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

Naslov master rada: Jezik Elixir sa primenom u sekvencioniranju genoma

Rezime:

Ključne reči:

Sadržaj

1 Uvod **1**

2 Elixir **2**

2.1 Razvojno stablo . . . 3
2.2 Osnovne karakteristike . . . 8
2.3 Osnovni tipovi podataka . . . 9
2.4 Osnovni operatori . . . 19
2.5 Poklapanje obrazaca . . . 21
2.6 Nepromenljivost podataka . . . 23
2.7 Odlučivanje . . . 24
2.8 Moduli . . . 25
2.9 Direktive . . . 26

3 Sekvencioniranje genoma **28**

3.1 Istorija sekvencioniranja genoma . . . 31
3.2 *Shotgun* sekvencioniranje celokupnog genoma . . . 32
3.3 *De novo* sekvencioniranje genoma za kratka očitavanja . . . 36
3.4 Korekcija grešaka . . . 37
3.5 Konstrukcija kontiga . . . 41

4 Opis implementacije algoritama i rezultati **48**

4.1 JellyFish algoritam . . . 49
4.2 DSK algoritam . . . 52
4.3 De Brujinov asembler . . . 53

5 Zaključak **54**

Literatura **55**

Glava 1

Uvod

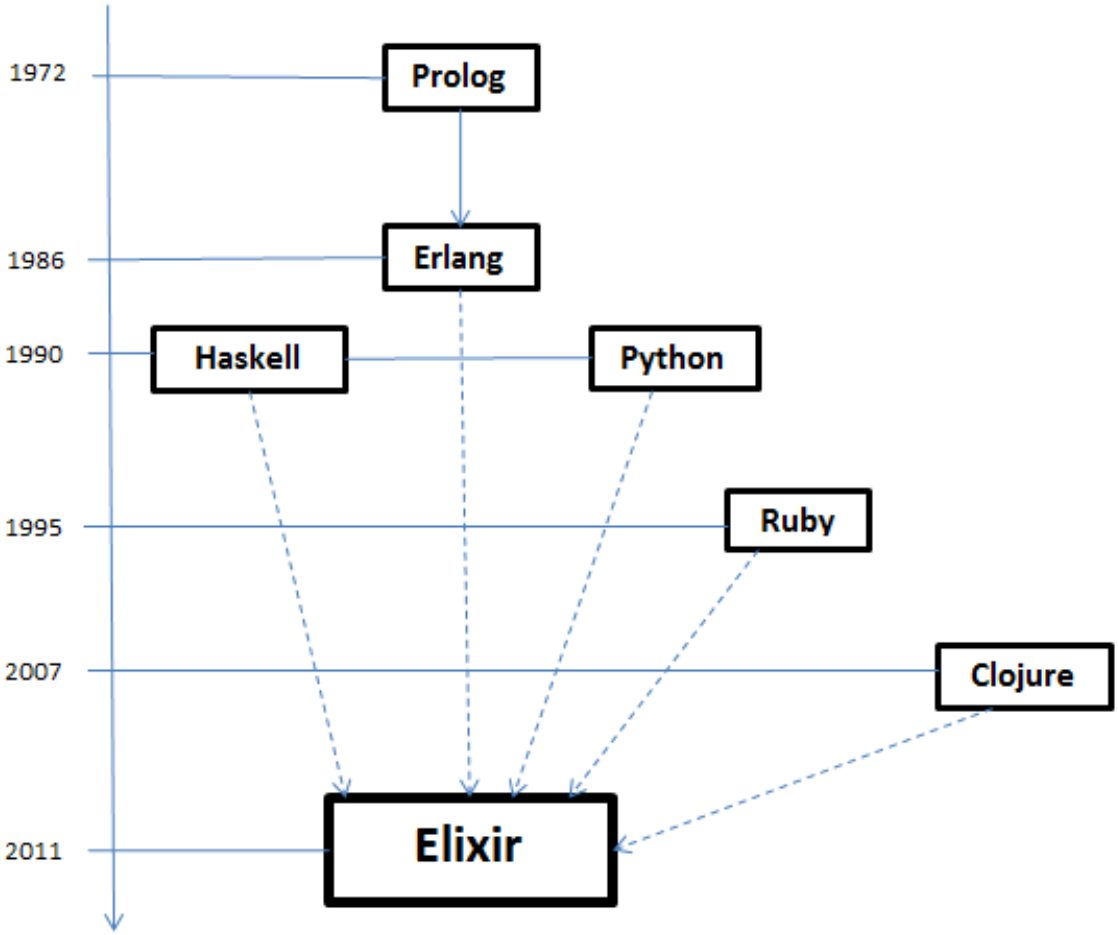
Glava 2

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, izuzetnog rada u distrubiranim sistemima i tolerancije na greške koja je na jako visokom nivou, u Elixir-u je rađeno mnogo zanimljivih projekata u vezi sa robotikom. 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.



Slika 2.1: Razvojno stablo jezika Elixir

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 [3]. Jedan od zadataka novonastale laboratorije bio je dodavanje konkurentnih procesa u programski jezik **Prolog** [1] i 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, neke delove sintakse i koncepte iz Prologa je zadržao (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, kao i **standardna biblioteka OTP** (engl. *The Open Telecom Platform*) [3]. 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 [1]. 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 [5], 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 nasledio 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 vec godinama i koja se pokazala pouzdanim rešenjem za skalabilne aplikacije.

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

Python

Python je interpretirani jezik opšte namene čiji tvorac je Gido van Rosum (engl. *Guido van Rossum*). Krajem 1980-ih je koncipiran kao naslednik jezika **ABC**, a prvi put je objavljen 1991. godine. Filozofija dizajna jezika Python naglašava čitljivost koda pridavajući veliki značaj razmaku. Njegove jezičke konstrukcije i objektno-orijentisani pristup imaju za cilj da pomognu programerima da napišu jasan i logičan kod za male i velike projekte. Python je dinamički tipiziran jezik i poseduje sistem za prikupljanje smeća (*garbage collector*). Podržava više paradigmi programiranja uključujući proceduralno, objektno-orijentisano i funkcionalno programiranje. Python interpreteri su dostupni za mnoge operativne sisteme. Globalana 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 i prilično različit od većine ostalih programskih jezika. Nazvan je po Haskel 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 kod, 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. Fokus kod ovog programskog jezika 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 kompjuteru 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).

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: Sintaksa jezika Ruby i Elixir

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 za višenitno programiranje. On je kompajlirani jezik, ali je i dalje potpuno dinamički tipiziran - svaka funkcija koju podržava Clojure je podržana u toku izvršava-

nja. 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 (nepromenljivost podataka), opcionalno lenjo izračunavanje i protokole.

[2]Lisp je programski 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.

2.2 Osnovne karakteristike

Hosé Valim je tokom 2010. godine bio zaposlen u kompaniji *Platformatec* [2] i radio 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 2 miliona *WebSocket* konekcija, dok je u 2017. za skalirani Elixir zabeležena obrada 5 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 sa 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.

2.3 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 2.3, predstavljaju prvu vrednost elementa liste i često se koriste da označe uspeh (*:ok*) ili grešku (*:error*).

Celi brojevi

Celi brojevi su slični 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.

Elixir koristi uglaste zagrade ([]) da označi listu vrednosti. Vredosti mogu biti bilo kog tipa, a primer liste sa vrednostima različitih tipova prikazan je na listingu 2.1.

