f(w_i)$ mnogostrukost od w_i. Prilikom spajanja dve putanje koje pripadaju istom mehuriću, zadržava se ona sa većom težinom.

Nakon što se uklone vrhovi i spoje mehurići u De Brojnovom grafu, može se dalje redukovati šum u grafu uklanjanjem k-mera sa mnogostrukošću manjom ili jednakom nekom pragu.

Kombinovanjem ove tri tehnike: (1) otklanjanje vrhova, (2) spajanje mehurića i (3) filtriranje k-mera sa niskom mnogostrukošću dobija se algoritam *Velvet* [19] za manipulaciju De Brojnovim grafovima i sastavljanje genomske sekvence.

Kako izabrati k?

Odabir parametra k može značajno uticati na performanse De Brojnovog algoritma. Pokretanjem algoritma *Velvet* za više različitih parametara k, može se izbeći odabir jedinstvenog k. Nakon izvršavanja ovog algoritma vrši se grupisanje i spajanje dobijenih kontiga. Jedan od problema je što kontige dobijene za različito k imaju različit kvalitet. Kontige dobijene na osnovu H_k, gde je k malo, veoma su tačne. Međutim, takve kontige su kratke, jer postoji veliki broj grana zbog ponovaka. Kontige dobijene na osnovu H_k gde je k veliko su duže, ali one mogu sadržati mnogo grešaka.

IDBA asembler[11] počiva na ideji da ne treba graditi De Brojnov graf nezavisno za različite k. Umesto toga, *IDBA* gradi De Brojnov graf H_k postepeno krećući od malih k i idući ka većim. Kada je k malo, mogu se dobiti visokokvalitetne kontige, iako su kratke. Zatim se ove kontige koriste za ispravljanje grešaka u očitavanjima. Postepeno se izgrađuje De Brojnov graf H_k za sve veće k. Kako su očitavanja u R korigovana, to je šum u grafu H_k redukovan. Na ovaj način se za veliko k omogućava dobijanje dugih kontiga visokog kvaliteta.

[11] IDBA - *A Practical Iterative de Bruijn Graph De Novo Assembler.*

Glava 4

Implementirani algoritmi i rezultati

U glavi 3 data je biološka osnova i motivacija za algoritme koji se primenjuju u sekvencioniranju genoma. U ovom poglavlju će biti dati opisi implementacije tih algoritama u programskom jeziku Elixir sa osvrtom na njihovu prostornu i vremensku složenost, kao i opisi aplikacije i rezultati izvršavanja algoritama u okviru nje.

4.1 Algoritam *JellyFish*

Algoritam $JellyFish$ je algoritam koji se koristi za brojanje k-mera, a kao potprogram programa za korigovanje grešaka u očitavanjima. Pseudokod ovog algoritma dat je na slici 4.1.

Neka je h heš funkcija, Z niz k-mera i $H[0..\frac{N}{\alpha} - 1]$ heš tabela koja čuva niz Z, gde je $N = |Z|$ i α faktor opterećenja ($0 < \alpha \leq 1$). Potrebno je izgraditi tabelu $Count[0..\frac{N}{\alpha} - 1]$, gde $Count[i]$ čuva broj pojavljivanja za k-mer $H[i]$. Za svaki k-mer z iz Z vrši se njegovo heširanje u neku vrednost $H[i]$, gde se vrednost indeksa i izračunava pomoću heš funkcije h sa parametrom z. Ako je vrednost $H[i]$ postvaljena i različita od z, z se ne može čuvati u $H[i]$, odnosno došlo je do kolizije. Ona može biti razrešena pomoću mehanizma otvorenog adresiranja. Na primer, kolizija se može razrešiti linearnim popunjavanjem. Kada se kolizija dogodi, indeks i se uvećava za jedan sve dok se ne pronađe prvo prazno mesto $H[i]$ ili mesto gde $H[i]$ ima željenu vrednost z.

Funkcija $hashEntry()$ sa slike 4.2 ilustruje šemu linearnog popunjavanja za razrešavanje kolizije. Ako $hashEntry(H, z, h, \frac{N}{\alpha})$ vraća indeks i za koji vrednost $H[i]$ nije postavljena, onda z ne postoji u heš tabeli i postavlja se da vrednost $H[i]$ bude z, pri čemu se njegov broj pojavljivanja $Count[i]$ postavlja na jedan. U suprotnom,

Algoritam JellyFish(Z, α, h)
Ulaz: Z je skup od N k-mera, α je faktor opterećenja koji kontroliše veličinu heš tabele i h je heš funkcija.
Izlaz: Broj pojavljivanja svakog k-mera u skupu Z.
1: Kreirati praznu heš tabelu $H[1..\frac{N}{\alpha}]$ tako da svaki ulaz zahteva 2k bitova
2: Kreirati praznu tabelu $Count[1..\frac{N}{\alpha}]$ tako da svaki ulaz zahteva 32 bita
3: foreach k-mer $z \in Z$ do:
4: i = $hashEntry(z, h, \frac{N}{\alpha})$;
5: if $H[i]$ is empty then
6: $H[i] = z$ and $Count[i] = 1$;
7: else
8: $Count[i] = Count[i] + 1$;
9: end if
10: end for
11: Output $(H[i], Count[i])$ za sve ulaze $H[i]$ različite od 0.

Slika 4.1: *JellyFish* algoritam [12]

ako $hashEntry(H, z, h, \frac{N}{\alpha})$ vraća indeks i za koji je vrednost $H[i]$ jednaka z, broj pojavljivanja $Count[i]$ za k-mer z se uvećava za jedan. Nakon sto su svi k-meri iz Z obrađeni, prikazuje se uređeni par $(H[i], Count[i])$ koji sadrži k-mer i njegov broj pojavljivanja, za sve vrednosti $H[i]$ različite od nule.

Algoritam hashEntry(H, z, h, $size$)
1: $i = h(z)$ mod $size$;
2: while $H[i] \neq z$ do
3: $i = i + 1$ mod $size$; /* linearno popunjavanje */
4: end while
5: Return i;

Slika 4.2: Funkcija *hashEntry* [12]

Algoritam *JellyFish* je efikasan, ukoliko ne postoji kolizija. U praksi je očekivani broj kolizija manji, ukoliko za faktor opterećenja važi $\alpha \leq 0.7$. Zatim, očekivano vreme izvršavanja je $O(N)$. Što se tiče prostorne složenosti, tabele H i $Count$ zahtevaju $\frac{N}{\alpha}(2k + 32)$ bitova, pod pretpostavkom da broj zauzima 32 bita [12].

Urađene su dve implementacije ovog algoritma. Prva implementacija simulira prikazani pseudokod, dok druga implementacija koristi mapu iz Elixir-a i njen meha-

nizam razrešavanja kolizija. Kako je druga implementacija efikasnija, ona se koristi u aplikaciji i eksperimentalni rezultati za nju su prikazani u poglavlju 4.7.

Na listingu 4.1 može se videti implementacija funkcije *calculate* algoritma *JellyFish*, koja računa broj pojavljivanja svakog k-mera u datoj listi k-mera. Argumenti ove funkcije su lista k-mera i prazna mapa *count_table* koja na kraju izvršavanja funkcije treba za svaki k-mer iz liste da sadrži informaciju koliko se puta pojavljuje u toj listi. Ona rekurzivno za svaki k-mer proverava da li on već postoji u mapi *count_table*. Provera se vrši pomoću funkcije *Map.get/2*. Ako je povratna vrednost ove funkcije *nil*, to znači da dati k-mer ne postoji u mapi *count_table*, da ga treba dodati i dodeliti mu vrednost 1. U suprotnom, treba samo ažurirati trenutnu vrednost mape za dati k-mer, odnosno uvećati je za jedan. Dodavanje novog elementa u mapu i ažuriranje već postojećeg izvršeno je pomoću funkcije *Map.put/3*. U ovoj implementaciji je iskorišćena mogućnost da se promenljivoj *count_table* dodeli vrednost čitavog *if* bloka, umesto da se dodela vrednosti promenljivoj vrši unutar bloka.

```
1 def calculate([], count_table), do: count_table
2 def calculate([head|tail], count_table) do
3   count_table = if(Map.get(count_table, head) == nil) do
4     Map.put(count_table, head, 1)
5   else
6     Map.put(count_table, head, Map.get(count_table, head) + 1)
7   end
8   count_table = calculate(tail, count_table)
9 end
```

Listing 4.1: Pomoćna funkcija *calculate*

Moguće je izvršiti paralelizaciju izvršavanja algoritma *JellyFish*, jer se pojedinačni rezultati izvršavanja mogu sabrati i od njih se može sklopiti jedan krajnji rezultat. Paralalizacija je u Elixir-u implementirana pomoću modula ***Task***. Najpre se ulazni skup podataka deli na onoliko podskupova koliko računar ima procesora. Zatim se za svaki podskup kreira *task* pomoću funkcije *Task.async/1*. Task predstavlja proces unutar koga se pokreće algoritam *JellyFish*. Funkcija *Task.await/1* kao argument prima kreirani proces i čeka da proces završi izvršavanje i pošalje poruku sa rezultatima izvršavanja. Rezultati se na kraju sabiraju i dobija se konačan rezultat za čitav polazni skup podataka. Na ovaj način izvršavanje je paralelizovano, a ostvareno ubrzanje će biti i eksperimentalno pokazano u poglavlju 4.7.

