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

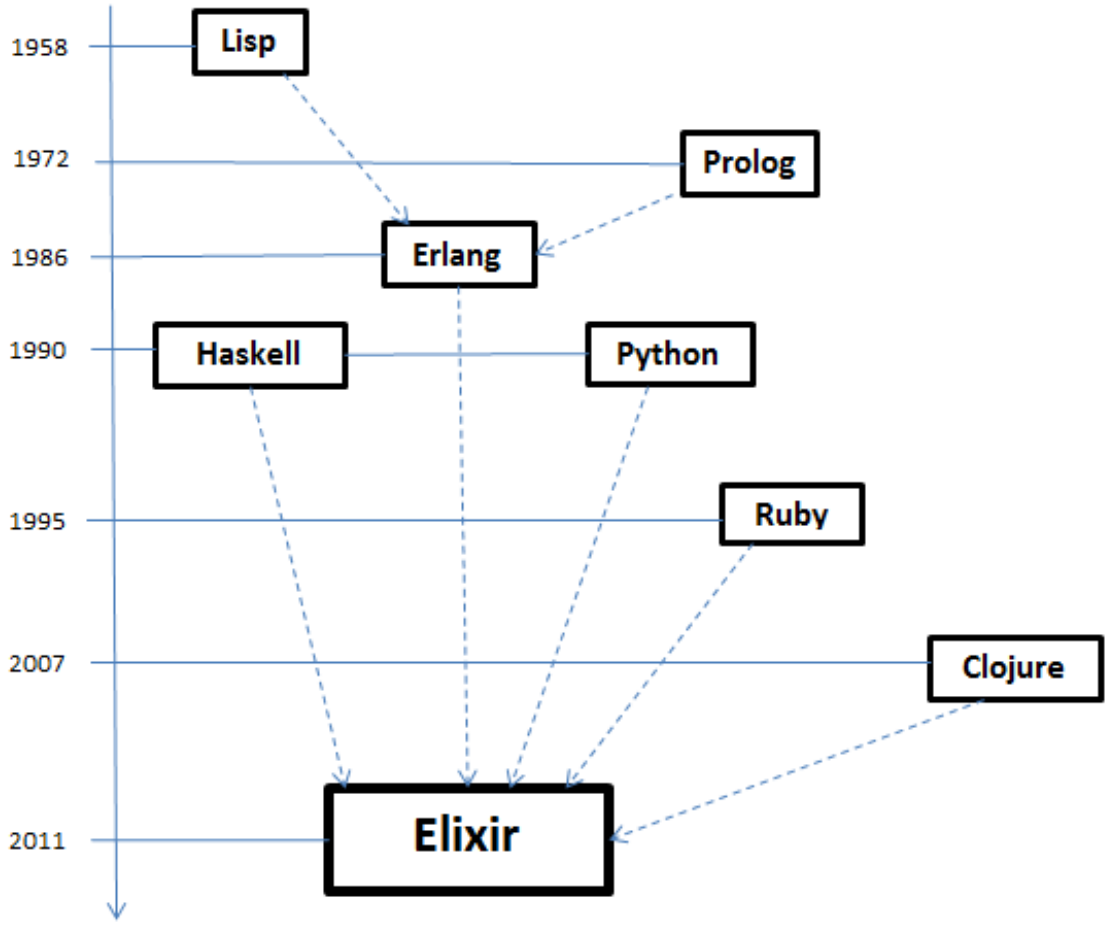
Glava 1

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*).

1.1 Jezici sa najviše uticaja

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. Dijagram uticaja ovih jezika na Elixir, kao i vreme nastanka svakog od njih može se videti na slici 1.1:



Slika 1.1: Dijagram uticaja ostalih jezika na Elixir

Lisp

Lisp je drugi najstariji viši programski jezik koji se i danas veoma koristi. Imao je uticaj na programski jezik Erlang, pa je samim tim uticao i na Elixir. Lisp je projektovao **Džon Makarti** 1958. godine, a prvi put je implementiran od strane **Stiva Rasela** na IBM 704 računaru. Rasel je pročitao Makartijev rad i zaključio da izračunavanje u Lispu može da se implementira na mašinskom jeziku. Rezultat toga je bio interpretator koji je mogao da pokreće Lisp programe, ali i da računa vrednosti Lisp izraza. Lisp je zasnovan na matematičkoj teoriji rekurzivnih funkcija (u kojoj se funkcija pojavljuje u sopstvenoj definiciji), a Lisp program je funkcija koja se primenjuje na podatke. On koristi vrlo jednostavnu notaciju u kojoj se operacije i njihovi operandi prikazuju u zagradi. Na primer, $(+a(*bc))$ predstavlja $a+b*c$. Iako se čini neugoDNK, ova notacija dobro funkcioniše na računarima. Ime LISP je nastalo od *„**LIS**t **P**rocessor"*, a povezane liste su jedan od glavnih tipova podataka. Danas postoji veliki broj dijalekata Lisp-a, a najpoznatiji među njima su **Common Lisp** i **Scheme**. Osnova Lisp-a je funkcionalno programiranje, ali se Lisp zbog raznih drugih svojstava smatra multiparadigmatskim programskim jezikom.

Prolog

Prolog (engl. ***PRO****gramming in* ***LOG****ic*) je deklarativan programski jezik namenjen rešavanju zadataka simboličke prirode. On bio osnova za Erlang, te je imao indirektan uticaj i na Elixir. Prolog se temelji na teorijskom modelu logike prvog reda. Početkom 1970-ih godina **Alain Kolmerauer** (engl. *Alain Colmerauer*) i **Filipe Rousel** (engl. *Philippe Roussel*) na Univerzitetu u Marselju (engl. *University of Aix-Marseille*), zajedno sa **Robertom Kovalskim** (engl. *Robert Kowalski*) sa Departmana veštačke inteligencije (engl. *Department of Artifical Intelligence*) na Univerzitetu u Edinburgu (engl. *University of Edinburgh*), razvili su osnovni dizajn jezika Prolog.

Erlang

Firma Erikson je 1981. godine oformila novu laboratoriju **Erikson CSLab** (eng. *The Ericsson CSLab*) sa ciljem da predlaže i stvara nove arhitekture, koncepte i strukture za buduće softverske sisteme. Jedan od projekata nove laboratorije je bio eksperimentisanje sa dodavanjem konkurentnih procesa u programski jezik Prolog i on predstavlja začetak novog programskog jezika. Taj programski jezik 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 mogao posmatrati kao dijalekt Prologa sve do 1990. godine. Od tada, on ima svoju sintaksu i postoji kao potpuno samostalan programski jezik. Nakon mnogo godina rada nastajale su sve brže, bolje i stabilnije verzije jezika, kao i **standarDNK biblioteka OTP** (eng. *The Open Telecom Platform*) (referenca). Erlang i OTP su postali deo slobodnog softvera (eng. *open source software*) u decembru 1998. godine i mogu se slobodno preuzeti sa zvaničnog sajta jezika Erlang. Veliki broj kompanija danas koristi Erlang u razvoju svojih softverskih rešenja. Neke od njih su: Erikson (engl. *Ericsson*), Motorola, Votsap (eng. *WhatsApp*), Jahu (eng. *Yahoo!*), Fejsbuk (eng. *Facebook*) (referenca).

Erlang je funkcionalan, deklarativan i konkurentan programski jezik. Na planu konkurentnosti je svojevrstan primer. Programski jezik Lisp je prvi predstavio funkcionalnu paradigmu i kao takav je imao veliki uticaj na Erlang. Neki od koncepata i osobina koje je nasledio od Lisp-a su koncept rekurzije, *list comprehension* - sintaktička konstrukcija za kreiranje liste zasnovane na postojećim listama, koja sledi formu matematičke notacije *set-builder* za razliku od upotrebe funkcija mapiranja i filtera. Na početku je Erlang stvaran kao dodatak na Prolog, vremenom je prerastao u dijalekt Prologa, a kasnije je zbog svoje kompleksnosti i sveobuhvatnosti evoluirao u potpuno novi programski jezik. Sintaksa jezika Erlang u velikoj meri podseća na sintaksu jezika Prolog (na primer, promenljive moraju počinjati velikim slovom, svaka funkcionalna celina se završava tačkom). Oba jezika u velikoj meri koriste poklapanje obrazaca (eng. *pattern matching*).

Sa druge strane, Erlang je uticao na nastanak programskog jezika Elixir. Uz izmenjenu Erlangovu sintaksu i dopunjenu Erlangovu standardnu biblioteku, Elixir uživa široku popularnost.