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

Listing 2.1: Primer liste

Nadovezivanje ili oduzimanje 2 liste korišćenjem operatora $++/2$ i $--/2$ prikazano je na listingu 2.2.

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

Listing 2.2: Nadovezivanje i oduzimanje dve liste

Operatori liste 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 kod. 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 2.3. 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 2.3: 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 2.4.

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

Listing 2.4: Lista vrednosti pod jednostrukim navodnicima

Preuzimanje informacija o tipu neke vrednosti može se izvršiti pomoću funkcije *i/1* i može se videti na listingu 2.5.

```
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 2.5: 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 2.6.

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

Listing 2.6: 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 2.7.

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

Listing 2.7: 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. *File.read/1* je funkcija koja se može koristiti za čitanje sadržaja datoteke. 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 2.8.

```
iex(1)>File.read("C:\elixir\text_document.txt")
{:ok, "Hello, world!"}
```

Listing 2.8: Primer upotrebe funkcije *File.read/1*

Indeksi torke počinju od nule, a primer izdvajanje elementa sa indeksom 1 može se videti na listingu 2.9.

```
iex(1)>tuple = {:ok, "hello", 1}
{:ok, "hello", 1}
iex(2)>elem(tuple, 1)
"hello"
```

Listing 2.9: Izdvajanje elementa torke sa indeksom 1

Umetanje novog elementa na odredjeno mesto u torki vrši se pomoću funkcije *put_elem*/3. Ona vraća novu torku, dok originalna torka ostaje neizmenjena. Primer koda koji ilustruje upotrebu ove funkcije prikazan je na listingu 2.10.

```
iex(1)>tuple = {:ok, "hello", 1}
{:ok, "hello", 1}
iex(2)>put_elem(tuple, 1, "world")
{:ok, "world", 1}
iex(3)>tuple
{:ok, "hello", 1}
```

Listing 2.10: 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. Lista torki gde je prvi element torke atom (tj. ključ) u Elixir-u se naziva **lista ključnih reči**. Elixir podržava posebnu sintaksu za definisanje takvih lista: $[key : value]$. Primer oba načina definisanja prikazan je na listingu 2.11.

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

Listing 2.11: 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 2.12.

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

Listing 2.12: Dodavanje nove vrednosti listi ključnih reči

Elementima liste ključnih reči se pristupa na način prikazan na listingu 2.13.

```
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 2.13: Pristup elementu liste ključnih reči

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

1. Ključevi moraju biti atomi.
2. Ključevi su uredjeni, 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 vrmena 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 bilo kog 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 2.14.

```
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 2.14: Primer mape i pristupa njenim elementima

Modul **Map** obezbeđuje razne funkcije za manipulaciju mapama, a neke od njih mogu se videti na listingu 2.15.

```
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 2.15: Neke od funkcija modula Map

Mape imaju sintaksu za ažuriranje vrednosti ključa prikazanu na listingu 2.16

```
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 2.16: 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 2.17.

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

Listing 2.17: 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 2.18.

```
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 2.18: 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.

Na listingu 2.19 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)>users = [john: %{name: "John", age: 27, languages:
["Erlang", "Ruby", "Elixir"]}, mary: %{name: "Mary", age: 29,
languages: ["Elixir", "F#", "Clojure"]}]
[
  john:%{age:27,languages:["Erlang","Ruby","Elixir"],name:"John"},
  mary:%{age:29,languages:["Elixir","F#","Clojure"],name:"Mary"}
]
```

Listing 2.19: Struktura koja predstavlja listu korisnika

Pristup Džonovim godinama mogao bi se izvršiti na način prikazan na listingu 2.20.

```
iex(2)>users[:john].age
27
```

Listing 2.20: Pristup godinama od Džona

Ista sintaksa se može koristiti i za ažuriranje vrednosti i prikazana je na listingu 2.21.

```
iex(3)> users = put_in users[:john].age, 31
[
  john:%{age:31,languages:["Erlang","Ruby","Elixir"],name:"John"},
  mary:%{age:29,languages:["Elixir","F#","Clojure"],name:"Mary"}
]
```

Listing 2.21: Ažuriranje vrednosti

Makro *update_in*/2 je sličan, ali daje mogućnost prosleđivanja funkcije koja kontorliše kako se vrednost menja. Na primer, uklanjanje programskog jezika Clojure sa Marijinog spiska jezika može se uraditi na način prikazan listingom 2.22.

```
iex(4)> users = update_in users[:mary].languages,
fn languages -> List.delete(languages, "Clojure") end
[
  john:%{age:31,languages:["Erlang","Ruby","Elixir"],name:"John"},
  mary:%{age:29,languages:["Elixir","F#"],name:"Mary"}
]
```

Listing 2.22: Brisanje jezika iz liste

Funkcija *get_and_update_in* omogućava izvlačenje 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.

2.4 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 nesto što ima vrednost *true* ili *false* kao svoj prvi operand. Primer koda koji ilustruje ovu osobinu prikazan je na listingu 2.23.

```
1 iex(1)true and true
2 true
3 iex(2)>false or is_atom(:example)
4 true
```

Listing 2.23: Primer upotrebe logičkih operatora

Ukoliko kao prvi operand prosledimo nesto čija vrednost nije logička, rezultat je greška kao na listingu 2.24.

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

Listing 2.24: 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. Sve vrednosti osim **false** i **nil** će biti procenjene na *true*, što se može videti na primeru prikazanom listingom 2.25.

```
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
false
iex(7)>!nil
true
```

Listing 2.25: 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 2.26.

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

Listing 2.26: Poređenje tipova

2.5 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 $2 + 2 = 5$ bi rezultirao greškom datom na listingu 2.27.

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

Listing 2.27: 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 2.28 nije greška, već uspešno poklapanje obrazaca:

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

Listing 2.28: Uspešno poklapanje obrazaca

Slično, dva identična stringa sa obe strane znaka jednakosti će dati rezultat prikazan listingom 2.29.

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

Listing 2.29: Uspešno poklapanje obrazaca sa stringovima

Poklapanje obrazaca se može prikazati i na pimeru sa listama. Neka je data lista osoba koja je prikazana listingom 2.30.

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

Listing 2.30: Lista osoba

Neka prve tri osobe treba da budu zapamćene, dok četvrta osoba nije bitna. U te svrhe se može iskoristiti poklapanje obrazaca dato na listingu 2.31.

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

Listing 2.31: 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 2.32.

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

Listing 2.32: Iščitavanje sadržaja promenljivih

Mape su vrlo korisne kod poklapanja obrazaca. Kada se koristi u poklapanju obrazaca, mapa će se uvek podudarati sa poskupom date vrednosti kao što se može videti na listingu 2.33.

```
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 2.33: 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 2.34.

```
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 2.34: Upotreba promenljivih u mapama

2.6 Nepromenljivost podataka

U mnogim programskim jezicima je dozvoljeno dodeljivanje vrednosti promenljivoj, a zatim njeno menjanje tokom izvršavanja programa. Mogućnost da 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 izvršavanje 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 vredosti 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.

2.7 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 obezbedjuje **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 naredba 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 naredba može biti praćena naredbom else (unutar do..end bloka), koja se izvršava, ako je ločički izraz netačan.

3. *unless naredba* - Naredba unless ima isto telo kao i if naredba. Kod 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. Kod unutar unless..else naredbe se izvršava samo kada je navedeni uslov netačan.

5. *cond* - Naredba cond se koristi ukoliko treba izvršiti neki kod na osnovu nekoliko uslova. Radi kao if..else if..else kod drugih programskih jezika.

6. *case* - Naredba case se može smatrati zamenom za **switch** naredbu 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 kod povezan sa tim slučajem i izlazi iz case naredbe.

2.8 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 2.35.

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

Listing 2.35: Kreiranje modula i funkcija

Moduli mogu biti ugnježdeni u Elixir-u. Ova osobina jezika omogućava bolje organizovanje koda. Na listingu 2.36 se može videti definisanje dva modula: **Math** i **Math.Adding**, pri čemu je drugi ugnježden unutar prvog. Drugom se može pristupati samo pomoću *Adding* unutar *Math* modula sve dok su u istom leksičkom opsegu. Ako se kasnije *Adding* modul premesti izvan definicije *Math* modula, onda se mora referencirati njegovim punim imenom *Math.Adding* ili pseudonim mora biti postavljen pomoću direktive aliasa.