4.2 Algoritam *DSK*

Algoritam *DSK* je još jedan algoritam za brojanje k-mera, ali vremenski i prostorno efikasniji od algoritma $JellyFish$. Na slici 4.3 se može videti pseudokod ovog algoritma.

Algoritam DSK(Z, M, D, h)
Ulaz: Z je skup od N k-mera, M (bitova) je veličina memorije za čuvanje lista, D (bitova) je veličina diska za čuvanje podlista i h je heš funkcija.
Izlaz: Broj pojavljivanja svakog k-mera u skupu Z.
1: $n_{list} = \frac{2kN}{D}$;
2: $n_{sublist} = \frac{D(2k+32)}{0.7(2k)M}$;
3: for $i = 0$ to $n_{list} - 1$ do
4: Inicijalizovati skup praznih podlista $d_0, ..., d_{n_{sublist}-1}$ na disku
5: for each k-mer $z \in Z$ do
4: if $h(z)$ mod $n_{list} = i$ then
5: $j = (h(z)$ div $n_{list})$ mod $n_{sublist}$;
6: Upisati z na disk u podlistu d_j
7: end if
10: end for
11: for $j = 0$ to $n_{sublist} - 1$ do
12: Učitaj j-tu podlistu d_j;
13: Pokreni $JellyFish(d_j, 0.7, h)$ za ispisivanje broja pojavljivanja svakog k-mera u podlisti d_j
14: end for
15: end for

Slika 4.3: Funkcija *DSK algoritam* [12]

Neka je dat skup k-mera Z, broj bitova D na disku za čuvanje svake liste u koju se smeštaju k-meri, broj bitova M u memoriji za čuvanje svake podliste k-mera i heš funkcija h. Prvo se k-meri iz skupa Z dele u n_{list} lista približno slične dužine. Kako disk ima D bitova i svaki k-mer može biti reprezentovan u $2k$ bitova, svaka lista može čuvati $l_{list} = \frac{D}{2k}$ k-mera. Kako je N broj k-mera u Z, to je $n_{list} = \frac{N}{n_{list}} = \frac{2kN}{D}$. Ovo deljenje se obavlja heš funkcijom h koja ravnomerno mapira sve k-mere u n_{list} lista. Preciznije, za svaki k-mer z iz skupa Z, z se dodeljuje i-toj listi, ako je $h(z)$ mod $n_{list} = i$. Zatim, svaka lista se dalje deli u podliste, pri čemu je svaka dužine $l_{sublist}$. Svaka podlista će biti obrađena u memoriji pomoću algoritma $JellyFish$, koji zahteva $\frac{l_{sublist}}{0.7}(2k+32)$ bitova. Kako memorija ima M bitova, tako je $l_{sublist} = \frac{0.7M}{(2k+32)}$.

Broj podlista je $n_{sublist} = \frac{n_{list}}{n_{sublist}} = \frac{D(2k+32)}{0.7(2k)M}$.

Svaka lista je podeljena u podliste heš funkcijom h. Preciznije, za svaki k-mer s u i-toj listi, s je dodeljen j-toj podlisti, ako je $(\frac{h(s)}{n_{list}})\ mod\ n_{sublist} = j$. Za svaku podlistu dužine $l_{sublist} = 0.7M2k + 32$, koristeći M bitova, vrši se brojanje pojavljivanja svakog k-mera u podlisti koristeći algoritam $JellyFish(d_j, 0.7, h)$ sa slike 4.1.

Algoritam *DSK* će zapisati samo jednom na disk svaki k-mer iz Z, iako će svaki k-mer pročitati n_{list} puta. Stoga, algoritam neće generisati mnogo pristupa disku radi pisanja. Što se tiče vremenske složenosti, za i-tu iteraciju, algoritam numeriše sve k-mere u Z, što oduzima $O(N)$ vremena. Zatim, algoritam identifikuje $\frac{D}{2k}$ k-mera koji pripadaju i-toj listi i zapisuje ih na disk, što oduzima $O(\frac{D}{k})$ vremena. Nakon toga, algoritam čita $\frac{D}{2k}$ k-mera i izvodi brojanje, što oduzima $O(\frac{D}{k})$ vremena. Tako da svaka iteracija zahteva $O(N + \frac{D}{k}) = O(N)$ vremena, gde je $N > \frac{D}{2k}$. Kako je $n_{list} = \frac{2kN}{D}$ broj iteracija, algoritam se izvršava u $O(kN^2)$ očekivanom vremenu. Kada je $D = \theta(N)$, algoritam se izvršava u $O(kN)$ očekivanom vremenu.

Minimizacija broja pristupa disku u implementaciji ovog algoritma nije ostvarena, jer u Elixir-u ne postoji kontrola za to, već je algoritam implementiran samo prateći osnovnu ideju. Osnovna ideja je da se ulazna lista k-mera podeli na više manjih podlista nad kojima će se pokrenuti algoritam *JellyFish* i da se rezultati pojedinačnih pokretanja objedine i prikazu kao krajnji rezultat. Ovakva podela je izuzetno važna u situacijama kada se zbog nedovoljne količine memorije kompletni podaci ne mogu istovremeno obrađivati.

Heš funkcija h, koja se koristi i u algoritmu DSK i u algoritmu $JellyFish$, u okviru ovog rada je predstavljena funkcijom *pattern_to_number* koja koristi pomoćnu funkciju *symbol_to_number*. Pomoćna funkcija *symbol_to_number* kodira nukleotide pomoću mape, gde su A, T, C i G ključevi sa vrednostima 00, 01, 10, 11, respektivno. Funkcija *pattern_to_number* prvo na osnovu stringa *pattern* formira listu karaktera *chars* pomoću funkcije *String.graphemes/1*. Zatim se funkcijom *Enum.map/2* kreira nova lista *mapped_chars* transformacijom svakog karaktera liste *chars* pomoću funkcije *symbol_to_number*. Nova lista karaktera se spaja u string *new_pattern* pomoću funkcije *Enum.join/2*. Na kraju se string *new_pattern* pretvara u ceo broj funkcijom *String.to_integer* i vraća kao povratna vrednost funkcije *pattern_to_number*. Implementacija ovih funkcija u Elixir-u može se videti na listingu 4.2.

```
def symbol_to_number(c) do
  map = %{"A" => "00", "T" => "01", "C" => "10", "G" => "11"}
  map[c]
end

def pattern_to_number(pattern) do
  chars = String.graphemes(pattern)
  mapped_chars = Enum.map(chars, fn char -> symbol_to_number(char)
   end)
  new_pattern = Enum.join(mapped_chars, "")
  String.to_integer(new_pattern, 10)
end
```

Listing 4.2: Funkcija *pattern_to_number* i pomoćna funkcija *symbol_to_number*

4.3 Algoritam *DeBrujinGraph*

De Brojnov grafovski pristup je jedan od pristupa u izgradnji kontiga koji se danas najčešće koristi. Cilj ovog algoritma je da pronađe sve kontige određene veličine, a uz pomoć De Brojnovog grafa. Slika 4.4 daje pseudokod ovog jednostavnog metoda koji na osnovu skupa očitavanja R i parametra k najpre izgrađuje De Brojnov graf H_k, a zatim pronalazi sve maksimalne proste putanje i od njih formira kontige.

Algoritam *DeBrujinAssembler*(R, k)
Ulaz: R je skup očitavanja i k je parametar De Brojnovog grafa.
Izlaz: Skup kontiga A.
1: Generisati De Brojnov graf H_k za skup R;
2: Izdvojiti sve maksimalne proste putanje u H_k kao kontige;

Slika 4.4: De Brojnov grafovski asembler [12]

U okviru ovog rada, implementiran je algoritam *DeBrujinGraph* koji se koristi u prvom koraku De Brojnovog grafovskog asemblera. Argumenti ovog alogoritma su string koji predstavlja jedno očitavanje iz skupa R i parametar k. Prvi korak je kreiranje skupa k-mera na osnovu očitavanja i parametra k. Iz datog stringa, koji predstavlja očitavanje, izdvaja se skup svih mogućih podstringova dužine k. Ovaj skup predstavlja čvorove budućeg De Brojnovog grafa. Zatim se vrši povezivanje

svaka dva k-mera tako da je prefiks prvog jednak sufiksu drugog. Na taj način se kreira De Brojnov graf.

Povratna vrednost algoritma je mapa koja predstavlja De Brojnov graf. Svaki ključ mape predstavlja jedan čvor grafa, a njegova vrednost je lista k-mera koji predstavljaju čvorove sa kojima pomenuti čvor formira granu.