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 takođe 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. JeDNK 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. JeDNK 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 priroDNK za čitanje i lako pisanje. Ruby je takođe i interpretirani programski jezik, što znači da se izvorni kod prevodi u kod 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. Na slici 1.2 su prikazane osobine Ruby-ja koje se direktno preslikavaju u osobine Elixir-a:

Ruby	Elixir
`irb`	`iex`
`rake` tasks	`mix` (built in)
`bundler` dependencies	`mix` (built in)
`binding.pry`	`IEx.pry` (built in)
Polymorphism	Protocols
Lazy Enumerables	Streams
Metaprogramming	Macros (used sparingly)
Rails	Phoenix

Slika 1.2: Osobine nasleđene od jezika Ruby

Takođe, na slici 2.1 se može videti i primer koda napisan u oba jezika:

Ruby example (OOP style)

```
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 example

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

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

Slika 1.3: 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 izvršavanja. Predstavlja dijalekt jezika Lisp 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 nasledio neke od najboljih Clojure karakteristika - efikasne, nepromenljive strukture podataka (imutabilnost), opcionalno lenjo izračunavanje, protokole.

1.2 Osnovne karakteristike

Tokom 2010. godine Hosé Valim, u to vreme zaposlen na poziciji programera u kompaniji *Platformatec*, radio je na poboljšanju performansi okruženja *Ruby on Rails* na višejezgarnim sistemima.(referenca) 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*.

Elixir je dinamički tipiziran, funkcionalni programski jezik koji se pokreće na vituelnoj mašini jezika Erlang, pa samim tim i nasleđuje pogoDNK svojstva koje dolaze sa ovim okruženjem kao što su **konkurentnost** i **tolerisanje grešaka**. Iz tačke gledišta olakšavanja svakodnevnog razvoja sofvera, mnogi koncepti su falili ekosistemu jezika Erlang. Neki od njih su **metaprogramiranje** - tehnika kojom programi imaju mogućnost da druge programe posmatraju kao svoje podatke i na taj način čitaju, pa čak i modifikuju njihov, a samim tim i svoj kod u vreme izvršavanja, **polimorfizam**, **makroi**, **podrška za alate**. Upravo navedene koncepte je Elixir nadomestio. Cilj ovog rada je da čitaoca bliže upozna sa osnovnim osobinama, funkcionalnostima i specifičnostima ovog jezika, kao i da kroz primere pokuša da približi programerske prakse korišćene u ovom jeziku.

U ovom delu će biti opisane osobine jezika Elixir, osnove njegove sintakse, semantike, kao i podrška za osnovne koncepte funkcionalnih jezika poput poklapanja obrazaca (eng. *Pattern matching*) i imutabilnosti podataka.

Pre nego što započnemo priču o tipovima, treba pomenuti **Kernel**. To je podrazumevano okruženje koje se koristi u Elixir-u. Ono sadrži primitive jezika kao što su: aritmetičke operacije, rukovanje procesima i tipovima, makroe za definisanje novih funkcionalnosti (funkcija, modula...), provere guard-ova - predefinisanog skupa funkcija i makroa koji proširuju mogućnost pattern matching-a, itd.

1.3 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. Oni su ključevi za listu ključnih reči, koji se često koriste da označe uspeh ili grešku, npr. :ok i mnogi drugi.

Celi brojevi

Celi brojevi su slični kao i u većini ostalih 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 celosti.

Brojevi u pokretnom zarezu

Brojevi u pokretnom zarezu se u memoriji zapisuju po standardu *IEEE 754*, a za zapisivanje konstanti ovog tipa koristi se tačka između najmanje 2 cifre. Takođe je moguće koristiti notaciju koja obuhvata navođenje eksponenata.

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 družine 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 1.1:

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

Listing 1.1: Primer liste

Nadovezivanje ili oduzimanje 2 liste korišćenjem operatora $++/2$ i $--/2$ prikazano je na listingu 1.2:

```
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 1.2: 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. JeDNK od prednosti nepromenljivosti je jasniji kod. Omogućeno je slobodno prosleđivanje podatka sa garancijom da neće biti izmenjeni u memoriji - samo transformisani.

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 1.3:

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

Listing 1.3: Izdvajanje glave i repa liste

Ako pokušamo da izdvojimo glavu ili rep prazne liste, dobićemo grešku.

Prilikom kreiranja liste može se desiti da je rezultat lista vredosti pod jednostrukim navodnicima. Primer koda koji ilustruje ovo može se videti na listingu 1.4:

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

Listing 1.4: Lista vrednosti pod jednostrukim navodnicima

Kada Elixir vidi listu *ASCII* brojeva za štampanje, ispisaće je kao *charlist*-u (doslovno listu znakova). Charlist-e su uobičajene kada se povezuju sa postojećim Erlang kodom. Preuzimanje informacija o vrednosti za koju nismo sigurni kog je tipa može se izvršiti pomoću funkcije *i/1* i može se videti na listingu 1.5:

```
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 1.5: Preuzimanje informacija o tipu vrednosti

Treba imati na umu da reprezentacije sa jednostrukim i dvostrukim navodnicima u Elixir-u nisu ekvivalentne i da predstavljaju različite tipove. Primer možemo videti na listingu 1.6:

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

Listing 1.6: 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 1.7:

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

Listing 1.7: 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. Indeksi torke počinju od nule, a primer se može videti na listingu 1.8:

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

Listing 1.8: Izdvajanje elementa torke sa indeksom 1

Takođe je moguće umetnuti novi element na odredjeno mesto u torki pomoću funkcije *put_elem*/3. Primer koda koji ilustruje upotrebu ove funkcije prikazan je na listingu 1.9:

```
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 1.9: Umetanje novog elementa u torku

Treba obratiti pažnju da je funkcija *put_elem*/3 vratila novu torku. Originalna torka smeštena u promenljivoj *tuple* nije izmenjena. 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 zamenjen. Drugim rečima, torke i liste u Elixir-u mogu da dele svoj sadržaj. Ovo smanjuje količinu memorije koju jezik treba da zauzme i moguće je samo zahvaljujući nepromenljivoj semantici jezika.

Ove karakteristike performansi diktiraju upotrebu struktura podataka. Kao što je već pomenuto, jedan od uobičajenih slučajeva korišćenja torki je prilikom vraćanja dodatnih vrednosti iz funkcije. *File.read/1* je funkcija koja se može koristiti za čitanje sadržaja datoteke. Ona vraća torku, što se može videti na listingu 1.10:

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

Listing 1.10: Primer korišćenja funkcije *File.read/1*

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.

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: $[key : value]$. Zapravo, liste ključnih reči mapiraju liste torki. Primer ovakvog mapiranja prikazan je na listingu 1.11:

```
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 1.11: 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 1.12:

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

Listing 1.12: Dodavanje nove vrednosti listi ključnih reči

Elementima liste ključnih reči se pristupa na način prikazan na listingu 1.13:

```
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 1.13: 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 sto 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 : vrednsot, jer mape ne prate nikakvo uređenje.

Mapa se definiše pomoću sintakse %{} na način prikazan na listingu 1.14:

```
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 1.14: Primer mape i pristupa njenim elementima

Za razliku od liste ključnih reči, 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 1.15:

```
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 1.15: Mape pri podudaranju 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 1.16:

```
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 1.16: Upotreba promenljivih u mapama

Modul **Map** obezbeđuje razne funkcije za manipulaciju mapama, a neke od njih mogu se videti na listingu 1.17:

```
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 1.17: Neke od funkcija modula Map

Mape imaju sintaksu za ažuriranje vrednosti ključa prikazanu na listingu 1.18

```
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 1.18: 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 1.19:

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

Listing 1.19: 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 1.20:

```
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 1.20: 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 nepromenljiva svojstva jezika.

Neka je data struktura prikazana listingom 1.21:

```
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 1.21: Struktura koja predstavlja listu korisnika

Prikazana je lista ključnih reči korisnika, gde je svaka vrednost mapa koja sadrži ime, starost i listu programskih jezika koje svaki korisnik voli. Pristup godinama od Džona mogao bi se izvršiti kao na listingu 1.22:

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

Listing 1.22: Pristup godinama od Džona

Ista sintaksa se može koristiti i za ažuriranje vrednosti kao što je dato listingom 1.23:

```
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 1.23: 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 1.24:

```
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 1.24: Brisanje jezika iz liste

Postoji i funkcija *get_and_update_in* koja omogućava da izvlačenje vrednosti i ažuriranje strukture podataka odjednom. Takođe postoje i funkcije *put_in*/3, *update_in*/3 i *get_and_update_in*/3 koje omogućavaju dinamički pristup strukturama podataka.

1.4 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 3 bool 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 1.25:

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

Listing 1.25: Primer upotrebe bool operatora

Ukoliko kao prvi operand prosledimo nesto čija vrednost nije tipa bool, dobićemo grešku kao na listingu 1.26:

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

Listing 1.26: Greška pri upotrebi bool operatora

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

Pored ovih boolean 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 1.27:

```
1  iex(1)>1 || true
2  1
3  iex(2)>false || 11
4  11
5  iex(3)>nil && 13
6  nil
7  iex(4)true && 17
8  17
9  iex(5)>!true
10 false
11 iex(6)!1
12 false
```

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

Listing 1.27: Operatori koji prihvataju argumente bilo kog tipa

Može se smatrati pravilom da kada se očekuju bool vrednosti, treba koristitioperatore and i or, a ako bilo koji od operanada ima vrednosti koji nisu tipa bool, 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 1.28:

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

Listing 1.28: Poređenje tipova

1.5 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** (eng. *matching*). On se uspešno izvršava, ako pronađe način da izjeDNKči levu stanu (svoj prvi operand) sa desnom (drugi operand).

Na primer, izraz $2 + 2 = 5$ bi rezultirao greškom datom na listingu 1.29:

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

Listing 1.29: 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 1.30 nije greška, već uspešno poklapanje obrazaca:

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

Listing 1.30: Uspešno poklapanje obrazaca

Slično, 2 identična stringa sa obe strane znaka jeDNKkosti će dati rezultat prikazan listingom 1.31:

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

Listing 1.31: Uspešno poklapanje obrazaca sa stringovima

Poklapanje obrazaca se može prikazati i na pimeru sa listama. Neka je data lista osoba koja je prikazana listingom 1.32:

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

Listing 1.32: Lista osoba

Prve tri osobe treba da budu zapamćene, dok četvrta osoba nije bitna. U te svrhe se može iskoristiti poklapanje obrazaca dato na listingu 1.33:

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