```
1 defmodule Math do
2   defmodule Adding do
3     def sum(a, b) do
4       a + b
5     end
6   end
7 end
```

Listing 2.36: Ugnježdavanje modula

U Elixir-u nema potrebe za definisanjem modula *Math*, kako bi se definisao modul *Math.Adding*, 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 *Math.Adding.Sum*, iako prethodno nije definisan modul *Math* i *Math.Adding*.

2.9 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 2.37.

```
1 #Alias modula tako da se moze pozvati sa Adding umesto sa
2 #Math.Adding
3 alias Math.Adding, as: Adding
4
5 #Obezbedjuje da je modul kompajliran i dostupan (obicno za makroe)
6 require Math
7
8 #Ukljucuje prilagodjen kod definisan u Math kao prosirenje
9 use Math
```

Listing 2.37: Primer upotrebe direktiva

Aliasi

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

Require

Elixir obezbeđuje makroe kao mehanizam za meta-programiranje (pisanje koda koji generiše kod). 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.

Import

Direktivu **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.

Use

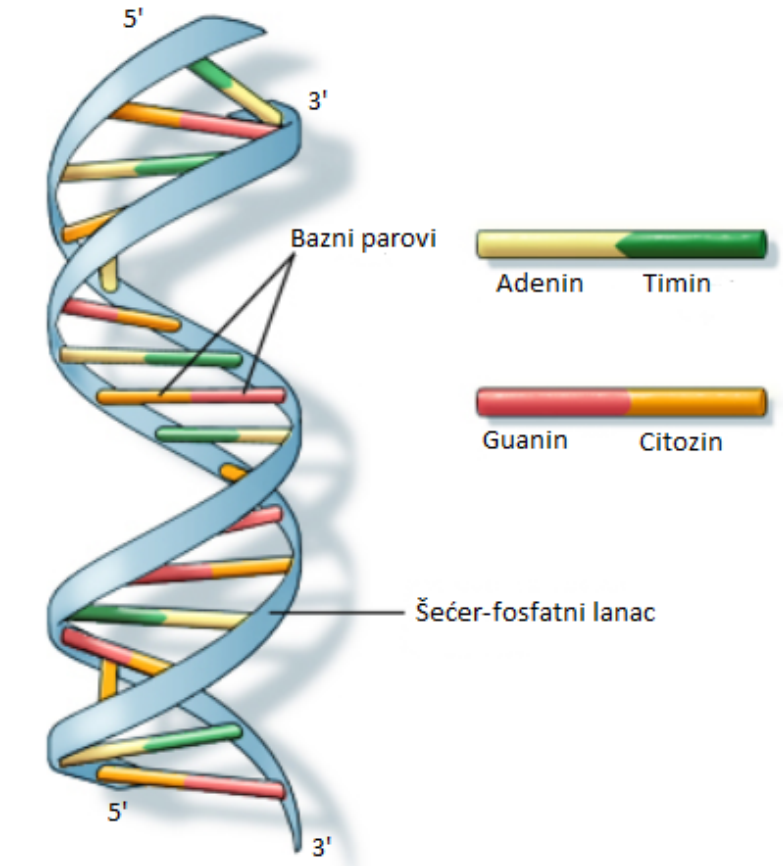
Iako nije direktiva, **use** je makro koji je usko povezan sa zahtevom koji omogućava korišćenje modula u trenutnom kontekstu. Makro use se često koristi za unos spoljne funkcionalnosti u trenutni leksički opseg, često modula.

Glava 3

Sekvencioniranje genoma

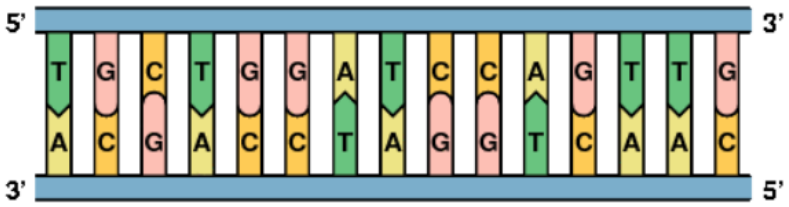
DNK (dezoksiribonukleinska kiselina) je nukleinska kiselina koja sadrži uputstva za razvoj i pravilno funkcionisanje svih živih organizama. Informacije u DNK se čuvaju kao kôd koji čine **četiri hemijske baze**: **adenin** (A), **guanin** (G), **citozin** (C) i **timin** (T). Ljudski DNK se sastoji od 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 imaju antiparalelnu[1] orijentaciju i koji čine spiralu koja se naziva **dvostruka spirala** [14]. 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):

[1]U biohemiji, dva molekula su antiparalelna ako su locirani jedan pored drugog i usmereni u suprotnim pravcima, ili ako su dva lanca međusobno komplementarna.



Slika 3.1: Struktura DNK [14]

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):



Slika 3.2: Odmotan DNK [4]

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 [4].

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 koja je sa računarske strane niska nad azbukom $\{A, C, G, T\}$. Svaki gen predstavlja 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.

Sekvenciranje genoma podrazumeva otkrivanje sastava genoma. U pitanju je eksperimentalan proces – 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 koje određuju redosled nukleotida nazivaju se **sekvenceri**. Sekvencioniranje DNK uključuje bilo koji metod ili tehnologiju koja se koristi za određivanje redosleda četiri baze iz našeg genoma (A, C, G, T) [8]. Savremene laboratorijske metode za dati uzorak krvi pacijenta 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**. Genomi i 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. Što su delovi veći, slagalicu je lakše sastaviti.

3.1 Istorija sekvencioniranja genoma

Sanger sekvencioniranje 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 Sanger sekvencioniranja: *OLC* [2] asembler **Celera** i asembler **Ojler** zasnovan na De Brujinovim grafovima. **Humani referentni genom** sastavljen je korišćenjem ova dva pristupa. Humani referentni genom je digitalna baza podataka o nukleinskim kiselinama, koju su naučnici prikupili kao reprezentativni primer skupa gena čoveka. 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 Bafala (engl. *Buffalo*). 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 Sanger sekvencioniranje 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 **panda** genoma. 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.

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

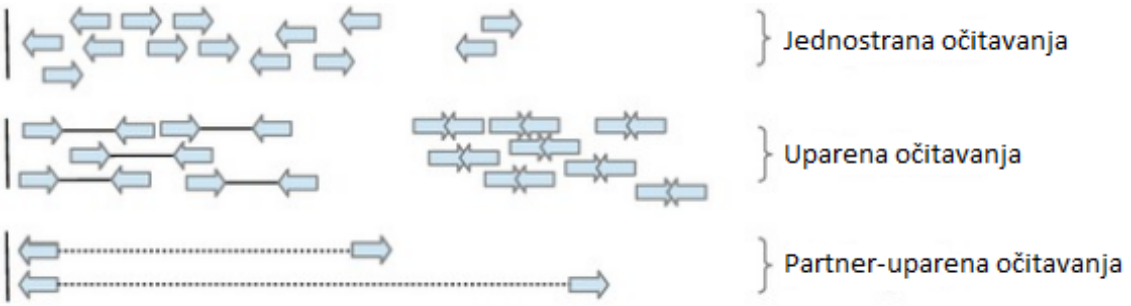
[3] *De novo* asembleri su programi koji vrše sklapanje tako što proširuju kratka očitavanja spajanjem susednih očitavanja u dužu sekvencu, bez korišćenja referentne sekvence.

3.2 *Shotgun* sekvencioniranje celokupnog genoma

Jedna od tehnika druge generacije sekvencioniranja je *shotgun* sekvencioniranje 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*)
- partner-uparena očitavanja (engl. *mate-pair reads*)

Prikaz ovih vrsta očitavanja može se videti na slici 3.3.



Slika 3.3: Vrste očitavanja

[6] str. 1030, slika 2

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 broj nukleotida u genomu, doći će do značajnog broja grešaka u sekvencioniranju. Pored toga, mnoge 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 *shotgun* sekvencioniranja celokupnog genoma postoje dva protokola – **sekvencioniranje celokupnog genoma** i **sekvencioniranje partner-uparenih očitavanja**.

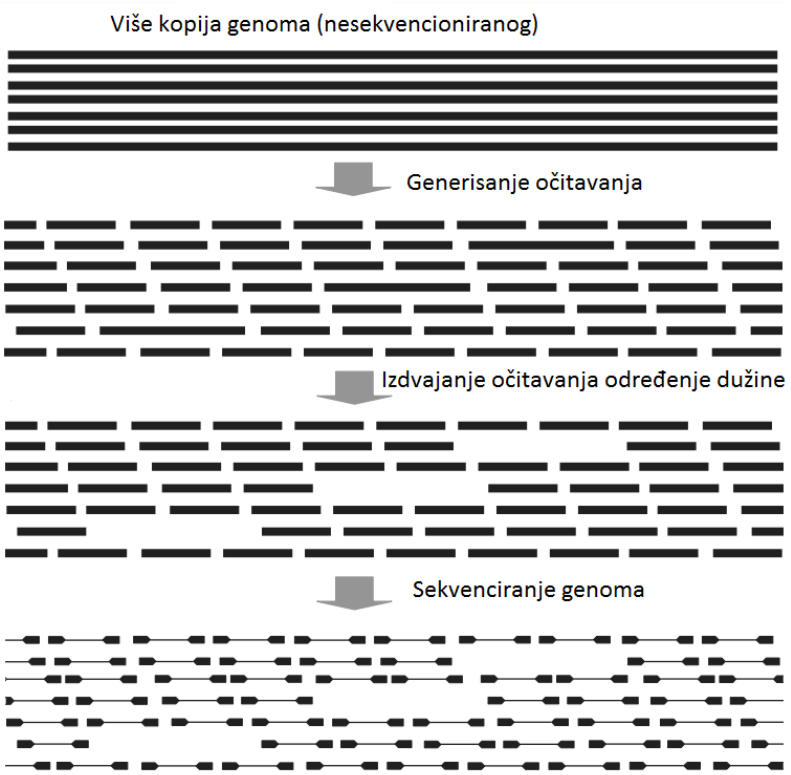
[4] Pojava zamene mesta jednog nukleotida nekim drugim nukleotidom.

Sekvencioniranje celokupnog genoma

Sekvencioniranje celokupnog genoma uključuje tri koraka:

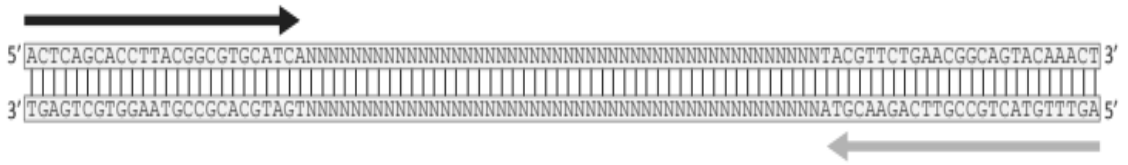
- razbijanje genoma
- izbor dužine očitavanja i izdvajanja fragmenata te dužine
- sekvencioniranje očitavanja

Prvo se uzorak genoma na slučajan način razbija na DNK fragmente. Zatim sledi korak odabira dužine očitavanja u kom se vrši ekstrahovanje DNK fragmenata te dužine. Na kraju se vrši 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: *Shotgun* sekvencioniranje celokupnog genoma [12]

Na slici 3.5 prikazan je odmotan i rotiran DNK fragment ekstrahovan iz uzorka genoma:



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

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, pri čemu je drugo očitavanje inverzni komplement prvog, odnosno:

- $ACTCAGCACCTTACGGCGTGCATCA$
- $AGTTTGTACTGCCGTTCAGAACGTA$

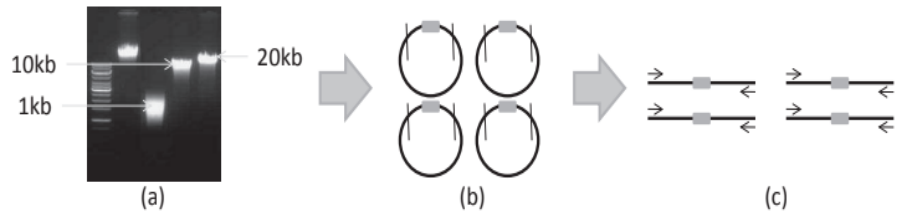
Za asembliranje genoma, dužina očitavanja je veoma važan parametar. Različite tehnologije sekvencioniranja imaju različita ograničenja na dužinu očitavanja. Na primer, *Illumina Hi-seq* [5] sekvencioniranje može izvršiti samo sekvencioniranje uparenih očitavanja dužine manje od 1000 bp. Za treću generaciju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

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

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 možemo 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)).



Slika 3.6: Sekvencioniranje partner-uparenih očitavanja [12]

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:

- 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*).

[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.

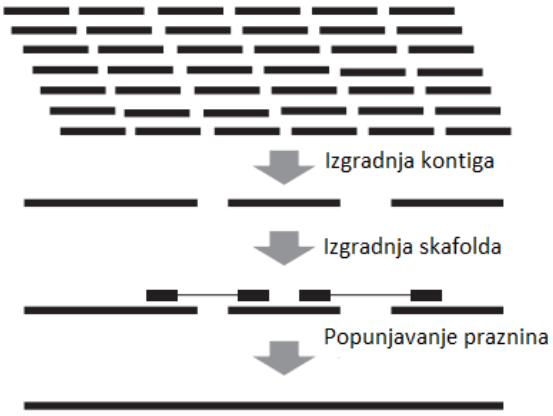
3.3 *De novo* sekvencioniranje genoma za kratka očitavanja

Druga generacija tehnika sekvencioniranja omogućava dobijanje skupa jednostranih ili uparenih kratkih očitavanja celokupnog genoma. *De novo* sekvencioniranje 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 [7]. 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 superstring $ACATGCGTGTACGT$).

Mnogi *de novo* asembleri predlažu asembliranje kratkih očitavanja. Opšte rešenje uključuje četiri koraka. U prvom koraku se koriguju greške sekvencioniranja u očitavanjima. Na osnovu korigovanih očitavanja, u drugom koraku se vrši spajanje očitavanja preklapanjem. 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 preklapanjem očitavanja grade kontinuiranu sekvencu, takozvanu **kontigu** (engl. *contig*). Kontige obično predstavljaju jednu **konsenzus nisku** [7]. 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.

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



Slika 3.7: *De novo* sekvencioniranje 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 on je nastao kao posledica grešaka prilikom sekvencioniranja očitavanja i verovatno se neće naći 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 sekvencioniranju. Ove razlike se mogu iskoristiti za filtriranje grešaka u k-merima i selektivno uklanjanje očitavanja koja sadrže greške sekvencioniranja iz skupa podataka [12]. Greške u sekvencioniranju očitavanja mogu zbuniti *de novo* asemblere. Da bi se to izbeglo, vrši se korigovanje tih grešaka pre početka asembliranja genoma.

Brojanje k-mera

Jedan konceptualno jednostavan, ali osnovni problem je **brojanje k-mera**. To je potprogram koji se koristi u korekciji grešaka u očitavanjima. Može biti korišćen i u koraku asembliranja, detekciji ponovaka i kompresiji genomskih podataka. Ima značajnu ulogu u utvrđivanju da li je došlo do grešaka u sekvencioniranju ili je u pitanju jednonukleotidni polimorfizam. Ulaz u ovaj potprogram je skup očitavanja R i parametar k. Neka je Z skup svih mogućih k-mera koji se pojavljuju u R. Problem je izračunavanje frekvencije pojavljivanja k-mera u Z. U nastavku će biti razmatrana 4 rešenja: **(1) jednostavno heširanje, (2) JellyFish, (3) BFCounter i (4) DSK.**

Jednostavno heširanje

Problem brojanja k-mera može biti rešen implementacijom asocijativnog niza koristeći **heširanje**. Heširanje je tehnika kojom se vrsi 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. Svaki k-mer z može biti kodiran kao 2k-bitni binarni ceo broj $b(z)$ zamenom A, C, G i T u z sa 00, 01, 10, 11, respektivno. Tako se izgrađuje tabela $Count[0..4^k - 1]$ veličine 4^k u kojoj svaka ulazna vrednost $Count[b(z)]$ čuva frekvenciju k-mera z u skupu Z. Preciznije, prvo se vrši inicijalizacija svake ulazne vrednosti u $Count[0..4^k - 1]$ na 0. Zatim se iterativno skenira svaki k-mer z iz Z i uvećava $Count[b(z)]$ za 1. Na kraju, sve ulazne vrednosti različite od nule u $Count[]$ predstavljaju k-mere koji se pojavljuju u Z kao i broj njihovih pojavljivanja.

Neka je $N = |Z|$. Tada je navedeni pristup veoma efikasan. Njegovo vreme izvršavanja je $O(N+4^k)$, a kako treba izgraditi tabelu veličine 4^k, prostorna složenost je $O(4^k)$. Kada je k veliko, navedeni algoritam ne može da radi, jer zahteva previše prostora.

JellyFish algoritam

Moguće je smanjiti veličinu heš tabele 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 **duplo heširanje**.

Ovaj mehanizam je iskorišćen u *JellyFish* algoritmu. Uvodi se heš funkcija $h()$. Heš tabela $H[0..\frac{N}{\alpha} - 1]$ čuva niz k-mera iz Z, gde je α **faktor opterećenja** [9] $(0 < \alpha \leq 1)$. Svaki k-mer iz Z je heširan u neku vrednost $H[i]$ gde je $i = h(z)$. Izgradjuje se tabela $Count[0..\frac{N}{\alpha} - 1]$, pri čemu $Count[i]$ čuva broj pojavljivanja za k-mer

[8] Izraz koji potiče od latinske reči *collisio* i znači sudar, sukob. U ovom kontekstu, moguće je da heš funkcija za dva razlicita k-mera da istu vrednost, te kažemo da su ta 2 k-mera u koliziji, tj. sukobu.

[9] Faktor opterećenja - broj koji kontroliše veličinu heš tabele.

$H[i]$. Ukoliko pri heširanju funcijom $h()$ dodje do kolizije, ona se razrešava mehanizmom otvorenog adresiranja. Na ovaj način se pokušava smanjivanje prostora koji je potreban za obavljanje brojanja k-mera.

Iako $JellyFish$ algoritam koristi manje prostora od metode naivnog prebrojavanja, $JellyFish$ heš tabela mora biti veličine koja je jednaka bar broju jedinstvenih k-mera iz Z. $JellyFish$ i dalje zahteva mnogo memorije u slučaju da je broj jedinstvenih k-mera u Z veliki.

BFCounter algoritam

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 da odredi da li se k-mer pojavljuje najmanje q puta. To je 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.