4.4 Algoritam *AllEulerianCycles*

U poglavlju 3.5 je objašnjen pojam Ojlerove putanje i značaj u procesu rekonstrukcije genoma. U tom procesu, algoritam za pronalaženje Ojlerove putanje je neophodan. Ovaj algoritam nije NP-kompletan i putanja se može efikasno pronaći [11]. Opis problema pronalaženja Ojlerove putanje na osnovu De Brojnovog grafa može se videti na slici 4.5. U okviru ovog rada implementiran je algoritam koji pronalazi sve Ojlerove putanje u grafu i njegov pseudokod se može videti na slici 4.6.

Problem Ojlerove putanje: Pronaći Ojlerovu putanju u grafu.
Ulaz: Graf.
Izlaz: Putanja koja posećuje svaku granu u grafu tačno jednom.

Slika 4.5: Problem Ojlerove putanje

Pri pronalaženju svih Ojlerovih putanja u grafu problem je što neki čvorovi mogu imati više izlaznih i ulaznih grana, te se ne može jednoznačno odrediti čvor koji treba sledeći da se poseti. Broj ulaznih grana u dati čvor naziva se **ulazni stepen** čvora, a broj izlaznih grana iz datog čvora **izlazni stepen** čvora. Sa druge strane, u prostom usmerenom grafu u kome svaki čvor ima i ulazni i izlazni stepen jednak 1, može se jednoznačno odrediti naredni čvor koji treba da se poseti, jer u takvom grafu postoji samo jedan Ojlerov ciklus. Ideja algoritma *AllEulerianCycles* je da se usmereni graf koji sadrži $n \geq 1$ Ojlerovih ciklusa transformiše na n prostih usmerenih grafova koji sadrže tačno jedan Ojlerov ciklus. Ova transformacija ima svojstvo da je lako invertibilna, odnosno na osnovu jedinstvenog Ojlerovog ciklusa jednog od prostih usmerenih grafova moguće je rekonstruisati originalni Ojlerov ciklus polaznog grafa.

Neka je dat čvor v iz grafa G koji ima ulazni stepen veći od jedan sa ulaznom granom (u, v) i izlaznom granom (v, w). Konstruiše se jednostavniji (u, v, w)-obilazni

graf u kome se uklanjaju grane (u, v) i (v, w), a dodaje se novi čvor x zajedno sa granama (u, x) i (v, x). Nove grane (u, x) i (v, x) u obilaznom grafu nasleđuju oznake uklonjenih grana (u, v) i (v, w), respektivno. S obzirom na ulaznu granu (u, v) u čvor v, koji ima k izlaznih grana (v, w_1),..., (v, w_k), može se konstruisati k različitih obilaznih grafova. Nikoja dva obilazna grafa nemaju isti Ojlerov ciklus. Cilj je iterativno konstruisati svaki mogući obilazni graf za dati graf sve dok se ne dobije velika porodica prostih usmerenih grafova. Svaki od ovih grafova će odgovarati različitom Ojlerovom ciklusu datog grafa [5].

Algoritam AllEulerianCycles(*graph*)
Ulaz: De Brojnov graf *graph*.
Izlaz: Sve Ojlerove putanje u De Brojnovom grafu *graph*.
1: *AllGraphs* = skup koji se sastoji samo od jednog grafa *graph*
2: while postoji složeni graf G u skupu *AllGraphs*
3: v = čvor sa ulaznim stepenom većim od 1 u grafu G
4: foreach ulazna grana (u, v) u čvor v
5: *NewGraph* = (u, v, w)-obilazni graf grafa G
6: if *NewGraph* je povezan
7: Dodati *NewGraph* u skup *AllGraphs*
8: Ukloniti graf G iz skupa *AllGraphs*
9: foreach graf G iz skupa *AllGraphs*
10: Output Ojlerov ciklus u G

Slika 4.6: Algoritam *AllEulerianCycles* [5]

Funkcija *calculate_in_degree*, čija je implementacija prikazana na listingu 4.3, računa ulazne stepene za sve čvorove datog grafa *graph*. Graf je u implementaciji predstavljen mapom. Ključ mape je čvor grafa, a lista vrednosti ključa čvorovi sa kojima dati čvor gradi grane u grafu, odnosno njegovi izlazni čvorovi. Ulazni stepen čvora je broj grana koje ulaze u taj čvor, odnosno broj ključeva mape u čijem skupu vrednosti se nalazi taj čvor. Funkcija *calculate_in_degree* koristi tri pomoćne funkcije: *initialize*, *iterate_keys* i *iterate_values*. Pomoćna funkcija *initialize* inicijalizuje mapu *all_degrees* tako da joj ključevi budu čvorovi grafa i da svaki ima vrednost 0. Pomoćne funkcije *iterate_keys* i *iterate_values* omogućavaju da se u jednom prolazu kroz graf izračuna ulazni stepen svakog čvora. Za svaki čvor grafa se prolazi kroz listu njegovih izlaznih čvorova i svakom od izlaznih čvrova ulazni stepen se uvećava za jedan, odnosno njegova vrednost u mapi *all_degrees* se uvećava za jedan. Funkcija *iterate_keys* služi za obilazak svih čvorova grafa, a funkcija

iterate_values za obilazak svih izlaznih čvorova za dati čvor i uvećavanje njihovih ulaznih stepena.

```
def iterate_values([], all_degrees), do: all_degrees
def iterate_values([head|tail], all_degrees) do
  all_degrees = Map.put(all_degrees, head, Map.get(all_degrees,
   head) + 1)
  all_degrees = iterate_values(tail, all_degrees)
end

def iterate_keys([], graph, all_degrees), do: all_degrees
def iterate_keys([head|tail], graph, all_degrees) do
  all_degrees = iterate_values(Map.get(graph, head), all_degrees)
  all_degrees = iterate_keys(tail, graph, all_degrees)
end

def initialize([], all_degrees), do: all_degrees
def initialize([head|tail], all_degrees) do
  all_degrees = Map.put(all_degrees, head, 0)
  all_degrees = initialize(tail, all_degrees)
end

def calculate_in_degree(graph) do
  all_degrees = initialize(Map.keys(graph), %{})
  all_degrees = iterate_keys(Map.keys(graph), graph, all_degrees)
end
```

Listing 4.3: Funkcija *calculate_in_degree*

4.5 Uputstvo za pokretanje aplikacije

Celokupna implementacija grafičkog korisničkog interfejsa i algoritama je dostupna i može se preuzeti sa adrese [1]. Kako Elixir nema naprednu podršku za grafički korisnički interfejs i to nije njegova osnovna namena, za potrebe ovog rada iskorišćena je i podrška programskog jezika Python i integrisanog razvojnog okruženja ***PyCharm*** [24].

Projekat je organizovan tako da se sastoji od četiri foldera: *data_files*, *result_files*, *source_files* i *executables*. U folderu *data_files* nalaze se tekstualni fajlovi koji sadrže ulazne podatke na kojima je moguće testirati implementirane algoritme (*JellyFishData.txt*, *DSKData.txt*, *DeBrujinGraphData.txt*, *AllEulerianCycles-*

Data.txt). U folderu *result_files* nalaze se tekstualni fajlovi u kojima se nalaze rezultati izvršavanja algoritama (*JellyFishOutput.txt*, *DSKOutput.txt*, *DeBrujinGraphOutput.txt*, *AllEulerianCyclesOutput.txt*). Format ulaznih i izlaznih fajlova za svaki pojedinačni algoritam biće opisan u poglavlju 4.6. Folder *source_files* sadrži izvorne fajlove u programskom jeziku Elixir u kojima se nalazi implementacija algoritama (*jellyfish.exs*, *dsk.exs*, *debrujin_graph.exs*, *all_eulerian_cycles.exs*). Izvršne verzije izvornih fajlova nalaze se u folderu *executables*. Pored ova četiri foldera, projekat sadrži i *.py* fajl (*interface.py*) u kome se nalazi interfejs koji sjedinjuje algoritme u jedinstvenu aplikaciju.