```
3  "Lena Markovic"]
```

Listing 1.33: 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 1.34:

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

Listing 1.34: Iščitavanje sadržaja promenljivih

1.6 Imutabilnost 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 programe. Ali šta se dešava kada se vrednost u memoriji promeni u trenutku kada je koristi više instanci programa? Neka se ne promeni samo vrednost, već i tip. Ovakvo ponasanje je poznato kao **mutabilnost (promenljivost)**. U konkurentnim okruženjima je izvor grešaka koje je veoma teško pratiti i reprodukovati. Mutabinost takođe vodi veoma komplikovanom kodu, pisanom ad-hoc kako bi se rešili problemi sinhronizacije. Ovo može smanjiti rizik da konkurentni procesi pristupaju istim resursima, ali po veoma visokoj ceni.

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

1.7 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 satoji od bool 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 bool 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.

1.8 Moduli

U Elixir-u možemo grupisati nekoliko funkcija u module. Već su pomenuti različiti module 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 1.35:

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

Listing 1.35: Kreiranje modula i funkcija

Moduli mogu biti ugnježdeni u Elixir-u. Ova osobina jezika omogućava organizovanje koda na što bolji način. Za ugnježdavanje modula, koristi se sintaksa sa listinga 1.36:

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

Listing 1.36: Ugnježdavanje modula

Gore navedeni primer definiše 2 modula: **Math** i **Math.Adding**. 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 potavljen pomoću direktive aliasa.

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.

1.9 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 1.37:

```
1 #Alias modula tako da se mo e pozvati sa Adding umesto sa Math.
    Adding
2 alias Math.Adding, as: Adding
3
4 #Obezbe uje da je modul kompajliran i dostupan (obi no za makroe)
5 require Math
6
7 #Uklju uje prilago en kod definisan u Math kao pro irenje
8 use Math
```

Listing 1.37: 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 od strane progaramera za unos spoljne funkcionalnosti u trenutni leksički opseg, često modula.

Glava 2

Sekvencioniranje genoma

Genom je skup gena jednog organizma. Svaki gen predstavlja pravilo za sintezu jednog proteina u ćeliji, koji je neophodan za njeno pravilno funkcionisanje. Geni su delovi **DNK** [1] sekvence koja je jedinstvena za svaki organizam, a koja je sa računarske strane niska nad azbukom nukleotida $\{A, C, G, T\}$. Savremene laboratorijske metode za dati uzorak 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**. Rekonstrukcija genoma može se uporediti sa kompletiranjem slagalice, gde su očitavanja delovi slagalice. Što su delovi veći, slagalicu je lakše sastaviti. Genomi se sastoje od niza uparenih baza. Na taj način se genomi i očitavanja DNK mogu meriti u **baznim parovima (bp)**, pa se mogu razlikovati **kratka** i **duga** očitavanja. Kratka očitavanja imaju dužinu od 50 bp do 400 bp, a duga očitavanja dužinu veću od 400 bp.

[1] DNK (dezoksiribonukleinska kiselina) je nukleinska kiselina koja sadrži uputstva za razvoj i pravilno funkcionisanje svih živih organizama. DNK ima veoma važnu ulogu ne samo u prenosu genetičkih informacija sa jedne na drugu generaciju, već sadrži i uputstva za građenje neophodnih ćelijskih organela.

2.1 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. **Referentni genom** (poznat i kao referentni sklop) je digitalna baza podataka o nukleinskim kiselinama, koju su naučnici prikupili kao reprezentativni primer seta gena vrste. Kako su često sastavljeni sekvencioniranjem DNK jednog broja davalaca, referentni genomi ne predstavljaju skup gena nijedne pojedinačne osobe. **Humani genom *GRCh37*** [3] je izveden 2009. godine iz DNK trinaest anonimnih dobrovoljaca iz Bafala (engl. *Buffalo*) (Njujork). Referentni genomi se obično koriste kao uputstvo na osnovu koga se grade novi genomi, što im omogućava da se sastave mnogo brže i povoljnije. Međutim, kako je Sanger sekvencioniranje niskopropusno (ima manju stopu greške) i ne tako povoljno, samo nekoliko genoma je sastavljeno Sanger sekvencioniranjem.

Razvoj tehnika za sekvencioniranje **druge generacije** značajno je doprineo efikasnijem i ekonomičnijem sekvencioniraranju stotine miliona očitavanja. Međutim, očitavanja druge generacije sekvencioniranja su kratka. Po narudžbini su pravljeni genomski asembleri za rekonstrukciju genoma na osnovu kratkih očitavanja. Njihova pojava je dovela do većeg broja uspešnih ***de novo*** asemblerskih 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** [4], dugi.

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 ispravnije postavljanje ovih karakteristika 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] OLC je skraćenica od *overlap layout consensus*, gde *overlap* predstavlja izgradnju grafa preklapanja, *layout* spajanje putanja u grafu u kontige, a *concensus* određivanje najverovatnije sekvence nukleotida za svaku kontigu

[3] **GRCh37** - The Genome Reference Consortium human genome (build 37)

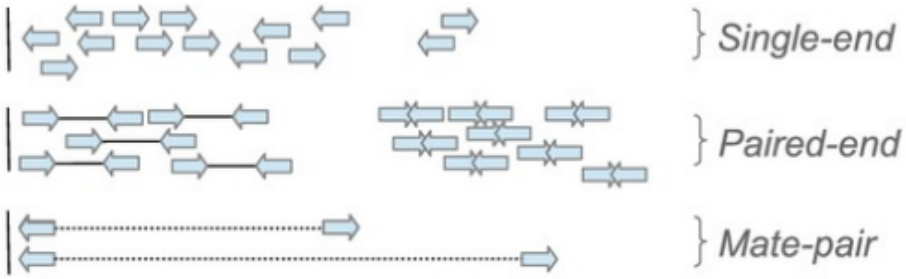
[4] Ponovci su fenomen kada se šablon od nekoliko nukleotida pojavljuje više puta u nizu.

2.2 *Shotgun* sekvencioniranje celokupnog genoma

Prvi korak u procesu sekvencioniranja genoma je njegovo razbijanje na skup očitavanja, a na osnovu uzorka genoma. Postoji više vrsta očitavanja:

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

Sekvenceri pročitaju deo DNK fragmenta neke unapred zadate dužine koja se naziva **dužina očitavanja**. Kako bi se dobila potrebna **dubina pokrivanja**, isti deo fragmenta biva pročitan veći broj puta. Dubina pokrivanja neke pozicije u DNK sekvenci je velika, ako je nukleotid na toj poziciji pročitan veliki broj puta u jedinstvenim očitavanjima. Ovako nastaju jednostrana očitavanja, dok uparena očitavanja nastaju pri očitavanju u oba smera. Posebna vrsta uparenih očitavanja su **partner-uparena očitavanja** (engl. *mate-pair reads*). Prikaz svih vrsta očitavanja može se videti na slici :



Slika 2.1: Vrste očitavanja

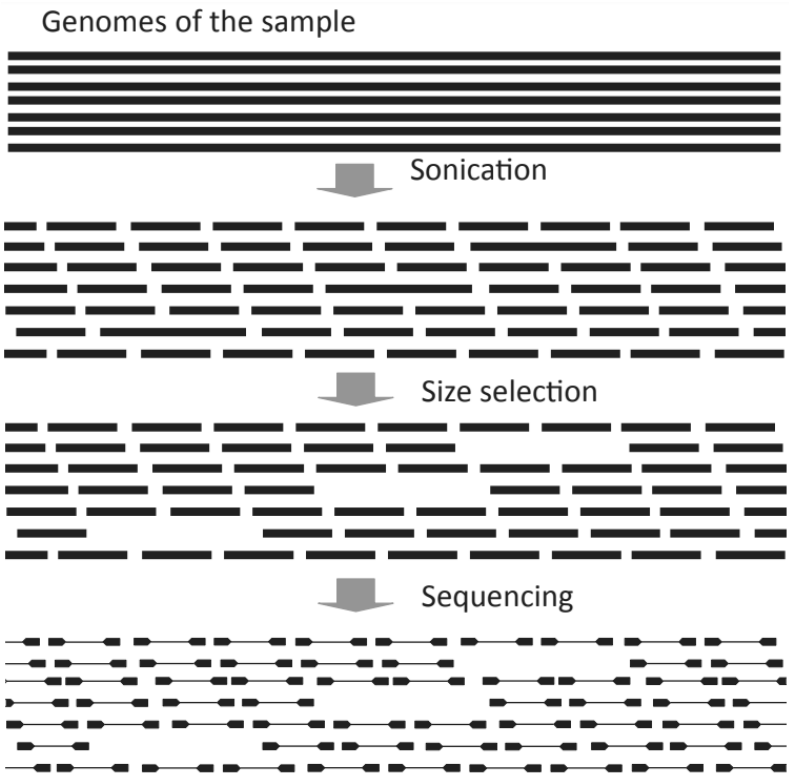
[1]

Za potrebe *shotgun* sekvencioniranja celokupnog genoma postoje 2 protokola:

- sekvencioniranje celokupnog genoma
- sekvencioniranje partner-uparenih očitavanja

Sekvencioniranje celokupnog genoma

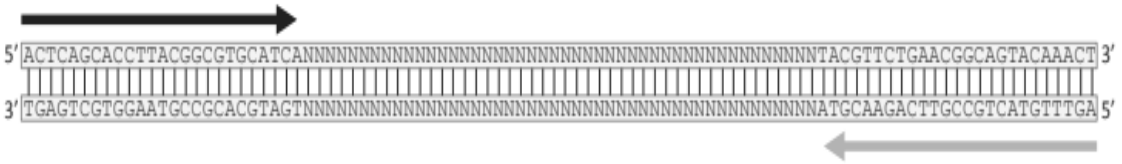
Sekvencioniranje celokupnog genoma uključuje 3 koraka koja su prikazana na slici 2.2:



Slika 2.2: Ilustracija protokola *shotgun* za sekvencioniranje celokupnog genoma. Uključuje 3 koraka: (1) korak razbijanja genoma (*sonication*), (2) korak odabira dužine očitavanja i izdvajanja fragmenata te dužine (*size selection*), (3) korak očitavanja kraj(ev)a svakog fragmenta (*sequencing*)

[1] str. 122, slika 5.1

Prvo, uzorak genoma se na slučajan način razbija na DNK fragmente pomoću *sonication* ili *enzimskog* sečenja. 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 jendostranih očitavanja, sekvencer čita DNK fragment u jednom smeru, dok pri sekvencioniranju uparenih očitavanja, sekvencer čita DNK fragment u oba smera. (Sekvenceri treće generacije čitaju čitav DNK fragment. To se smatra jednostranim očitavanjem.) Slika 2.3 daje jedan takav primer:



Slika 2.3: Razmatra se DNK fragment ekstrahovan iz uzorka genoma. Upareno očitavanje se formira izdvajanjem dve DNK podniske (koje se takođe nazivaju oznake (engl. *tags*)) sa oba kraja DNK fragmenta.

[1] str. 122, slika 5.2

Za sekvencioniranje jednostranih očitavanja, dobija se očitavanje sa 5' kraja [5] *forward* šablona DNK fragmenta, tj. $ACTCAGCACCTTACGGCGTGCATCA$. Za sekvencioniranje uparenih očitavanja, dobijaju se 5' očitavanja i *forward* i *reverse* šablona DNK fragmenta (u unutrašnjoj orijentaciji, tj. očitavanje šablona od 5' ka 3'), tj:

- $ACTCAGCACCTTACGGCGTGCATCA$
- $AGTTTGTACTGCCGTTCAGAACGTA$

Drugo očitavanje je inverzni komplement prvog, jer se ono dobija iz *reverse* šablona.

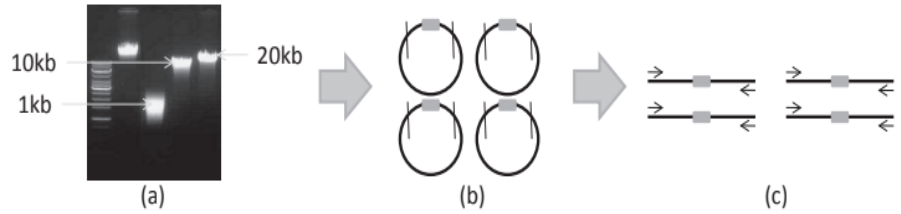
Za asembliranje genoma, dužina očitavanja je veoma važan parametar. Različite tehnologije sekvencioniranja imaju različita ograničenja na dužinu očitavanja. Na primer, *Illumina Hi-seq* [6] sekvencioniranje može izvršiti samo sekvencioniranje uparenih očitavanja dužine manje od 1000 bp. Za treću generaciju sekvencioniranja, ograničenje dužine očitavanja može biti veće od 10000 bp.

[5] $5'$ i $3'$ označavaju brojeve ugljenika koji se nalaze u osnovi DNK šećera. $5'$ ugljenik ima fosfatnu grupu koja je vezana za njega, a $3'$ ugljenik-hidroksilnu (-OH) grupu. Ova asimetrija daje DNK pravac $5' \rightarrow 3'$. Na primer, DNK polimeraza deluje u pravcu $5' \rightarrow 3'$, tj. dodaje nukleotide na $3'$ kraj molekula napredujući u tom smeru.

[6] Illumina je američka kompanija koja je proizvodila sekvencere sa malom stopom greške (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** prikazano na slici 2.4 :



Slika 2.4: Ilustracija protokola sekvencioniranja partner-uparenih očitavanja. (a) Izdvajanje svih fragmenata određene dužine očitavanja, npr. 10000 bp, gelom za rezanje (cutting gel?).(b) Svaki fragment je cirkularizovan sa adapterom (predstavljen blokom sive boje) u sredini. Zatim se vrši sečenje 2 područja oko adaptera. (c) Konačno, uparena očitavanja se ekstrahuju sekvencioniranjem uparenih očitavanja.

[1] str. 123, slika 5.3

Prvo se dugi fragmenti DNK neke fiksirane dužine očitavanja (npr. 10000 bp) biraju sečenjem gena. Zatim se dugi fragmenti cirkularizuju pomoću adaptera [7]. Cirkularizovani DNK-ovi su fragmentisani i samo se zadržavaju fragmenti koji sadrže adaptere. Na kraju, pomoću sekvencioniranja uparenih očitavanja, partner-uparena očitavanja se sekvencioniraju na osnovu fragmenata DNK sa adapterima.

Orijentacija uparenog očitavanja očitanog od strane sekvencera partner-uparenog očitavanja se razlikuje od onog koji je očitan od strane sekvencera uparenog očitavanja. Sekvenceri partner-uparenih očitavanja daju dva očitavanja sa oba kraja svakog fragmenta DNK u spoljašnjoj orijentaciji umesto u unutrašnjoj. Npr. za DNK fragment na slici 2.3 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 DNK za pripremu sekvencerskih biblioteka i sklon je lažnim greškama.

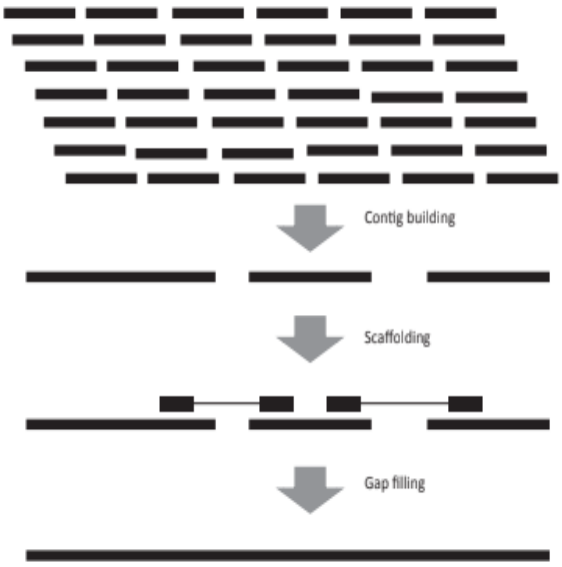
[7] Adapter sekvence su kratke hemijski sintetizovane sekvence nukleotida koje se mogu vezati za krajeve nepoznatih DNK sekvenci i neophodne su u nekim koracima sekvencioniranja.

2.3 *De novo* sekvencioniranje genoma za kratka očitavanja

Druga generacija 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 suprstringa** za koji se zna da je NP-kompletan (problem superstringa - na osnovu skupa stringova S, teži se pronalaženju superstringa, najkraćeg stringa P takavog da je svaki string s iz skupa S substring stringa P. Na primer, ako je $S = \{ACATGC, ATGCGTGT, GTGTACGT\}$, onda je superstring ACATGCGTGTACGT).

Mnogi *de novo* asembleri [8] predlažu asembliranje kratkih očitavanja. Opšte rešenje uključuje 4 koraka prikazana na slici 2.5:



Slika 2.5: Četiri koraka u sekvencioniranju genoma. 1. Korekcija očitavanja: Ispravljanje greškaka sekvencioniranja u očitavanjima. 2. Izgradnja kontiga: Spajanje očitavanja radi formiranja kontiga. 3. Formiranje skafolda: Korišćenjem uparenih očitavanja se vrši povezivanje kontiga kako bi se formirali skafoldi. 4. Popunjavanje praznina: Za susedne kontige u skafoldima, vrši se popunjavanje praznina.

[1] str. 124, slika 5.4

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

U prvom koraku se ispravljaju 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. Zbog ponovaka se javljaju dvosmislenosti, te nije moguće rekonstruisati kompletan genom. Postojeće metode preklapanjem očitavanja daju kontinuiranu sekvencu, takozvanu **kontigu** (engl. *contig*). Kontige obično predstavljaju jednu **konsenzus nisku** [9]. Zatim, koristeći uparena očitavanja, pokušava se rekonstrukcija redosleda kontiga tako da se formiraju **skafoldi** (engl. *scaffolds*) [10]. Na kraju se vrši preuređivanje očitavanja u skafoldima kako bi se popunile praznine između susednih kontiga.

2.4 k-mer counting

Jedan konceptualno jednostavan, ali osnovni problem je **brojanje k-mera** (k-mer - podniska dužine k). Predstavlja potprogram koji se koristi u korekciji grešaka u očitavanjima. Takođe može biti korišćen u koraku asembliranja, detekciji ponovaka i kompresiji genomskih podataka. Ulaz 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 [11]. Kada je k malo (npr. manje od 10), koristi se **savršeno heširanje**. Savršeno heširanje garantuje da neće doći do **kolizije** [12]. 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

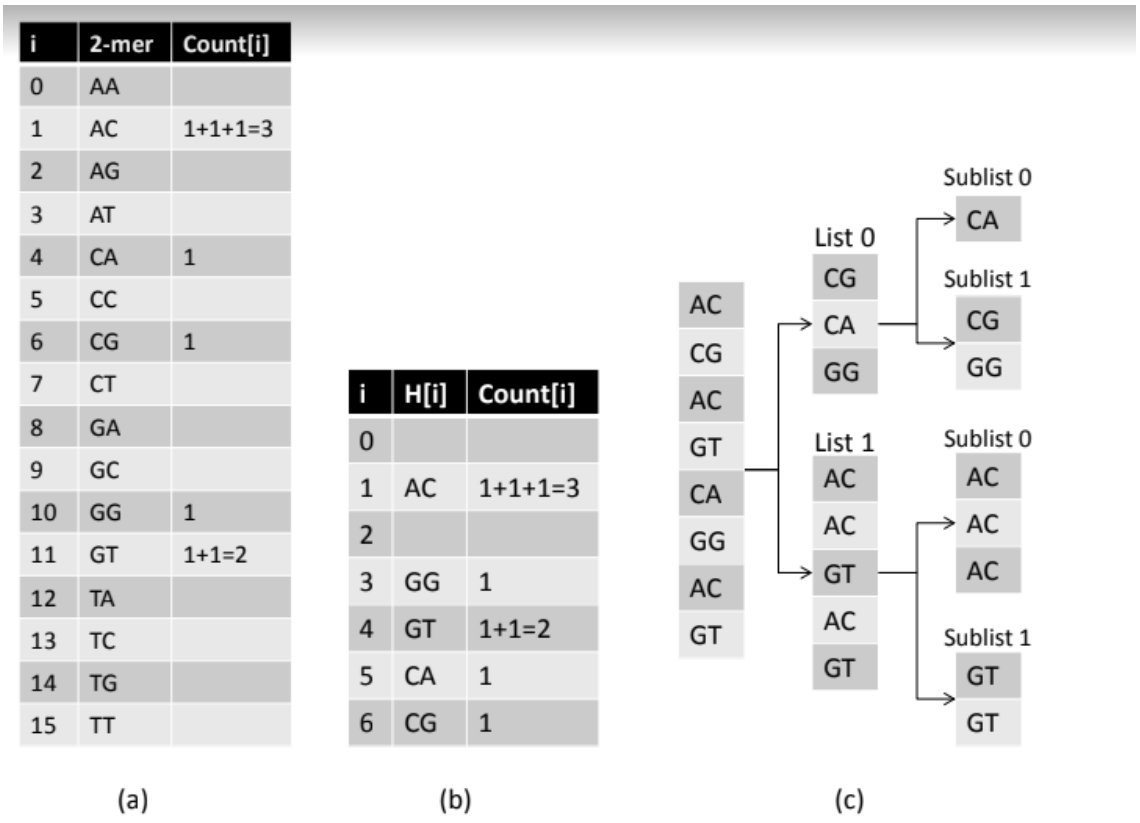
[9] Niska sastavljena od najfrekventnijih nukleotida na pozicijama poravnatih sekvenci

[10] Svaki skafold je niz kontiga, a jos se naziva i **superkontig** ili **metakontig**

[11] 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.

[12] 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.

skupu Z. Preciznije, vrši se 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. Slika 2.6(a) predstavlja primer koji ilustruje ovaj jednostavni metod prebrojavanja:



Slika 2.6: Razmatra se skup 4-mera $Z = \{AC; CG; AC; GT; CA; GG; AC; GT\}$: (a) Ilustruje jednostavan metod za brojanje k-mera koji koristi *Count* tabelu veličine 4k. (b) Ilustruje *JellyFish* metod brojanja k-mera koja koristi heš tabelu veličine 7. Heš funkcija je $h(z) = b(z)$ *mod* 7. Na primer, GT se čuva u tabeli $Count$ sa indeksom 4, jer je $h(GT) = 4$. U ovom primeru se javlja je DNK kolizija. Pošto je i $h(CA) = 4$, CA je u koliziji sa GT. Linearnim isprobavanjem CA se ipak čuva u tabeli $Count$ sa indeksom 5. (c) Ilustruje DSK metod brojanja k-mera. Pretpostavka je da je $h(z) = b(z)$, $n_{list} = 2$ i $n_{sublist} = 2$. DSK deli Z u 4 $(= n_{list} * n_{sublist})$ podliste, a zatim pokreće *JellyFish* algoritam za brojanje k-mera u svakoj podlisti.

[1] str. 131, slika 5.8

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 mehanizma **otvorenog adresiranja** [13]. Neka je $h()$ heš funkcija i $H[0..\frac{N}{\alpha}-1]$ heš tabela koja čuva niz k-mera, gde je α **faktor opterećenja** [14] $(0 < \alpha \leq 1)$. Potrebno je izgraditi tabelu $Count[0..\frac{N}{\alpha}-1]$ gde $Count[i]$ čuva broj 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 $H[i] \neq z$, ne možemo čuvati $z >$ 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 2.7 (donji deo slike) 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.

JellyFish algoritam je detaljnije objašnjen na slici 2.7 (gorni deo slike), dok slika 2.6 (b) daje primer koji ga ilustruje. On 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. Gore pomenuta ideja smanjivanja veličine heš tabele je iskorišćena u $JellyFish$ algoritmu.

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.