Vreme izvršavanja $BFCounter$ algoritma je $O(n)$. Što se tiče prostorne složenosti, *counting Bloom* filter zahteva $O(Nlog(q))$ prostora. Prostor za $H[]$ i $Count[]$ je $\frac{N'}{\alpha}(2k + 32)$ bitova, gde je N' broj k-mera koji se pojavljuju najmanje q puta. Primetimo da je $N' \leq \frac{N}{q}$.

DSK algoritam

Iako je $BFCounter$ prostorno efikasan, njegova prostorna složenost i dalje zavisi od broja N k-mera u Z. Neka je memorija fiksirana tako da bude M bitova i neka je memorija diska fiksirana tako da bude D bitova. Da li se može i dalje efikasno izračunati pojavljivanja k-mera? Gijom Rizk (engl. *Guillaume Rizk*) nam daje pozitivan odgovor i predlaže metod koji se naziva **DSK**.

Ideja ovog metoda je da se skup k-mera Z podeli u različite liste tako da svaka lista bude smestena 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 $JellyFish$.

Ovaj algoritam će zapisati samo jednom svaki k-mer iz Z, iako će svaki k-mer pročitati n_{list} puta. Stoga, on 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 $\frac{D}{2k}'s$ k-mera koji pripadaju i-toj listi i zapisuje ih na disk, što oduzima $O(\frac{D}{k})$ vremena. Nakon toga, algoritam čita $\frac{D}{2k}'s$ 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. Kad je $D = \theta(N)$, algoritam se izvršava u $O(kN)$ očekivanom vremenu.

3.5 Konstrukcija kontiga

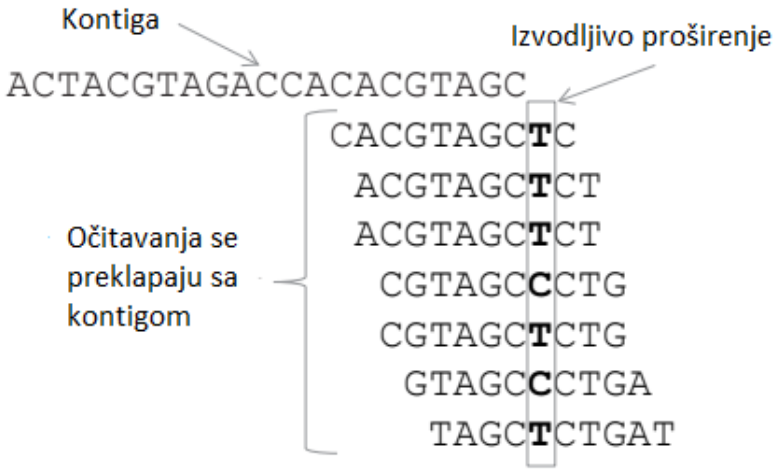
Nakon što su sva očitavanja korigovana, može se vršiti njihovo spajanje radi formiranja kontiga. Postoje dva pristupa za konstrukciju kontiga:

- pristup baznog proširenja (*base-by-base* pristup)
- De Brujinov 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 **konsenzusna 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.



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.

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

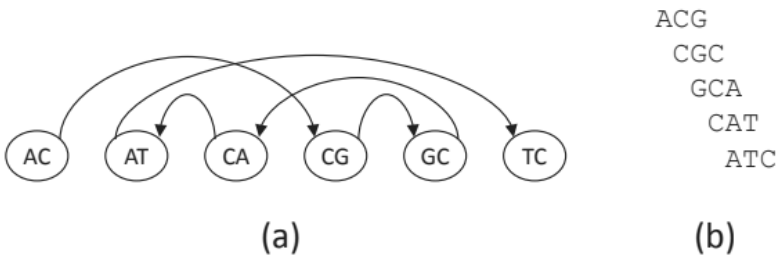
Iako je bazno proširenje jednostavno, ono često daje kratke kontige zbog dva problema. Prvo, početni šablon je nasumično izabrano očitavanje. Ukoliko ono sadrži greške sekvencioniranja ili se nalazi u ponovku, to će uticati na proširenje. Drugi problem je što se može desiti da se šablon proširi u neki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

De Brujinov grafovski pristup

De Brujinov grafovski pristup je zasnovan na De Brujinovim grafovima. Ovaj pristup su uveli Induri (engl. *Indury*) i Voterman (engl. *Waterman*) u njihovoj knjizi *„A new algorithm for DNK sequence assembly: Journal of Computational Biology"*. On je danas glavni pristup za asembliranje kratkih očitavanja.

Neka je dat skup očitavanja R i parametar k. De Brujinov 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 Brujinov graf može biti konstruisan u $O(N)$ vremenu.

Na slici 3.9(a) je prikazan De Brujinov graf H_3 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_3 je prikazano na slici 3.9(b).

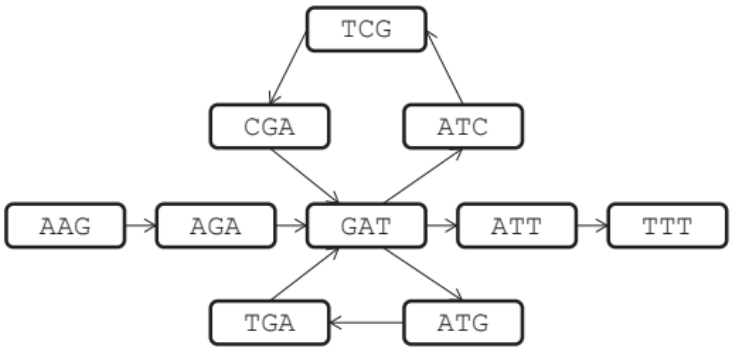


Slika 3.9: Konstrukcija De Brujinovog grafa [12]

Uzorak genoma se može predvideti 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 pre-

klapanjem očitavanja i dobijen na osnovu De Brujinovog grafa za skup R. Medjutim, 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 Brujinovom 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 Brujinov 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 mozda neće uvek dati ispravnu sekvencu. Čak još gore, Ojlerova putanja možda neće postojati u nekom grafu H_k.

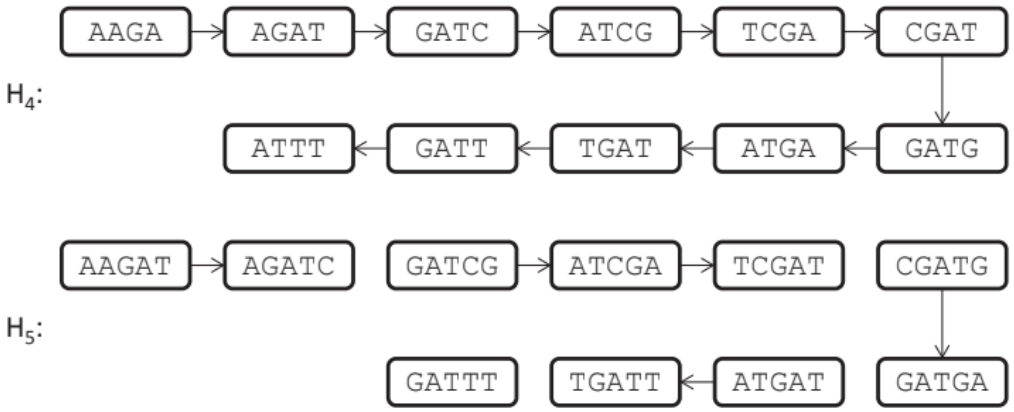
U nastavku će se govoriti o De Brujinovom grafovskom asembleru u slučaju kada:

- ne postoji greška sekvencioniranja
- postoji greška sekvencioniranja

De Brujinov 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. Kontiga je maksimalna prosta putanja u H_k. Preciznije, svaka maksimalna prosta putanja je maksimalna putanja u H_k tako da svaki čvor (osim početnog i krajnjeg) ima unutrašnji i spoljašnji stepen jedan.

Za De Brujinov 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 Brujinovom grafu za skup $R = \{AAGATC, GATCGAT, CGATGA, ATGATT, GATTT\}$ u slučajevima kada je $k = 4$ i $k = 5$.



Slika 3.11: De Brujinovi 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 je prosta putanja i može se konstruisati pet kontiga: $AAGATC$, $GATCGAT$, $CGATGA$, $ATGATT$, $GATTT$.

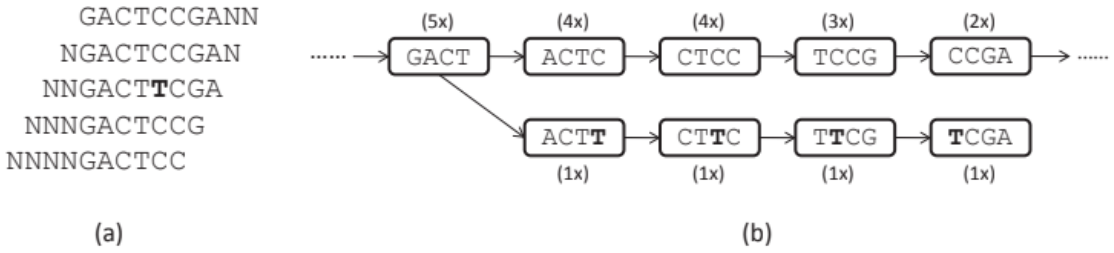
De Brujinov graf je dobar, ukoliko se zna ispravno k. Kada je k malo (pogledati H_3 na slici 3.10), postoji veliki broj grana zbog ponovaka i rezultat je veliki broj kratkih kontiga. Kada je k veliko (pogledati H_5 na slici 3.11), neki k-meri nedostaju, što rezultuje nepovezanim komponentama, koje takođe generišu veliki broj kratkih kontiga. Tako da se mora identifikovati ispravno k kako bi se pronašla ravnoteža između ova dva problema.

De Brujinov asembler sa greškama sekvencioniranja

Kod De Brujinovog asemblera bez grešaka sekvencioniranja pretpostavka je bila da ne postoje greške sekvencioniranja u očitavanjima, što je nerealno. Kada postoje greške, pokušava se njihovo uklanjanje „čišćenjem" De Brujinovog grafa. Rešenje koje će se razmatrati su predložili Zerbino (engl. *Zerbino*) i Birni (enlg. *Birney*). Kratka očitavanja imaju nisku stopu greške (sadrže jednu greška na svakih 100 baza), a većina k-mera sadrži najviše jednu grešku. Ovakvi k-meri sa greškom mogu kreirati dva moguća anomalna podgrafa u De Brujinovom grafu: **vrh (špic)** i **mehurić**.

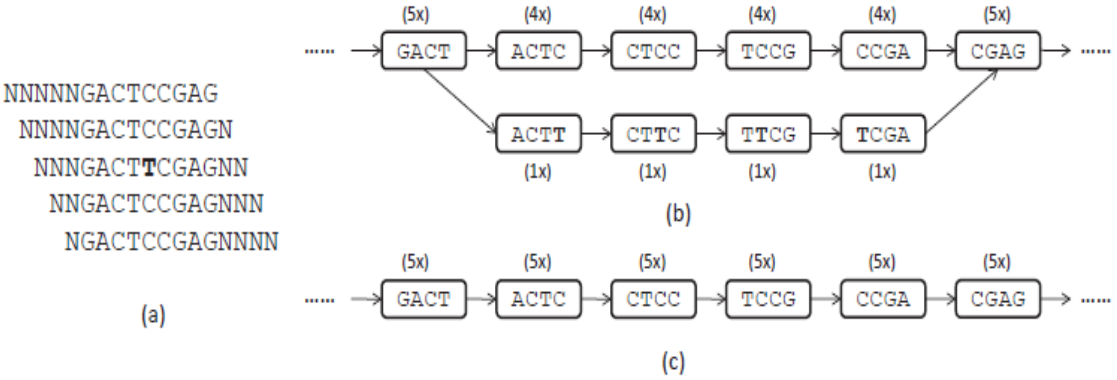
Vrh je putanja dužine najviše k gde svi unutrašnji čvorovi imaju ulazni i izlazni stepen 1, dok jedan od njihovih krajeva ima ulazni ili izlazni stepen 0. To može proizvesti potencijalnu kontigu dužine najviše $2k$.