Implementirani algoritmi se mogu pokretati direktno uz pomoć Elixir interpretera. Potrebno je izvršiti komandu *elixir ime_algoritma.exs* ili *iex ime_algoritma.exs* za kojom slede argumenti u zagradi. Izvršne verzije Elixir fajlova se mogu pokretati iz komandne linije i bez instaliranja Elixir-a. Python interfejs podrazumeva korišćenje izvršnih programa. Za dobijanje izvršnih programa iz *.exs* fajlova potrebno je izvršiti sledeće korake u komandnoj liniji:

- pozicionirati se na lokaciju gde zelite da kreirate projekat
- izvršiti komandu *mix*[1] *new ime_projekta*
- kompajlirati projekat pomoću komande *mix* ili *mix compile* u roditeljskom direktorijumu
- u *lib* folderu projekta kreirati novi folder koji nosi naziv željenog algoritma i u njemu fajl *cli.ex* koji ima sadržaj kao na listingu 4.4
- u *lib* folder prekopirati izvorni fajl algoritma
- izvršiti komandu *mix escript.build* u roditeljskom direktorijumu

Nakon izvršenih komandi, u roditeljskom direktorijumu će se pojaviti izvršna verzija izvornog fajla.

```
1 defmodule ImeAlgoritma.CLI do
2   def main(args) do
3     options = [switches: [file: :string],aliases: [f: :file]]
4     {opts,_,_}= OptionParser.parse(args, options)
5     IO.inspect opts, label: "Command Line Arguments"
6   end
```

[1]Mix je alat koji se isporučuje uz Elixir i omogućava kreiranje, kompajliranje i testiranje aplikacije, kao i upravljanje njenim zavisnostima [16].

```
7 end
```

Listing 4.4: Sadržaj fajla *cli.ex*

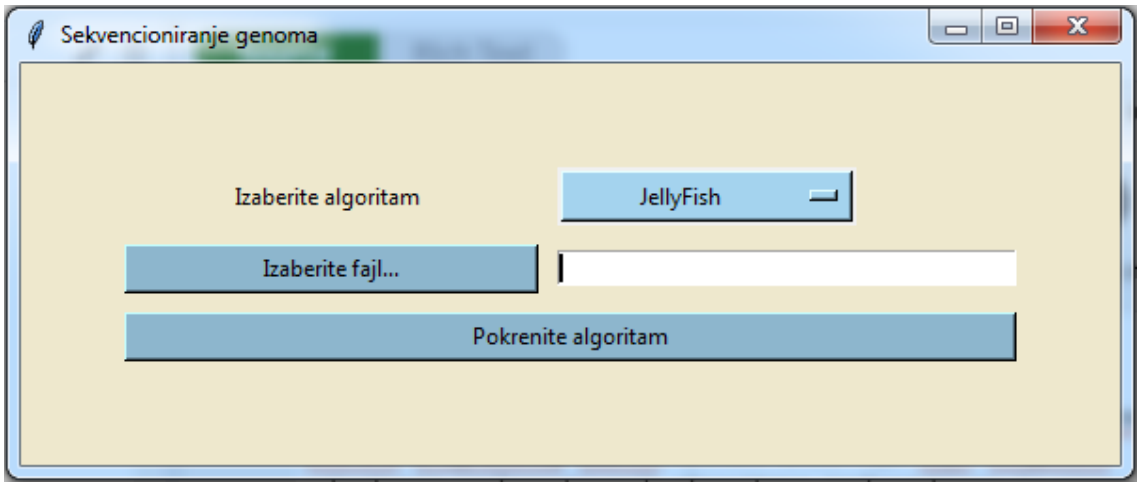
Aplikacija se može pokrenuti direktno iz Python interpretera pokretanjem izvršne verzije fajla *interface.py* koja se dobija njegovim kompajliranjem komandom *python interface.py*.

4.6 Opis aplikacije i primeri upotrebe

Elixir omogućava brzu obradu velike količine podataka i veoma je tolerantan na greške, te ga to čini dobrim izborom za probleme bioinformatičke prirode. U nastavku će biti dat kratak prikaz i opis funkcionalnosti koje pruža implementirani interfejs, a potom i primeri pokretanja algoritama za izabrane ulazne fajlove.

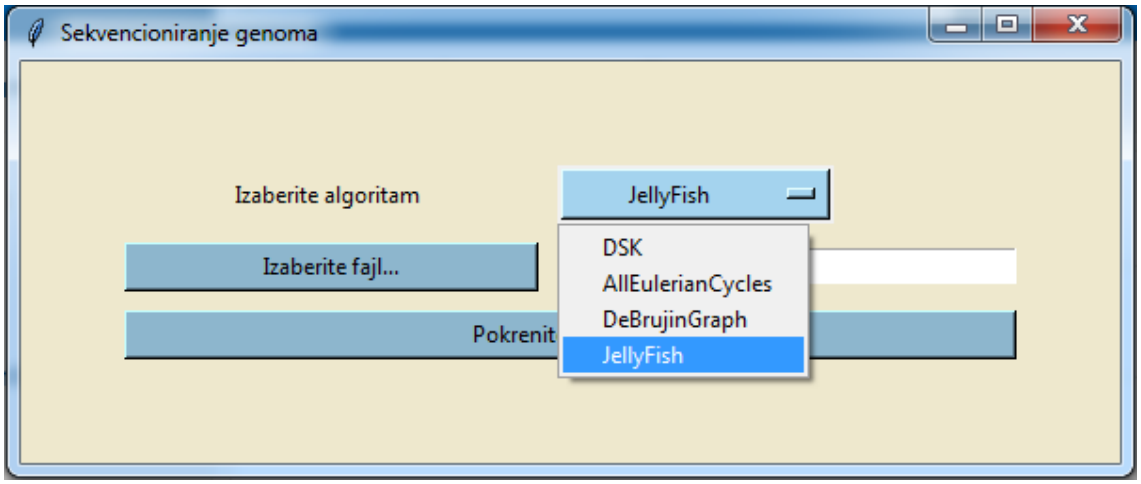
Opis aplikacije

Pri pokretanju aplikacije otvara se glavni prozor prikazan na slici 4.7. Klikom na padajuću listu mogu se videti dostupni algoritmi za izvršavanje, a potom i izabrati neki od njih. Prikaz padajuće liste i izbor jednog od algoritama može se videti na slici 4.8.

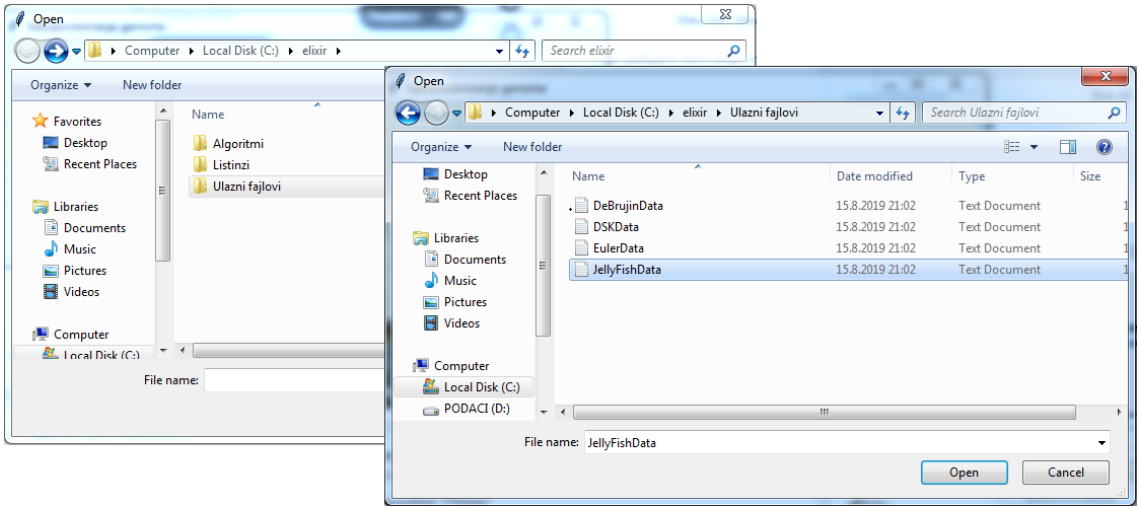


Slika 4.7: Glavni prozor programa

Nakon toga je potrebno izabrati fajl iz kojeg će se čitati ulazni podaci. Klikom na dugme *Izaberite fajl...* otvara se novi dijalog u okviru kojeg se može izabrati ulazni tekstualni fajl, što se može videti na slici 4.9.



Slika 4.8: Izbor algoritma

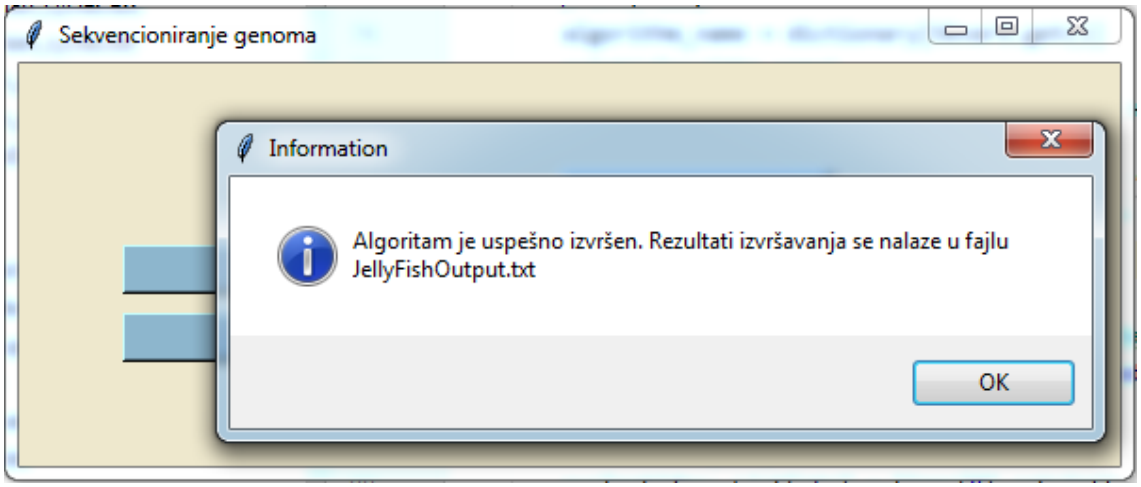


Slika 4.9: Izbor ulaznog fajla

Na kraju je potrebno pokrenuti algoritam klikom na dugme *Pokrenite algoritam*. U pozadini se pokreće odgovarajući izvršni fajl algoritma sa argumentima koji su pročitani iz ulaznog fajla. Na ovaj način se izbegava kompajliranje fajla u kome se nalazi implementacija algoritma pri svakom kliku na dugme za pokretanje. Pritom se omogućava i izvršavanje algoritama na mašini na kojoj ne postoji instaliran Elixir.