[13] 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 3 metode otvorenog adresiranja: linearno popunjavanje, kvadratno popunjavanje i duplo heširanje.

[14] Broj koji kontroliše veličinu heš tabele

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 $H[i]$;

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

Slika 2.7: Jellyfish algoritam i funkcija hashEntry

[1] str. 132, slika 5.9

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. Iako on može dati pogrešno pozitivan rezultat (pogrešan izveštaj da k-mer postoji ili pogrešno proceniti broj k-mera), ne može dati pogrešno negativan rezultat. BFCounter održava *counting Bloom* filter B i heš tabelu H. Sastoji se od 2 faze. Prva faza počinje sa praznim *counting Bloom* filterom B i praznom heš tabelom H. U ovoj fazi se skeniraju k-meri iz Z jedan po jedan. Za svaki k-mer z se proverava da li se z pojavljuje najmanje $q - 1$ puta u *counting Bloom* filteru B

utvrđivanjem da li je $countBloom(x, B) \geq q - 1$. Ako nije, vrši se umetanje z u Z pomoću $insertBloom(z, B)$. U suprotnom, z se pojavljuje najmanje q puta. Zatim se vrši proveravanje da li je z u heš tabeli H. Ako nije, z se umeće u neki prazan ulaz $H[i]$ i postavlja se $count[i]$ na nulu.

U drugoj fazi se obavlja stvarno brojanje. Vrši se skeniranje k-mera iz Z jedan po jedan. Za svaki k-mer z iz Z, ako se z pojavljuje u heširanom ulazu $H[i]$, onda se uvećava $count[i]$ za 1. Detaljan pseudokod je prikazan na slici 2.10:

Algorithm BFCounter(Z, q, 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: For every k-mer in Z occurring at least twice, report its count.
1: Set $H[1..\frac{N}{2\alpha}]$ be an empty hash table where each entry requires $2k$ bits;
2: Set $Count[1..\frac{N}{q\alpha}]$ be an empty vector where each entry requires 32 bits;
3: Set $B[0..m-1]$ be an empty vector where each entry requires $\lceil \log q \rceil$ bits and $m = 8N$;
4: **for** each k-mer z in Z **do**
5: **if** countBloom$(z, B) \geq q - 1$ **then**
6: $i = hashEntry(z, h, \frac{N}{q\alpha})$;
7: **if** $H[i]$ is empty **then**
8: Set $H[i] = z$ and $Count[i] = 0$;
9: **end if**
10: **else**
11: insertBloom(z, B);
12: **end if**
13: **end for**
14: **for** each k-mer z in Z **do**
15: $i = hashEntry(z, h, \frac{N}{q\alpha})$;
16: **if** $H[i] = z$ **then**
17: $Count[i] = Count[i] + 1$;
18: **end if**
19: **end for**
20: Output $(H[i], Count[i])$ for all non-empty entries $H[i]$ where $Count[i] \geq q$;

Slika 2.8: Prostorno efikasan algoritam za prebrojavanje k-mera koji broji samo k-mere koji se pojavljuju najmanje q puta.

[1] str. 134, slika 5.10

Vreme izvršavanja BFCounter algoritma je $O(n)$. Što se tiče prostorne složenosti, *counting Bloom* filter zahteva $O(Nlog(q))$. 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**, a čiji pseudokod je prikazan na slici 2.9:

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: $n_{list} = \frac{2kN}{D}$;
2: $n_{sublist} = \frac{D(2k+32)}{0.7(2k)M}$;
3: **for** $i = 0$ to $n_{list} - 1$ **do**
4: Initialize a set of empty sublists $\{d_0, \ldots, d_{n_{sublist}-1}\}$ 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 2.9: DSK algoritam

[1] str. 136, slika 5.11

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 $JellyFish$ sa slike 2.7.

Preciznije, 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 2.7.

Na slici 2.6(c) se može videti primer koji ilustruje izvršavanje algoritma DSK. Neka je $n_{list} = 2$, $n_{sublist} = 2$ and $h(z) = b(z)$ za svaki $z \in Z$. Kako je $n_{list} = 2$, algoritam izvršava 2 iteracije (u nastavku sledi opis nulte iteracije, jer se prva izvršava slično). Prva faza nulte iteracije skenira sve k-mere iz Z i identifikuje svaki k-mer $z \in Z$ koji pripada nultoj listi. Na primer, $h(GG) = 10$, kako je $h(GG)$ mod $n_{list} = 0$ i $\frac{h(z)}{n_{list}}$ mod $n_{sublist} = 1$, GG pripada nultoj listi i prvoj podlisti. Nakon toga, nulta lista se deli na nultu podlistu $\{CA\}$ i prvu podlistu $\{CG, GG\}$. Obe podliste su zapisane na disku. Druga faza čita svaku podlistu iz memorije i broji k-mere koristeći $JellyFish$ algoritam.

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.

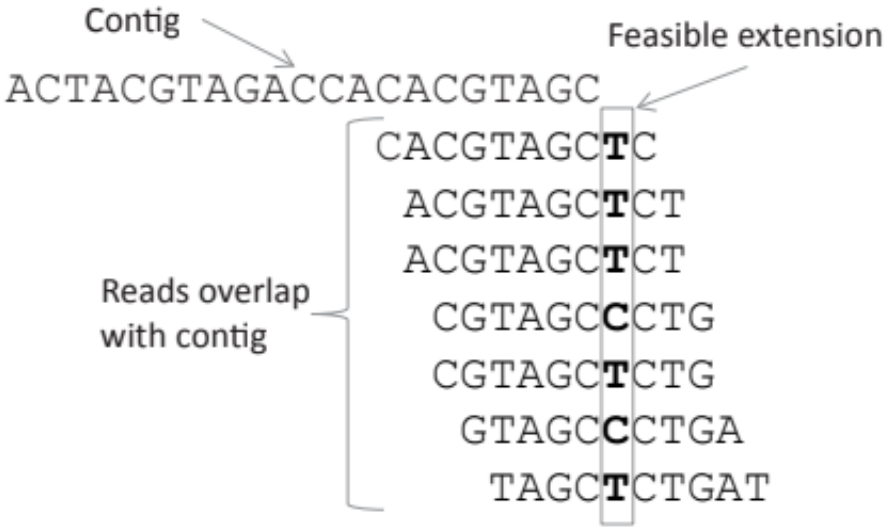
2.5 Konstrukcija kontiga

Nakon što su sva očitavanja korigovana, može se vršiti njihovo spajanje radi formiranja kontiga. Postoje 2 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 (baze su A, C, G, T). Metod počinje tako što se nasumično bira očitavanje koje će služiti kao šablon. Očitavanja su poravnata na oba kraja šablona (3' kraj i 5' kraj). Na osnovu poravnanja se dobija **konsenzusna baza** [15] i šablon se njome proširuje. Slika 2.10 ilustruje korak baznog proširenja:



Slika 2.10: Gornja sekvenca predstavlja šablon. Postoji 7 očitavanja koja su poravnata na $3'$ kraju šablona. Pravougaonik pokazuje da su baze C i T izvodljivo proširenje šablona. Kako je T konsenzusna baza, metoda baznog proširenja će proširiti kontigu bazom T.

[1] str. 137, slika 5.12

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

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. Slika 2.11 daje pseudokod ovog metoda:

Algorithm SimpleAssembler($\mathcal{R}$)
Require: $\mathcal{R}$ is a set of reads
Ensure: A set of contigs
1: **while** some read R are not used for reconstructing contigs **do**
2: Set the template $T = R$;
3: **repeat**
4: Identify a set of reads that align to the 3' end (or 5' end) of the template T;
5: Identify all feasible extensions of the template;
6: If there is a consensus base b, set $T = Tb$ if the extension is at 3'end and set $T = bT$ if the extension is at 5' end;
7: **until** there is no consensus base;
8: Report the template as a contig;
9: Mark all reads aligned to the contig as used;
10: **end while**

Slika 2.11: Jednostavan base-by-base asembler proširenja

[1] str. 138, slika 5.13

Iako je bazno proširenje jednostavno, ono često daje kratke kontige zbog 2 problema. Prvi, početni šablon je nasumično izabrano očitavanje. Ako očitavanje 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.

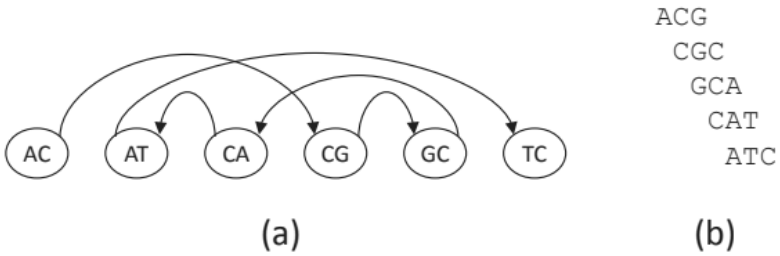
De Brujinov grafovski pristup

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.

Prvo treba definisati De Brujinov graf. Zatim se razmatra skup očitavanja R i parametar k. De Brujinov graf je graf $H_k = (V, E)$, gde je V skup svih k-mera skupa R. 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 primer, neka je dat skup stringova $R = \{ACGC, CATC, GCA\}$ i neka je potrebno izgraditi De Brujinov graf za date k-mere. Tada je skup čvorova $\{AC, AT, CA, CG, GC, TC\}$. Slika 2.12(a) daje prikaz De Brujinovog grafa:



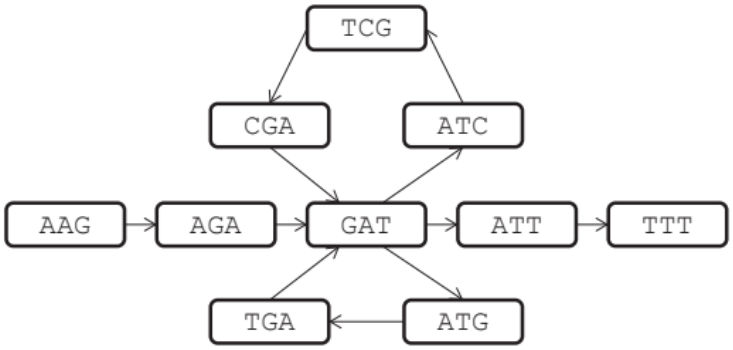
Slika 2.12: (a) De Brujinov graf H_3 za $R = \{ACGC, CATC, GCA\}$. (b) Preklapanje 5 3-mera koji odgovaraju granama grafa H_3.

[1] str. 139, slika 5.15

Uzorak genoma se može predvideti identifikovanjem **Ojlerove putanje** [16] grafa H_k. Ojlerova putanja grafa H_k može biti izračunata u $O(n)$ vremenu, ako H_k ima n grana. Na primer, na slici 2.12(a) postoji jedinstvena putanja od čvora AC do čvora TC. Preklapanjem svih ivica 3-mera u redosledu putanje (slika 2.12(b)) dobija se sekvenca $ACGCATC$. Sekvenca $ACGCATC$ je zapravo superstring formiran preklapanjem očitavanja i dobijen na osnovu De Brujinovog grafa za skup R.

Medjutim, Ojlerova putanja ne mora biti jedinstvena u H_k. Na primer, neka je skup očitavanja $R = \{AAGATC, GATCGAT, CGATGA, ATGATT, GATTT\}$ i neka je $k = 3$. De Brujinov graf H_3 je prikazan na slici 2.13:

[16] Ojlerova putanja je putanja koja obilazi svaku granu grafa H_k tačno jednom.



Slika 2.13: De Brujinov graf H_3 za $R = \{AAGATC, GATCGAT, CGATGA, ATGATT, GATTT\}$

[1] str. 140, slika 5.16

Kao što se može videti, postoje dve Ojlerove putanje u grafu H_3. Ukoliko se prvo obiđe gornji ciklus, dobija se $AAGATCGATGATTT$, a ukoliko se prvo obiđe donji ciklus, dobija se $AAGATGATCGATTT$. Ovaj primer ukazuje na to da Ojlerova putanja mozda neće uvek dati ispravnu sekvencu. Čak još gore, Ojlerova putanja možda neće postojati u nekom grafu H_k.

U daljem tekstu govoriće se o De Brujinovom grafovskom asembleru u slučaju kada:

- ne postoji greška sekvencioniranja
- postoji greška sekvencioniranja

De Brujinov asembler (bez greške sekvencioniranja)

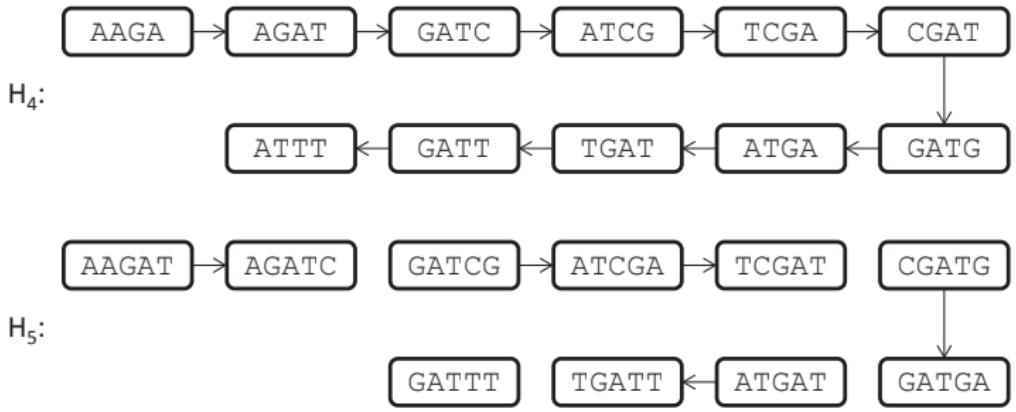
Kako Ojlerova putanja nije jedinstvena i možda i ne postoji, ne teži se dobijanju kompletnog genoma. Umesto toga, teži se dobijanju skupa kontiga. Kontiga je maksimalna prosta putanja u H_k. Preciznije, svaka maksimalna prosta putanja je maksimalna putanja u H_k tako da svaki čvor (osim početnog i krajnjeg) ima unutrašnji i spoljašnji stepen 1. Slika 2.14 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 2.14: Jednstavan De Brujinov grafovski asmbler

[1] str. 141, slika 5.17

Za De Brujinov graf H_3 sa slike 2.13 mogu se konstruisati 4 kontige: $AAGAT, GATCGAT, GATG$
Parametar k je veoma važan. Za različito k, mogu se dobiti različiti skupovi kontiga. Slika 2.15 ilustruje ovaj problem na De Brujinovom grafu za $k = 4$ i $k = 5$:



Slika 2.15: De Brujinov graf H_4 i H_5 za $R = AAGATC, GATCGAT, CGATGA, ATGATT, GATTT$

[1] str. 141, slika 5.18

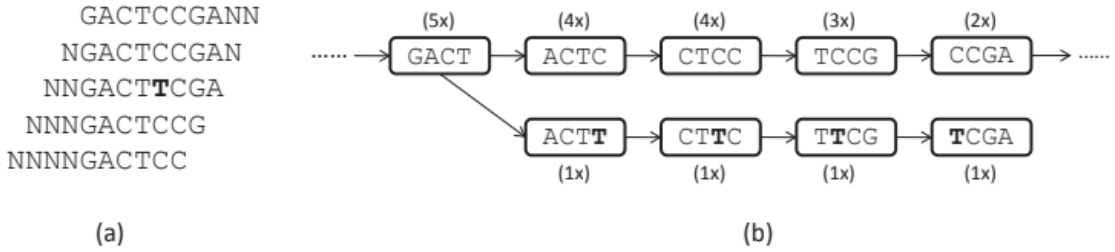
Kada je $k = 4$, H_4 je prosta putanja i može se konstruisati jedna kontiga $AAGATCGATGATTT$. U slučaju da je $k = 5$, H_5 sadrži 5 povezanih komponenti, svaka je prosta putanja i može se konstruisati 5 kontiga: $AAGATC$, $GATCGAT$, $CGATGA$, $ATGATT$, $GATTT$.

Iz gore navedenih primera, može se primetiti da je De Brujinov graf dobar, ukoliko se zna ispravno k. Kada je k malo (pogledati H_3 na slici 2.13), postoji veliki broj grana zbog ponovaka i rezultat je veliki broj kratkih kontiga. Kada je k veliko (pogledati H_5 na slici 2.15), neki k-meri nedostaju. Ovo 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)

U prethodnom odeljku se pretpostavljalo 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*). Treba obratiti pažnju da kratka očitavanja imaju nisku stopu greške (1 greška na svakih 100 baza). Većina k-mera sadrži najviše 1 grešku. Ovi k-meri sa greškom mogu kreirati 2 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 potencijalni kontigu dužine najviše $2k$. Slika ilustruje vrh kreiran na osnovu jednog nepoklapanja u jednom očitavanju:



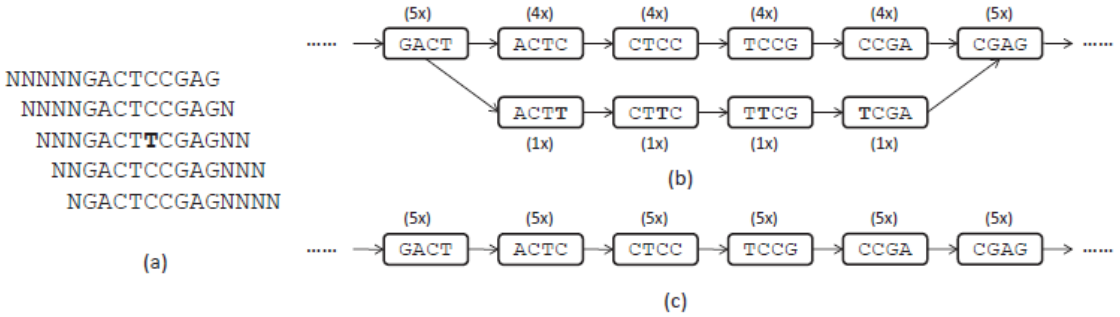
Slika 2.16: (a) Višestruko poravnanje skupa od 5 očitavanja, gde treće očitavanje ima grešku sekvencioniranja (prikazana podebljanim fontom). (b) De Brujinov graf koji odgovara skupu očitavanja iz (a). Vrh je napravljen. Brojevi u zagradama označavaju mnogostrukost 4-mera.

[1] str. 142, slika 5.19

Ako su svi čvorovi na vrhu niske mnogostrukosti (k-meri koji se mali broj puta pojavljuju u očitavanjima), takva kratka kontiga teško da može biti ispravna (preformulisati). Ovaj vrh se može ukloniti iz De Brujinovog grafa, ali treba primetiti da uklanjanje vrha iz De Brujinovog grafa može generisati nove vrhove. Tako da procedura uklanjanja mora uklanjati vrhove rekurzivno.

Mehurić se sastoji od dve kratke različite putanje sa istim početnim i krajnim čvorovima u De Brujinovom grafu, pri čemu te 2 putanje zapravo kontige koje se razlikuju u samo jednom nukleotidu. Na primer, slika 2.17(a) sadrži skup očitavanja gde treće očitavanje ima jedno nepoklapanje. Slika 2.17(b) prikazuje odgovarajući De Brujinov graf i jedan mehurić koji se formira. Gornja putanja mehurića predstavlja $GACTCCGAG$. Donja putanja mehurića predstavlja $GACTTCGAG$.

Kada su dve putanje u mehuriću veoma slične, putanja sa nižom mnogostukošću će verovatno biti lažno pozitivna. Može se pokušati sa spajanjem (otklanjanjem) mehurića. U ovom primeru, 2 putanje imaju samo jedno nepoklapanje. Budući da su čvorovi u donjoj putanji niže mnogostrukosti, vrši se spajanje (otklanjanje) mehurića i dobija se graf na slici 2.16(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.



Slika 2.17: (a) Višestruko poravnanje skupa od 5 očitavanja, gde treće očitavanje ima grešku sekvencioniranja (prikazana podebljanim fontom). (b) De Brujinov graf koji odgovara skupu očitavanja iz (a). Mehurić je napravljen. (c) Putanja dobijena spajanjem mehurića iz (b).

[1] str. 143, slika 5.20

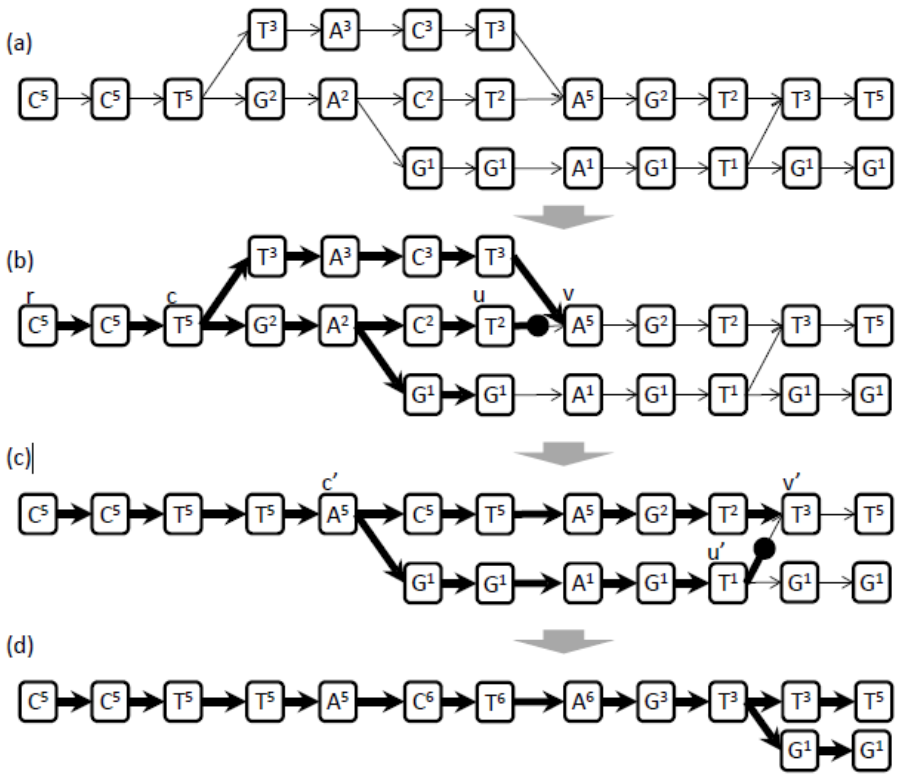
Za spajanje (otklanjanje) mehurića, može se koristiti **algoritam turneje** (engl. *tour bus algorithm*). Ovaj algoritam je nalik na Dijkstrinu pretragu u širinu baziranu na *BFS metodu*. Algoritam počinje od proizvoljnog čvora s i posećuje čvorove u rastućem poretku rastojanja od početnog čvora. Kada algoritam obrađuje neposećeni čvor u, on proverava svu njegovu decu v. Za svako dete v se izvode 2 koraka. Prvo, dodeljuje se u kao roditelj od v u BFS stablu postavljanjem $\pi(v) = u$. Drugo, ako je dete v od u posećeno, mehurić je detektovan. Algoritam izračunava najnižeg zajedničkog pretka c od u i v u BFS stablu definisanog sa $\pi()$. Zatim se putanje $c \rightarrow u$ i $c \rightarrow v$ upoređuju. Ako su dovoljno slične, spajaju se. Zadržavamo putanju sa većom težinom. Algoritam je detaljnije prikazan na slici 2.18:

```
Algorithm Tour_Bus(H, s)
Require: H is the de Bruijn graph and s is an arbitrary node in H
Ensure: A graph formed after merging the bubbles
 1: Set Q be a queue with one node s;
 2: while Q ≠ ∅ do
 3:    u = dequeue(Q);
 4:    for each child v of u do
 5:       if visited[v] = false then
 6:          Set π(v) = u;  /* set u as v's parent in the BFS tree */
 7:          Set visited[v] = true;
 8:          enqueue(Q, v);
 9:       else
10:          Find the lowest common ancestor c of u and v by π();
11:          if the paths c → u and c → v are similar enough then
12:             Merge the two paths and keep the path with the highest path
                weight;
13:          end if
14:       end if
15:    end for
16: end while
```

Slika 2.18: Algoritam turneje

[1] str. 144, slika 5.21

Slika 2.19 ilustruje korake algoritma turneje. Originalni De Brujinov graf je prikazan na slici 2.19(a). Počevši od čvora r, *BFS traversal* se izvodi kako bi se posetili svi potomci od r u rastućem poretku rastojanja od r. *BFS traversal* se zaustavlja kada dođe do posećivanja već posećenog čvora. Slika 2.19(b) prikazuje BFS stablo (podebljano) kada je čvor v ponovo posećen od strane čvora u. Identifikovanjem najnižeg zajedničkog pretka c u BFS stablu, pronalaze se 2 putanje $c \to u$ i $c \to v$ koje formiraju mehurić. Vrši se spajanje 2 putanje i zadržava se putanja sa većom težinom. Nakon formiranja mehurića, dobija se slika 2.19(c). Zatim se BFS nastavlja. Nakon što BFS poseti u', ponovo posećuje v'. c' je najniži zajednički predak od u' i v'. Pronalaze se 2 putanje $c' \to u'$ i $c' \to v'$ koje formiraju mehurić. Nakon otklanjanja mehurića, dobija se slika 2.19(d). Zatim, ne može se pronaći više nijedan mehurić i algoritam turneje se završava.



Slika 2.19: Ovaj primer ilustruje kako algoritam turneje obilazi De Brujinov graf. Radi jasnoće, svaki čvor pokazuje poslednju bazu svojih k-mera i odgovarajući celobrojni stepen je njegova mnogostrukost. Algoritam izvodi pretragu u dobinu (BFS) i BFS stablo je prikazano podebljanim ivicama. Na slici (a) postoje 2 (ugnježdena) mehurića. Na slici (b) se izvodi BFS počevši od r. Kada se poseti u, dete v od u je posećeno. Identifikuje se mehurić i vrši se njegovo spajanje. Zatim, dobija se slika (c). Na slici (c) se nastavlja izvođenje BFS algoritma. Kada se posete u', dete v' od u' je posećeno. Identifikuje se drugi mehurić i vrši se njegovo spajanje. Zatim se dobija slika (d). Na slici (d) se nastavlja izvođenje BFS algoritma. Kako se ne može identifikovati više nijedan mehurić, algoritam se završava.

[1] str. 145, slika 5.22

Nakon što se uklone vrhovi i spoje (otklone) mehurići u De Brujinovom grafu, može se dalje filtrirati buka uklanjanjem k-mera sa mnogostrukošću manjom ili jednakom nekom pragu m (npr. $m = 1$). Na primer na slici 2.19(d), ako je $m = 1$, treba ukloniti 2 ivice težine 1. Ova tehnika se takođe koristi pri korigovanju grešaka u sekvencioniranju. Ukratko, predstavljena su 3 trika: (1) otklanjanje vrhova, (2) spajanje mehurića i (3) filtriranje k-mera sa niskom mnogostrukošću. Kombinovanjem ovih tehnika, dobija se algoritam koji može da obradi greške sekvencioniranja i prikazan je na slici 2.20:

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 2.20: Algoritam za otklanjanje grešaka sekvencioniranja

[1] str. 146, slika 5.23

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 2.20 za različite k. Zatim se kontige grupišu i spajaju. Jedan od problema sa ovim jednostavnim rešenjem je što kontige dobijene za različito k imaju drugačiji kvalitet. Kontige dobijene od H_k gde je k malo su veoma tačne. Međutim, takve kontige su kratke, jer postoji veliki broj grana zbog ponovaka. Kontige dobijene od H_k gde je k veliko su duže. Međutim, one mogu sadržati mnogo grešaka.

IDBA asembler[17] 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, buka u H_k je redukovana. Na ovaj način se za veliko k omogućava dobijanje dugih kontiga visokog kvaliteta iz H_k. Slika 2.21 opisuje ideju IDBA asemblera:

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

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 2.21: IDBA

[1] str. 146, slika 5.24

Glava 3

Opis implementacije algoritama i rezultati njihovog pokretanja

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.

3.1 Opis JellyFish algoritma

Ulaz u glavnu funkciju:

- Z je skup od N k-mera, koji je u Elixir-u predstavljen listom stringova
- α je faktor opterećenja koji kontroliše veličinu heš tabele i koji je broj tipa float između 0 i 1
- h je heš funkcija koja k-mere, tj. stringove mapira u brojeve

Heš funkcija koja nosi naziv *pattern_to_number* prikazana je na listingu 3.1:

```
1 def pattern_to_number(_pattern, begin_string, index,
    length_of_pattern) when index == length_of_pattern do
2   String.to_integer(begin_string, 10)
3 end
4
5 def pattern_to_number(pattern, begin_string, index,
    length_of_pattern) do
6   begin_string = begin_string <> symbol_to_number(String.at(pattern
    , index))
7   pattern_to_number(pattern, begin_string, index + 1,
    length_of_pattern)
8 end
```

Listing 3.1: Heš funkcija *pattern_to_number*

Argumenti funkcije su string koji predstavlja k-mer (*pattern*), početni string koji je uvek prazan pri pozivu funkcije (*begin_string*), indeks koji je uvek 0 pri pozivu funkcije (*index*) i dužina k-mera (*length_of_pattern*). Ova funkcija je rekurzivna i poziva se za svaki karakter stringa *pattern*, počevši od nulte pozicije, sve dok se ne prođe kroz ceo string, tj. dok se *index* ne izjednači sa *length_of_pattern*. Dakle, uzima se prvi karakter *pattern*-a, kodira se pomoćnom funkcijom *symbol_to_number*, prikazanom na listingu 3.2 i nadovezuje se na *begin_string* pomoću operatora za nadovezivanje stringova <>. Zatim se prelazi na sledeći karakter, ponavlja se postupak i sve tako dok se ne dođe do kraja *pattern*-a. U tom trenutku, povratna vrednost funkcije je *begin_string* koji umesto A, C, G i T sadrži kodove 00, 01, 10, 11, respektivno i koji se pomoću funkcije *to_integer/2* iz modula **String** pretvara u ceo broj.

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

Listing 3.2: Pomoćna funkcija *symbol_to_number*

Glavna funkcija pre poziva pomoćnih funkcija obavlja početnu inicijalizaciju objekata koji su joj potrebni, tj. izračunava veličinu heš table H i *Count* tabele, a zatim ih inicijalizuje praznim stringovima i nulama, respektivno. Izgled glavne funkcije može se videti na listingu 3.3:

```
def jellyfish_algorithm(z_table, alpha) do
  n = Kernel.length(z_table)
  size_of_H = Kernel.round(Float.ceil(n/alpha))
  h_table = build_h_table(%{}, size_of_H)
  count_table = build_count_table(%{}, size_of_H)
  {h_table, count_table} = calculate(h_table, count_table, z_table,
    size_of_H)
end
```

Listing 3.3: Glavna funkcija *jellyfish_algorithm*

Funkcije za inicijalizaciju H tabele (*build_h_table*) i Count tabele (*build_count_table*), a koje su predstavljene mapama *h_table* i *count_table*, identične su i imaju iste ulazne parametre. Jedina razlika je što su vredosti mape *h_table* na početku prazni stringovi, a mape *count_table* nule, što se može videti na listingu 3.4:

```
def build_h_table(h_table_empty, 0), do: h_table_empty
def build_h_table(h_table_empty, size_of_H) do
    size_of_H = size_of_H - 1
    h_table_empty = Map.put(h_table_empty, size_of_H, "")
    build_h_table(h_table_empty, size_of_H)
end

def build_count_table(count_table_empty, 0), do: count_table_empty
def build_count_table(count_table_empty, size_of_count_table) do
    size_of_count_table = size_of_count_table - 1
    count_table_empty = Map.put(count_table_empty,
   size_of_count_table, 0)
    build_count_table(count_table_empty, size_of_count_table)
end
```

Listing 3.4: Funkcije za inicijalizaciju H i Count tabela

Pomoćna funkcija koja obavlja brojanje k-mera je funkcija *calculate* čiji su argumenti mape *h_table* i *count_table*, lista *z_table* i veličina pomenutih mapa *size_of_H*. Kod ove funkcije prikazan je na listingu 3.5:

```
def calculate(h_table, count_table, [], _size_of_H), do: {h_table, count_table}
def calculate(h_table, count_table, [head|tail], size_of_H) do
  i = hashEntry(h_table, head, size_of_H)
  if Map.get(h_table, i) == "" do
    h_table = Map.put(h_table, i, head)
    count_table = Map.put(count_table, i, 1)
    {h_table, count_table} = calculate(h_table, count_table, tail, size_of_H)
  else
    count_table = Map.put(count_table, i, Map.get(count_table, i) + 1)
    {h_table, count_table} = calculate(h_table, count_table, tail, size_of_H)
  end
end
```

Listing 3.5: Pomoćna funkcija *calculate*

Kao što se može videti, funkcija *calculate* je rekurzivna. Poziva se za svaki element liste *z_table*, počevši od njene glave, sve dok lista ne postane prazna. Za svaki string iz *z_table* prvo se poziva funkcija *hashEntry* koja vrši heširanje stringova u indeks *i* koji ključ za mape *h_table* i *count_table*. Kod ove funkcije dat je listingom 3.6:

```
def hashEntry(h_table, z, size) do
  i = rem(pattern_to_number(z, "", 0, String.length(z)), size)
  calculate_i(h_table, z, i, size)
end
```

Listing 3.6: Pomoćna funkcija *hashEntry*

Zatim se proverava da li je vredost ključa *i* u mapi *h_table* prazan string. Vrednost ključa neke mape se u Elixir-u dohvata pomoću funkcije *get/2* iz modula **Map** čiji su argumenti naziv mape i ključ. Ukoliko je vrednost prazan string, ključu se dodeljuje vrednost stringa koji se trenutno obrađuje i ključu u mapi *count_table i* se dodeljuje vrednost 1. Vrednost ključa nekoj mapi se dodeljuje pomoću funkcije *put/3* pomenutog modula. Njeni argumenti su naziv mape, ključ i vrednost koja se dodeljuje. Ako je vrednost ključa *i* u mapi *h_table* različita od praznog stringa, onda se samo vrednost ključa *i* u mapi *count_table* povećava za jedan i prelazi se na obradu sledećeg stringa iz liste *z_table*.

Na kraju, izlaz iz glavne funkcije je uređeni par $(h\_table, count\_table)$. Kao što je već pomenuto, u Elixir-u su predstavljeni mapama veličine koja je količnik dužine liste z_table i faktora opterećenja *alpha*. Obe mape imaju iste ključeve, dok su im vrednosti stringovi koji predstavljaju k-mere i brojevi koji predstavljaju broj pojavljivanja određenog stringa u listi z_table.

Na listingu 3.6 se može videti konkretan primer pokretanja algoritma:

```
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,
  4 => 1,
  5 => 0,
  6 => 0,
  7 => 1,
  8 => 0,
  9 => 2,
  10 => 3,
  11 => 0
}}
```

Listing 3.7: Primer pokretanja *JellyFish* algoritma

Ovo je primer izvršavanja algoritma u komandnom promptu. 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.

Glava 4

Zaključak

Literatura

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

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.