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 Brujinov 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 Brujinovog grafa, ali uklanjanje jednog vrha može generisati nove vrhove. Tako da procedura uklanjanja mora uklanjati vrhove rekurzivno.



Slika 3.12: Formiranje vrha [12]

Mehurić se sastoji od dve različite kratke putanje sa istim početnim i krajnim čvorovima u De Brujinovom 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 Brujinov graf i jedan mehurić koji se formira.



Slika 3.13: Formiranje mehurića [12]

Gornja putanja mehurića predstavlja *GACTCCGAG*, dok donja putanja predstavlja *GACTTCGAG*. Kada su dve putanje u mehuriću veoma slične, putanja sa

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. Onda, kada se dve putanje spoje u mehuriću, zadržava se ona sa većom težinom. Za spajanje mehurića, može se koristiti **algoritam turneje** (engl. *tourbus algorithm*) koji će biti detaljno objašnjen u poglavlju .

Nakon što se uklone vrhovi i spoje mehurići u De Brujinovom grafu, može se dalje „čistiti" graf 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 koji može da obradi greške sekvencioniranja i njegov psoudokod je prikazan na slici 3.14.

Algorithm Velvet$(\mathcal{R}, k)$
Require: $\mathcal{R}$ is a set of reads and k is de Bruijn graph parameter
Ensure: A set of contigs
1: Generate the de Bruijn graph H_k for $\mathcal{R}$;
2: Remove tips;
3: Merge bubbles;
4: Remove nodes with multiplicity $\leq m$;
5: Extract all maximal simple paths in H_k as contigs;

Slika 3.14: Algoritam za otklanjanje grešaka sekvencioniranja [12]

Kako izabrati k?

Kao što je već rečeno, odabir broja k može uticati na performanse De Brujinovog algoritma. Jednostavno rešenje je pokretanje algoritma sa slike 3.14 za različite k. Zatim se vrši grupisanje i spajanje kontiga. Jedan od problema sa ovim jednostavnim rešenjem 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 traba graditi De Brujinov graf nezavisno za različite k. Umesto toga, *IDBA* gradi De Brujinov 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 Brujinov graf H_k za sve veće k. Kako su očitavanja u R korigovana, to je graf H_k „očišćen". Na ovaj način se za veliko k omogućava dobijanje dugih kontiga visokog kvaliteta iz H_k. Na slici 3.15 se nalazi pseudokod koji opisuje ideju IDBA asemblera:

Algorithm IDBA$(\mathcal{R}, k_{min}, k_{max})$
Require: $\mathcal{R}$ is a set of reads and k_{min} and k_{max} are de Bruijn graph parameter
Ensure: A set of contigs
1: **for** $k = k_{min}$ to k_{max} **do**
2: Generate the de Bruijn graph H_k for $\mathcal{R}$;
3: Remove tips;
4: Merge bubbles;
5: Remove nodes with multiplicity $\leq m$;
6: Extract all maximal simple paths in H_k as contigs;
7: All reads in $\mathcal{R}$ are aligned to the computed contigs;
8: The mismatch in the read is corrected if 80% of reads aligned to the same position has the correct base;
9: **end for**
10: Extract all maximal simple paths in $H_{k_{max}}$ as contigs;

Slika 3.15: IDBA [12]

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

Glava 4

Opis implementacije algoritama i rezultati

U prethodnom delu data je biološka osnova za algoritme koji se primenjuju u sekvencioniranju genoma. U ovom delu će biti dati opisi implementacije tih algoritama u programskom jeziku Elixir, kao i delovi koda radi boljeg razumevanja. Svaki algoritam je definisan u okviru modula koji nosi naziv tog algoritma i koji sadrži glavnu funkciju i veliki broj pomoćnih.

4.1 JellyFish algoritam

Kao što je već pomenuto u prethodnoj glavi, *JellyFish* algoritam koristi se za brojanje k-mera, a kao potprogram programa za korigovanje grešaka u sekvencioniranju očitavanja. Pseudokod ovog algoritma dat je na slici 4.1.

Algorithm Jellyfish(Z, α, h)
Require: Z is a set of N's k-mers, α is a load factor that controls the hash table size and $h(.)$ is the hash function
Ensure: The count of every k-mer appearing in Z
1: Set $H[1..\frac{N}{\alpha}]$ be a table where each entry requires $2k$ bits
2: Set $Count[1..\frac{N}{\alpha}]$ be a table where each entry requires 32 bits
3: Initialize T to be an empty table
4: **for** each k-mer z in Z **do**
5: $i = hashEntry(z, h, \frac{N}{\alpha})$;
6: **if** $H[i]$ is empty **then**
7: $H[i] = z$ and $Count[i] = 1$;
8: **else**
9: $Count[i] = Count[i] + 1$;
10: **end if**
11: **end for**
12: Output $(H[i], Count[i])$ for all non-empty entries $H[i]$;

Slika 4.1: Jellyfish algoritam

[12] str. 132, slika 5.9

Neka je $h()$ heš funkcija i $H[0..\frac{N}{\alpha} - 1]$ heš tabela koja čuva niz k-mera, gde je 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 iz Z vrši se njegovo heširanje u neku vrednost $H[i]$ gde je $i = h(z)$. Ako $H[i]$ nije prazan i $H[i] \neq z$, z se ne može čuvati u $H[i]$, tj. došlo je do kolizije. Ona može biti razrešena pomoću mehanizma otvorenog adresiranja. Na primer, kolizija se može razrešiti **linearnim popunjavanjem**. Ovom metodom pokušavamo da uvećamo indeks i za 1 kada se kolizija dogodi sve dok je $H[i] = z$ ili je ulaz $H[i]$ prazan.

Funkcija $hashEntry()$ sa slike 4.2 ilustruje šemu linearnog popunjavanja za razrešavanje kolizije. Ako $hashEntry(z, h, \frac{N}{\alpha})$ vraća prazan ulaz $H[i]$, onda z ne postoji u heš tabeli i postavljamo $H[i] = z$ i $Count[i] = 1$. U suprotnom, ako $hashEntry(z, h, \frac{N}{\alpha})$ vraća ulaz $H[i] = z$, uvećavamo $Count[i]$ za jedan. Nakon sto su svi k-meri iz Z obrađeni, prikazujemo $(H[i], Count[i])$ za sve ulaze $H[i]$ različite od nule.

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

Slika 4.2: Funkcija hashEntry

[12] str. 132, slika 5.9

JellyFish algoritam je efikasniji, 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.

Na listingu 4.1 se može videti konkretan primer izvršavanja algoritma u komandnom promptu:

```
C:\elixir>iex jellyfish.ex
Interactive Elixir (1.8.0) - press Ctrl+C to exit (type h() ENTER
    for help)
iex(1)>JellyFish.jellyfish_algorithm(["AC", "CG", "AC", "GT", "CA",
     "GG", "AC","GT"], 0.7)
{%{
  0 => "",
  1 => "",
  2 => "",
  3 => "CG",
  4 => "CA",
  5 => "",
  6 => "",
  7 => "GG",
  8 => "",
  9 => "GT",
  10 => "AC",
  11 => ""
},
%{
  0 => 0,
  1 => 0,
  2 => 0,
  3 => 1,
```

```
23    4 => 1,
24    5 => 0,
25    6 => 0,
26    7 => 1,
27    8 => 0,
28    9 => 2,
29    10 => 3,
30    11 => 0
31 }}
```

Listing 4.1: Primer pokretanja *JellyFish* algoritma

Za kompajliranje fajla u kome se nalazi algoritam i pokretanje interaktivnog Elixir-a, navodi se komanda *iex jellyfish.ex*. Nakon toga mogu se pozivati funkcije tako što se prvo navede ime modula u okviru koga su definisane, tačka, pa naziv funkcije i njeni argumenti. U ovom slučaju to su modul *JellyFish*, funkcija *jellyfish_ algorithm*, lista stringova [”*AC*”,”*CG*”,”*AC*”,”*GT*”,”*CA*”,”*GG*”,”*AC*”, ”*GT*”] koji predstavljaju k-mere i faktor opterećenja 0.7.

4.2 DSK algoritam

DSK je još jedan algoritam za brojanje k-mera, ali vremenski i prostorno efikasniji od *Jellyfish* algoritma. Njegov pseudokod dat je na slici 4.3.

Algorithm DSK(Z, M, D, h)
Require: Z is a set of N's k-mers, target memory usage M (bits), target disk space D (bits) and hash function $h(.)$
Ensure: The count of every k-mer appearing in Z