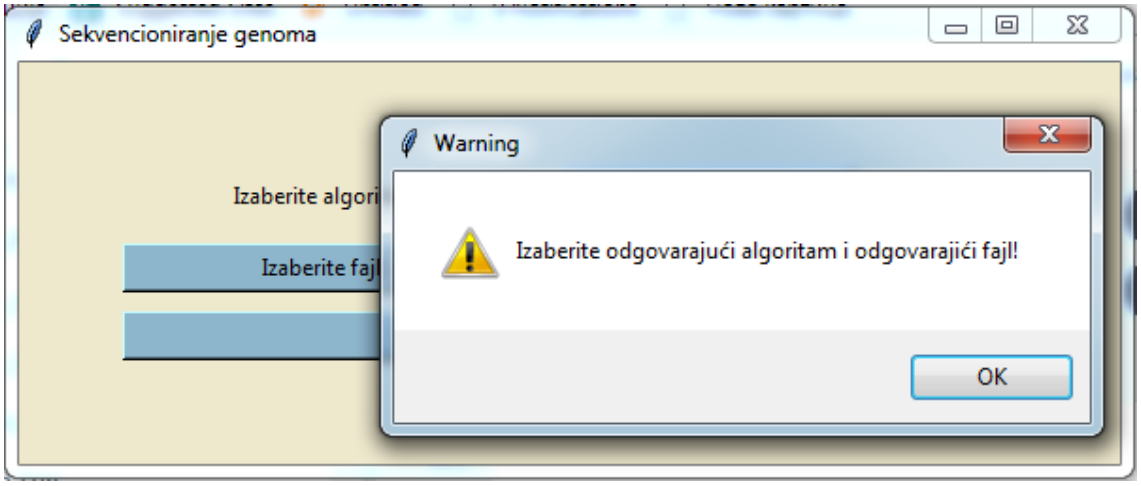
Ukoliko je algoritam uspešno izvršen, rezultati pokretanja će biti upisani u odgovarajući tekstualni fajl i pojaviće se obaveštenje o uspešnom izvršavanju i nazivu rezultujućeg fajla. Prikaz ovog obaveštenja dat je na slici 4.10.

Ukoliko je prilikom pokretanja algoritma detektovano nepoklapanje u izabranom



Slika 4.10: Uspešno izvršavanje algoritma

algoritmu i ulaznom fajlu, pojaviće se upozorenje koje je prikazano na slici 4.11. U tom slučaju je potrebno proveriti izabrane vrednosti i ponovo pokrenuti algoritam.



Slika 4.11: Pogrešan izbor algoritma ili ulaznog fajla

Primeri upotrebe

Algoritam *JellyFish* - Ulazni fajl *JellyFishData.txt* sadrži k-mere razdvojene zarezom. Nakon izbora ovog ulaznog fajla i pokretanja algoritma *JellyFish*, rezultat je smešten u tekstualnom fajlu *JellyFishOutput.txt*. I u sekvencijalnoj i u paralelnoj varijanti izvršavanja ulazni fajl ima isti sadržaj i format. Primer ulaznog fajla koji sadrži k-mere dužine dva može se videti na listingu 4.5. Rezultati izvršavanja su

isti u obe varijante, odnosno dati su mapom koja za svaki k-mer iz ulaznog skupa k-mera sadrži njegov broj pojavljivanja. Izlazni fajl za dati primer ulaznih podataka prikazan je na listingu 4.6.

```
AC,CG,AC,GT,CA,GG,AC,GT
```

Listing 4.5: Primer ulaznog fajla za algoritam *JellyFish*

```
%{
    "CG" => 1,
    "CA" => 1,
    "GG" => 1,
    "GT" => 2,
    "AC" => 3,
}
```

Listing 4.6: Rezultati izvršavanja algoritma *JellyFish*

Na malom skupu podataka, kakav je dat na listingu 4.5, nema mnogo smisla vršiti paralelizaciju izvršavanja. Primer gde ima smisla vršiti podelu ulaznog skupa podataka i paralelizovati izvršavanje biće dat u poglavlju 4.7.

Algoritam *DSK* - Ulazni fajl *DSKData.txt* za algoritam *DSK* ima isti format kao i ulazni fajl za algoritam *JellyFish*, samo što u prvoj liniji sadrži i broj koji predstavlja maksimalnu veličinu ulaznog fajla izraženu u kilobajtima koju algoritam *JellyFish* može da obradi s obzirom na dostupnu memoriju. Primer ulaznog fajla može se videti na listingu 4.7.

```
AC,CG,AC,GT,CA,GG,AC,GT
3
```

Listing 4.7: Primer ulaznog fajla za algoritam *DSK*

Rezultati izvršavanja algoritma *DSK* za dati primer ulaznog fajla smešteni su u fajlu *DSKOutput.txt* i mogu se videti na listingu 4.8. Izlazni fajl ima isti format kao i izlazni fajl za algoritam *JellyFish*.

```
%{
    "CG" => 1,
    "CA" => 1,
    "GG" => 1,
    "GT" => 2,
    "AC" => 3,
}
```

Listing 4.8: Rezultati izvršavanja *DSK* algoritma

Algoritam *DeBrujinGraph* - Ulazni fajl *DeBrujinData.txt* sadrži jedno očitavanje i u prvoj liniji parametar k. Primer ulaznog fajla parametrom $k = 2$ i očitavanjem dužine 17 može se videti na listingu 4.9.

```
TAATGCCATGGGATGTT
2
```

Listing 4.9: Primer ulaznog fajla za algoritam *DeBrujinGraph*

Nakon izvršavanja algoritma *DeBrujinGraph*, rezultat se može videti u tekstualnom fajlu *DeBrujinGraphOutput.txt*. Rezultat izvršavanja algoritma nad jednim očitavanjem predstavlja jedan De Brojnov graf. Na listingu 4.10 se nalaze rezultati izvršavanja za ulazne podatke koji su dati na listingu 4.9. Izlazni fajl sadrži mapu na osnovu koje se može konstruisati De Brojnov graf. Mapa je zapisana tako da elementi odvojeni sa ; predstavljaju jedan element mape, čiji je ključ k-mer sa leve strane znaka :, a vrednost lista k-mera koji se nalaze sa desne strane znaka : i razdvojeni su zarezom.

```
AA:AT;AT:TG,TG,TG;AT:TG,TG,TG;CA:AT;CC:CA;GA:AT;GC:CC;GG:GG,GA;GT:
TT;TA,AA;TG:GC,GG,GT;TT:
```

Listing 4.10: Rezultati izvršavanja algoritma *DeBrujinGraph*

Ovde je dat primer ulaznog fajla koji sadrži jedno očitavanje i rezultat izvršavanja algoritma nad njim. U implementaciji se algoritam izvršava nad fajlom koji sadrži više od jednog očitavanja tako što se redom poziva za svako očitavanje.

Algoritam *AllEulerianCycles* - Ulazni fajl EulerData.txt sadrži De Brojnov graf, koji je zapisan u istom formatu kao i izlazni fajl za algoritam *DeBrujinGraph* sa listinga 4.10. Primer ulaznog fajla može se videti na listingu 4.11.

```
AT:TC;TC:CG;CG:GA,GG;GA:AT,AC;AC:CG;GG:GA
```

Listing 4.11: Primer ulaznog fajla za algoritam *AllEulerianCycles* algoritma

Nakon izvršavanja algoritma, rezultat je smešten u tekstualnom fajlu *AllEulerianCyclesOutput.txt*. Izlazni fajl sadrži listu kontiga, koje predstavljaju cikluse u De Brojnovom grafu. Rezultati izvršavanja za ulazni fajl sa listinga 4.11 prikazani su na listingu 4.12.

```
["ACGATCGGAC","ATCGGACGAT","CGGACGATCG","CGATCGGACG","GACGATCGGA",
"GATCGGACGA","GGACGATCGG","TCGGACGATC","ACGGATCGAC","ATCGACGGAT",
"CGACGGATCG","CGGATCGACG","GATCGACGGA","GACGGATCGA","GGATCGACGG",
"TCGACGGATC"]
```

Listing 4.12: Rezultati izvršavanja *AllEulerianCycles* algoritma

Slično kao i u slučaju algoritma *DeBrujinGraph*, ovde je prikazan ulazni fajl koji sadrži samo jedan graf. U implementaciji algoritam *AllEulerCycles* obrađuje više od jednog grafa tako što se poziva za svaki graf iz ulaznog fajla.

4.7 Rezultati

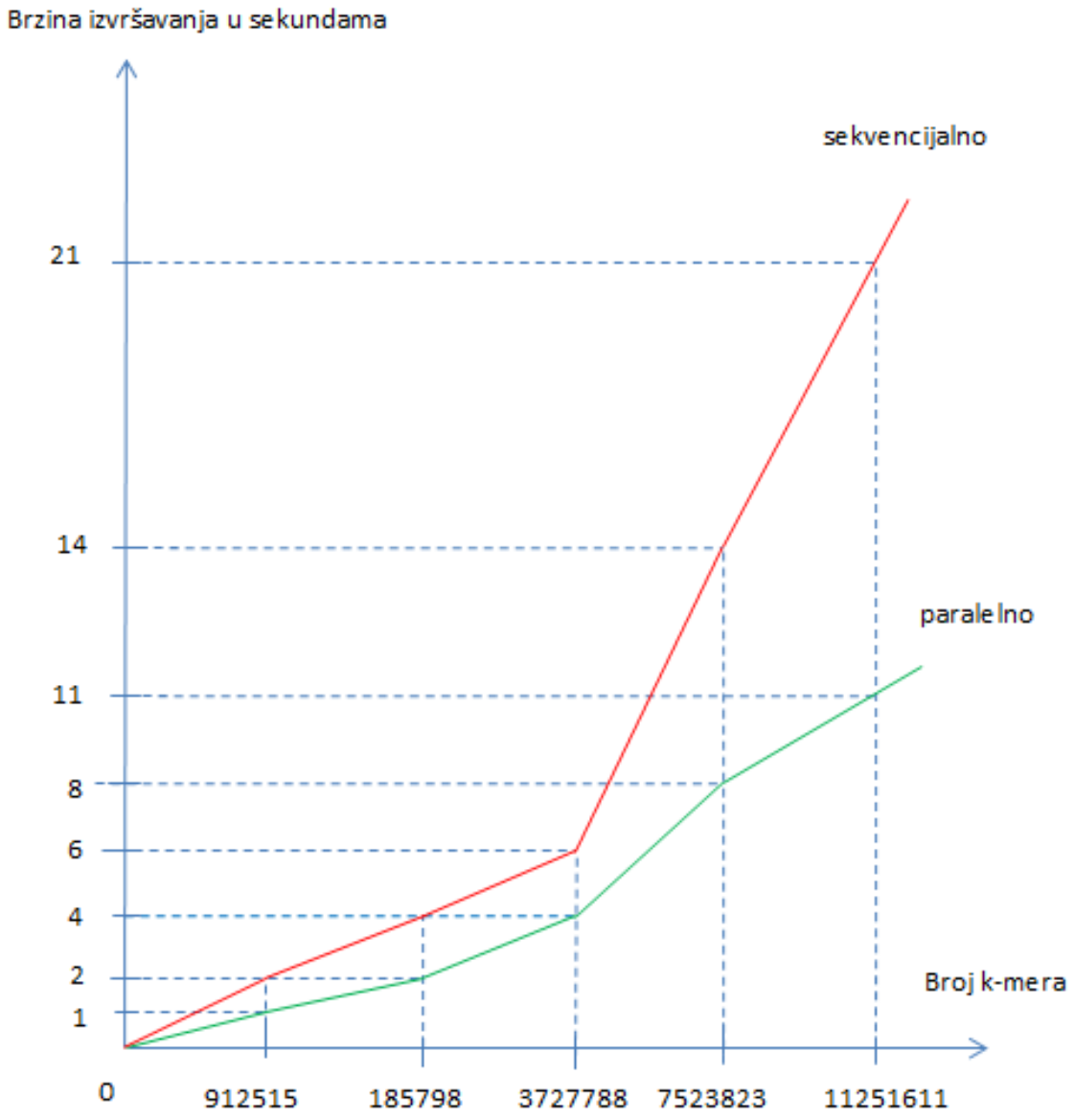
Prikazani algoritmi su pokretani nad k-merima ekstrahovanim iz očitavanja bakterije *Staphylococcus aureus* koja su napravljena sekvencerom *Illumina MiSeq*. Eksperimentalna merenja su izvršena na računaru marke *Asus* sa 64-bitim operativnim sistemom *Windows 7 Ultimate*, procesorom *Intel Core i3-2330M CPU @ 2.20GHz* sa četiri jezgra i 4GB RAM memorije. Merenja su izvršena sa ciljem eksperimentalnog utvrđivanja efikasnosti i skalabilnosti implementiranih algoritama.

Algoritam *JellyFish* - Vršeno je poređenje sekvencijalne i paralelne verzije algoritma *JellyFish* kako bi se utvrdio faktor ubrzanja postignut paralelizacijom. Na dijagramu sa slike 4.12 se mogu videti brzine sekvencijalnog i paralelnog izvršavanja u zavisnosti od količine podataka.

Eksperimentalnim merenjem je utvrđena granica veličine podataka, odnosno broj k-mera koji mogu biti obrađeni algoritmom *JellyFish* na datoj mašini. Ta granica iznosi 120KB, odnosno oko 11300000 k-mera (u našem slučaju k-meri su dužine deset). Vreme za koje ovaj broj k-mera biva obrađen paralelnim izvršavanjem je 11 sekundi, dok je u slučaju sekvencijalnog izvršavanja 21 sekund.

Algoritam *DSK* - Algoritmom *DSK* je izvršena obrada svih 90403620 k-mera dužine deset ekstarhovanih iz 462036 očitavanja različite dužine bakterije *Staphylococcus aureus*. Početni skup k-mera je najpre podeljen na podskupove koji se mogu obraditi sa dostupnom memorijom. Kako je memorijska granica za računar na kojem je vršena obrada 120KB, to je skup svih očitavanja bio podeljen na osam podskupova. Potom je svaki podskup obrađen paralelno, odnosno izvršeno je dalje deljenje svakog podskupa na četiri dela (jer računar poseduje četiri procesora) i na svakom delu je paralelno pokrenut algoritam *JellyFish*. Ukupno vreme izvršavanja bilo je 93 sekunde. Nakon završetka obrade, generisano je osam izlaznih fajlova sa rezultatima. Ove rezultate je moguće sabrati i od njih formirati krajnji izlazni fajl koji sadrži broj pojavljivanja za svaki k-mer dužine deset iz skupa svih k-mera bakterije. Sabiranje svih rezultata je izvršeno za 16 sekundi, tako da je ukupno vreme obrade svih k-mera algoritmom *DSK* 109 sekundi.

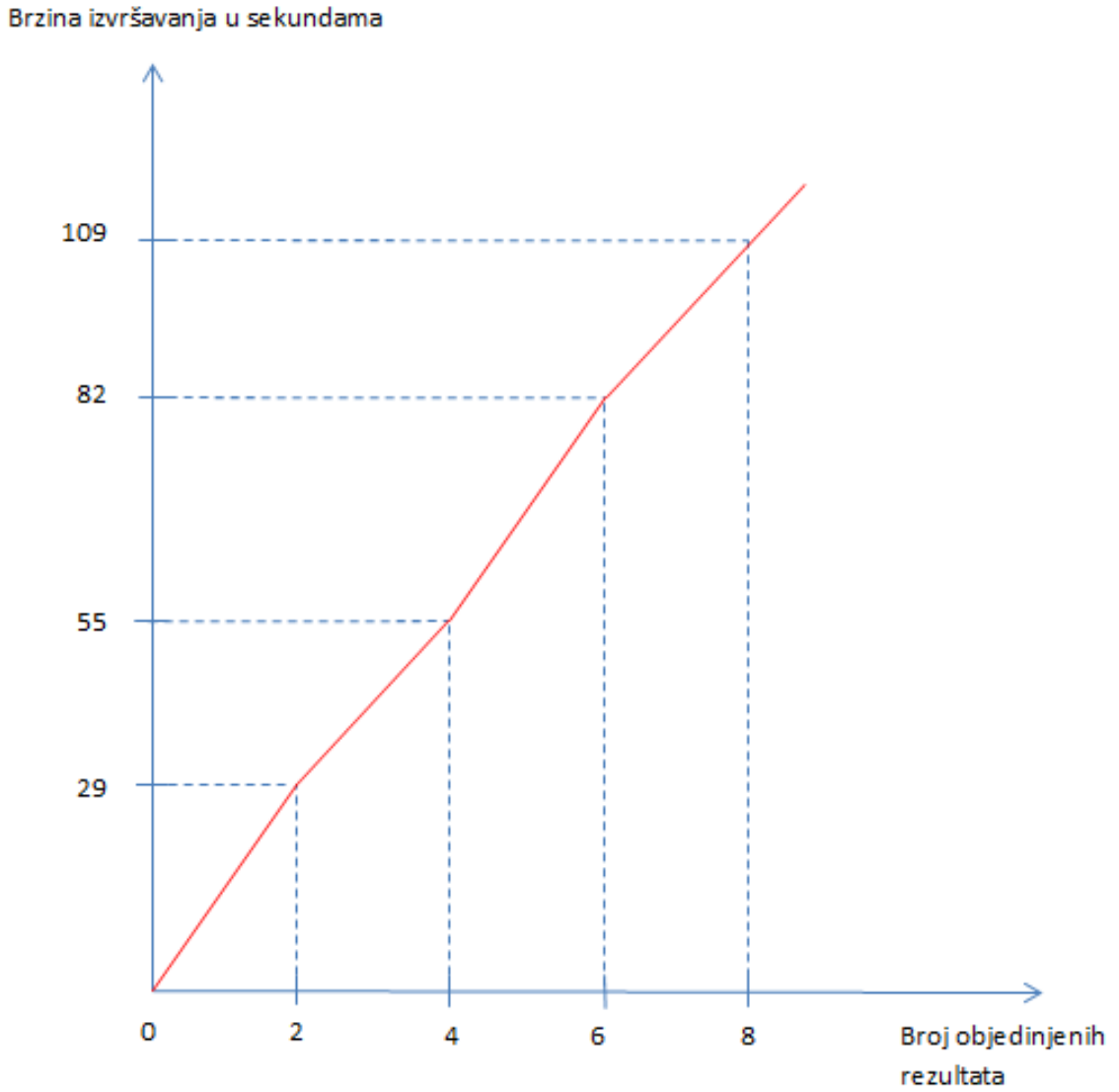
Kako bi se ocenile performanse algoritma *DSK*, merene su brzine njegovog izvr-