```
1: n_list = 2kN/D;
2: n_sublist = D(2k+32)/(0.7(2k)M);
3: for i = 0 to n_list − 1 do
4:    Initialize a set of empty sublists {d_0, ..., d_{n_sublist−1}} in disk;
5:    for each k-mer z in Z do
6:       if h(z) mod n_list = i then
7:          j = (h(z)/n_list) mod n_sublist;
8:          Write z to disk in the sublist d_j;
9:       end if
10:   end for
11:   for j = 0 to n_sublist − 1 do
12:      Load the jth sublist d_j in memory;
13:      Run Jellyfish(d_j, 0.7, h) (see Figure 5.9) to output the number of
         occurrences of every k-mer in the sublist d_j;
14:   end for
15: end for
```

Slika 4.3: DSK algoritam

[12] str. 136, slika 5.11

Neka je dat skup k-mera Z, broj bitova D na disku za čuvanje svake liste u koju se smeštaju k-meri, a broj bitova M u memoriji za čuvanje svake podliste k-mera i heš funkcija $h()$. Prvo, k-meri u Z su podeljeni 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 imamo N k-mera u Z, postavlja se $n_{list} = \frac{N}{n_{list}} = \frac{2kN}{D}$. Ovo deljenje se obavlja heš funkcijom $h()$ koja ravnomerno mapira sve k-mere u n_{list} lista. Preciznije, za svaki k-mer z iz 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 jednak $n_{sublist} = \frac{n_{list}}{n_{sublist}} = \frac{D(2k+32)}{0.7(2k)M}$. Slično, svaka lista je podeljena u

podliste heš funkcijom $h()$. Preciznije, za svaki k-mer s u i-toj listi, s je dodeljeno 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, brojimo pojavljivanja svakog k-mera u podlisti koristeći $JellyFish(d_j, 0.7, h)$ sa slike 4.2.

4.3 De Brujinov asembler

De Brujinov grafovski pristup je jedan od pristupa u izgradnji kontiga koji se danas najčešće koristi. Cilj ovog algoritma je da pronadje sve kontige određene veličine, a uz pomoć De Brujinovog grafa. Slika 4.4 daje pseudokod ovog jednostavnog metoda:

Algorithm De_Bruijn_Assembler$(\mathcal{R}, k)$
Require: $\mathcal{R}$ is a set of reads and k is de Bruijn graph parameter
Ensure: A set of contigs
1: Generate the de Bruijn graph H_k for $\mathcal{R}$;
2: Extract all maximal simple paths in H_k as contigs;

Slika 4.4: Jednostavan De Brujinov grafovski asmbler

[12] str. 141, slika 5.17

Neka je dat skup očitavanja R i parametar k. Izgrađuje se De Brujinov graf H_k, a zatim se pronalaze sve maksimalne proste putanje i od njih se formiraju kontige.

Glava 5

Zaključak

Literatura

[1] URL: http://erlang.org.

[2] URL: http://plataformatec.com.br/.

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

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

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

[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] 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.

[8] Mahmoud Parsian. English. Ed. by O'Reilly Media. 2015, pp. 486–502.

[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. The Pragmatic Bookshelf*. 2016.

[14] *What is DNA?* URL: https://ghr.nlm.nih.gov/primer/basics/dna (posećeno).

Biografija autora

Vuk Stefanović Karadžić (*Tršić, 26. oktobar/6. novembar 1787. — Beč, 7. februar 1864.*) bio je srpski filolog, reformator srpskog jezika, sakupljač narodnih umotvorina i pisac prvog rečnika srpskog jezika. Vuk je najznačajnija ličnost srpske književnosti prve polovine XIX veka. Stekao je i nekoliko počasnih mastera. Učestvovao je u Prvom srpskom ustanku kao pisar i činovnik u Negotinskoj krajini, a nakon sloma ustanka preselio se u Beč, 1813. godine. Tu je upoznao Jerneja Kopitara, cenzora slovenskih knjiga, na čiji je podsticaj krenuo u prikupljanje srpskih narodnih pesama, reformu ćirilice i borbu za uvođenje narodnog jezika u srpsku književnost. Vukovim reformama u srpski jezik je uveden fonetski pravopis, a srpski jezik je potisnuo slavenosrpski jezik koji je u to vreme bio jezik obrazovanih ljudi. Tako se kao najvažnije godine Vukove reforme ističu 1818., 1836., 1839., 1847. i 1852.