Slika 4.12: Dijagram sekvencijalnog i paralelnog izvršavanja

šavanja uz postepeno povećavanje broja izlaznih fajlova koje je potrebno objediniti u jedan. Dobijeni rezultati se mogu videti na dijagramu sa slike 4.13.

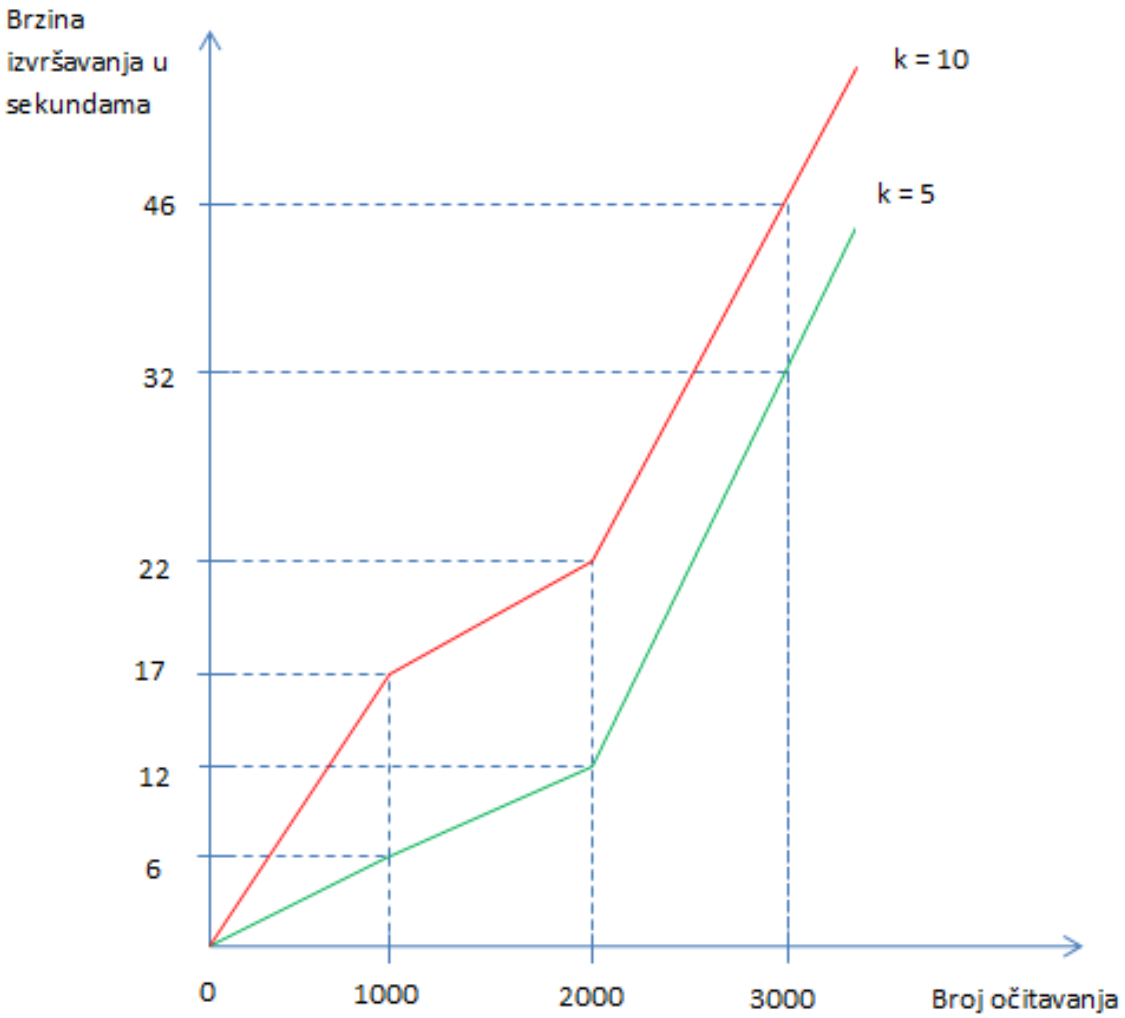
Algoritam *DeBrujinGraph* - Performanse algoritma *DeBrujinGraph* su merene na manjim podskupovima skupa svih očitavanja bakterije *Staphylococcus aureus*. Dobijeni rezultati su prikazani dijagramom na slici 4.14. Na dijagramu se može videti i zavisnost brzine izvršavanja algoritma od vrednosti parametra k. Za $k = 5$ algoritam se izvršava 43% brže nego u slučaju kada je $k = 10$. U oba slučaja



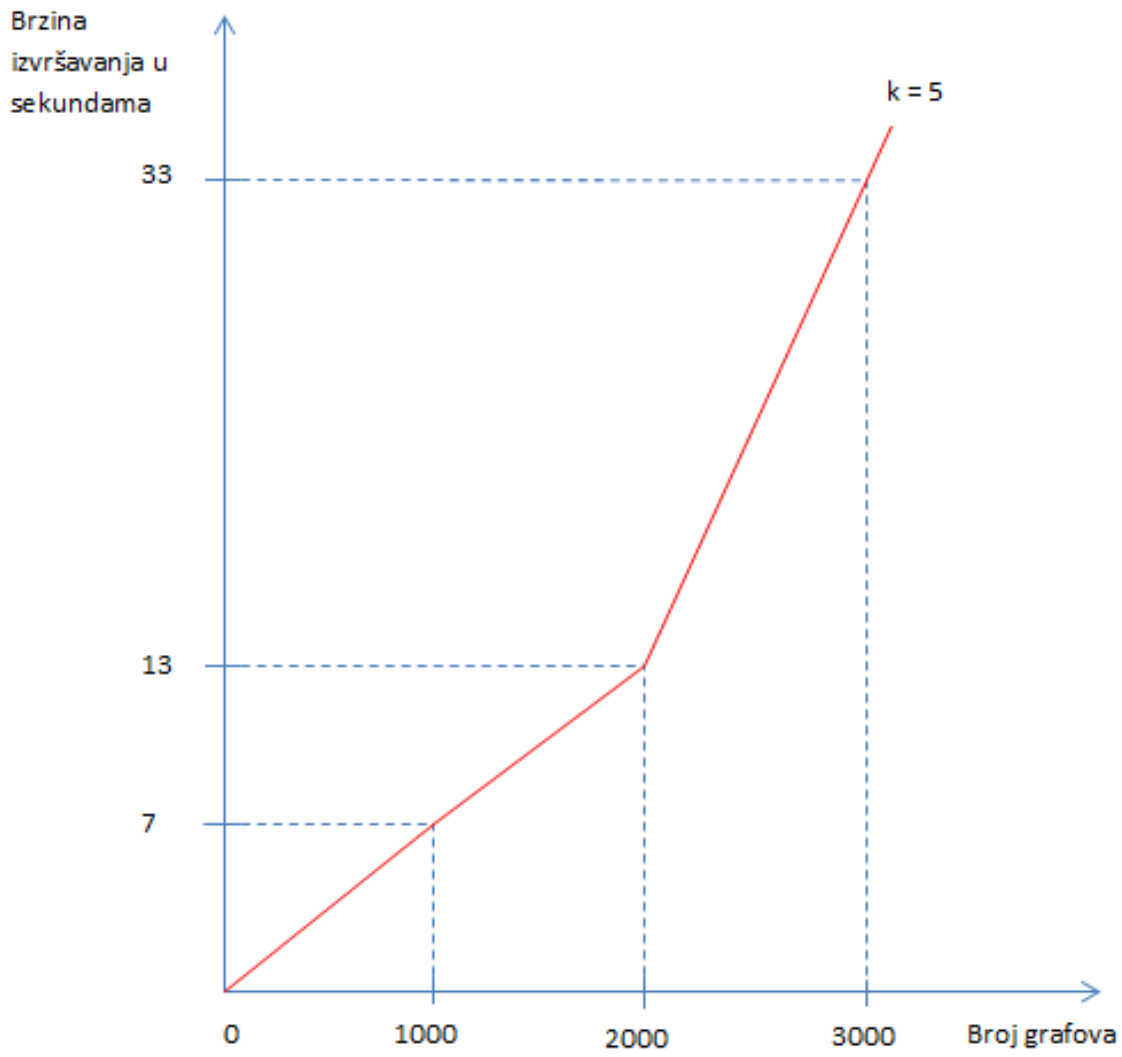
Slika 4.13: Dijagram izvršavanja algoritma *DSK*

se obrađuje ista količina podataka, odnosno isti broj očitavanja, ali je razlika u broju k-mera koji bivaju kreirani na osnovu očitavanja i obrađeni. Izršavanjem algoritma za 1000 očitavanja, kreira se i obrađuje 177634 k-mera dužine pet, a 172634 k-mera dužine deset.

Algoritam *AllEulerianCycles* - Performanse algoritma *AllEulerianCycles* merene su za 1000, 2000 i 3000 grafova koji su dobijeni kao rezultat izvršavanja algoritma *DeBrujinGraph* za 1000, 2000 i 3000 očitavanja, pri čemu je parametar $k = 5$. Na dijagramu sa slike 4.15 može se videti zavisnost brzine izvršavanja od broja ulaznih grafova.



Slika 4.14: Dijagram izvršavanja algoritma *DeBrujinGraph*



Slika 4.15: Dijagram izvršavanja algoritma *AllEulerianCycles*

Glava 5

Zaključak

U ovom radu su prikazane osnove sintakse i semantike programskog jezika Elixir. On je iskorišćen za implementaciju algoritama u oblasti sekvencioniranja genoma. Sekvencioniranje genoma je proces otkrivanja sastava genoma i od izuzetnog je značaja za modernu dijagnostiku, a posebno za dijegnostiku rizika od naslednih bolesti. Na ovaj način je prikazana primena Elixir-a u rešavanju realnih problema bioinformatičke prirode.

Kako je Elixir funkcionalni programski jezik, on omogućava da razmišljamo u terminima funkcija koje transformišu podatke. To znači da sama transformacija nikada ne menja podatke, već svaka primena funkcije potencijalno stvara novu, svežiju verziju podataka. U okviru rada ova osobina je iskorišćena u implementaciji svakog algoritma, jer se svaka implementacija sastoji od velikog broja funkcija koje obavljaju neki celovit posao. Zahvaljujući tome dobijen je koncizan kôd, povećana je njegova čitljivost i programiranje je bilo brže. Dodatno, ova karakteristika smanjuje potrebu za mehanizmima za sinhronizaciju podataka, pa je implementacija konkurentnosti algoritma *JellyFish* bila jednostavna i pouzdana. Konkurentnost je jedna od glavnih karakteristika Elixir-a. Stvaranje već samo četiri procesa je značajno smanjilo vreme izvršavanja algoritama *JellyFish* i *DSK*, a na mašini sa većim brojem dostupnih procesora moglo bi se očekivati i veće poboljšanje. Zahvaljujući jednostavnoj i modernoj sintaksi, koja je prirodna za čitanje, ovaj programski jezik se veoma brzo savladava i uči. To je pomoglo da se veoma brzo nakon upoznavanja sa osnovama sintakse i semantike počne sa implementacijom algoritama i postigne visoka produktivnost.

Algoritam *JellyFish* i algoritam *DSK* su testirani na realnim podacima. Algoritam *JellyFish* je pokazao izuzetnu brzinu i tačnost i prilikom sekvencijalne i

prilikom paralelne obrade. Implementacijom konkurentnosti dobijeno je skoro dva puta veće ubrzanje izvršavanja (na računaru sa svega četiri procesora) u odnosu na sekvencijalno izvršavanje. Na taj način je ubrzana i olakšana obrada celokupnog skupa podataka. Ostali algoritmi su testirani takođe na realnim podacima, ali na manjim skupovima podataka. Za njih nije implementirana konkurentnost, jer je to, zbog kompleksnosti algoritama, prilično izazovan zadatak. Rezultati eksperimentalnih merenja su zadovoljavajući, iako su merenja izvršena na kućnom računaru skromnog kapaciteta. Za realne slučajeve upotrebe, potrebno je koristiti računar značajno većeg kapaciteta. Na taj način bi i implementirana konkurentnost došla do većeg izražaja.

.

Literatura

[1] *Adresa za preuzimanje materijala sa github-a.* URL: https://github.com/MilenaDukanac/Master-rad.

[2] Joe Armstrong. "Making reliable distributed systems in the presence of software errors". PhD thesis. 2003.

[3] David Austin. *What is DNA?* URL: http://www.ams.org/publicoutreach/feature-column/fcarc-dna-puzzle.

[4] Francesco Cesarini and Simon Thompson. *Erlang Programming. A Concurrent Approach to Software Development.* O'Reilly Media, June 2009.

[5] Phillip Compeau i Pavel Pevzner. *Bioinformatics Algorithms: An Active Learning Approach.* Sv. 1. 2. Active Learning Publishers, 2015, str. 115–180. URL: http://bioinformaticsalgorithms.com/.

[6] Robert Ekblom and Jochen B W Wolf. "A field guide to whole-genome sequencing, assembly and annotation". In: *Evolutionary Applications* 7 (Nov. 2014), pp. 1026–1042. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4231593/.

[7] Ramana M. Idury i Michael S. Waterman. *A New Algorithm for DNA Sequence Assembly.* Sv. 2. 2. Mary Ann Liebert, Inc. Submission Benefits Package, Apr. 2009. URL: https://www.liebertpub.com/doi/10.1089/cmb.1995.2.291.

[8] Martin Middendorf. "The shortest common nonsubsequence problem is NP-complete". In: *Theoretical Computer Science* (1993), pp. 365–369. URL: https://www.sciencedirect.com/science/article/pii/030439759390200D.

[9] Charles Petit. "A Brief History of Erlang and Elixir". In: *Coder Stories* (Jan. 2019). URL: https://www.welcometothejungle.co/en/articles/history-erlang-elixir.

[10] *Sequencing Coverage*. URL: https://www.illumina.com/science/technology/next-generation-sequencing/plan-experiments/coverage.html.

[11] Faculty of Mathematics Students of University of Belgrade. *Uvod u bioinformatiku - beleške sa predavanja*. 2018, str. 47–68.

[12] Wing-Kin Sung. English. Ed. by CRC Press. 2017, pp. 123–146.

[13] Dave Thomas. *Programming Elixir 1.3*. Ed. by The Pragmatic Bookshelf. 2016.

[14] Dave Thomas. *Programming Elixir 1.6: Functional |>Concurrent |>Pragmatic |>Fun*. Ed. by The Pragmatic Bookshelf. 2018.

[15] *Uputstvo za instaliranje Elixir-a*. URL: https://elixir-lang.org/install.html.

[16] *Uvod u Mix*. URL: https://elixir-lang.org/getting-started/mix-otp/introduction-to-mix.html.

[17] *What are the best Elixir IDEs/editors?* URL: https://www.slant.co/topics/6921/~elixir-ides-editors.

[18] *What is DNA?* URL: https://ghr.nlm.nih.gov/primer/basics/dna.

[19] Daniel R. Zerbino and Ewan Birney. In: *Genome Research* (May 2008). URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2336801/.

[20] *Zvanična stranica kompanije Platformatec*. URL: http://plataformatec.com.br/.

[21] *Zvanična stranica programskog jezika Erlang*. URL: http://erlang.org.

[22] *Zvanična stranica programskog jezika Haskell*. URL: https://www.haskell.org/.

[23] *Zvanična stranica programskog jezika Python*. URL: https://www.python.org/.

[24] *Zvanična stranica PyCharm-a*. URL: https://www.jetbrains.com/pycharm/promo/?gclid=CjwKCAjw1_PqBRBIEiwA71rmtauyUPbqEGp8akacF4llZAV-AAJQsTXoTZGJOvnGYBwE.

[25] Rolf Zwart. In: (Apr. 2018). URL: https://reinout.vanrees.org/weblog/2018/04/25/origin-of-python-abc.html.