SVEUČILIŠTE U ZAGREBU

**FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA**

Projekt iz kolegija Bioinformatika 2017./2018.

**A representation of a compressed**

**de Bruijn graph for pan-genome**

**analysis that enables search**

Bruna Anđelić, 0036478714

Matija Čavrag,

Luka Kovačić, 0036479914

Zagreb, siječanj 2018.

# Uvod

Za što općenito služi de Brujin graf i koji je bio zadatak ovog projekta

# Strukture podataka i funkcije

U ovom će se poglavlju dati kratki pregled struktura i funkcija korištenih u ovom programskom rješenju. Za početak bitno je napomenuti formate ulaznih podataka. Ulazne datoteke koje su korištene u implementaciji primaju se u FAST formatu, odnosno sekvence ulazne datoteke odvojene su komentarima koji počinju oznakom „>“. U ovoj programskoj implementaciji pri čitanju datoteke na kraju svake sekvence stavlja se oznaka za kraj sekvence(u ovoj implementaciji znak „%“), a nakon što se pročita zadnja sekvenca stavlja se oznaka za kraj niza(u ovoj implementaciji znak „\0“). Abeceda korištena u ovoj implementaciji su redom od leksikografski najmanjeg do najvećeg : \0, %, A, C, G, T.

## 2.1 Struktura podataka

U sklopu ovog projekta implementirana su 2 algoritma, jedan za konstrukciju bit vektora i drugi za konstrukciju implicitnog grafa. Budući da 2. algoritam koristi 1., možemo reći da je konačni rezultat izvođenja oba algoritma implicitni graf. Implicitni graf pisan je vektorom instanci tipa *Node*, koji je struktura definirana u programskom rješenju. Struktura *Node* sastoji se od duljine intervale niza kojeg čvor predstavlja(*len*), indeksa lijeve granice intervale promatranog niza (*lb*, niz se nalazi u intervelu [lb .. lb+size-1]), duljine intervala (*size*) te indeksa lijeve granice intervala (*suffix\_lb*)*.*

Prikaz strukture slijedi u nastavku:

struct Node {

uint64\_t len;

uint64\_t lb;

uint64\_t size;

uint64\_t suffix\_lb;

};

## 2.2 Funkcije

Osnovne funkcije korištene u ovom programskoj implementaciji opisani su u nastavku, a pretežno su ostvarene korištenjem gotovih knjižnica da bi se ostvarila što bolje optimizacija. Sve potrebne dorade nad rješenjima gotovih knjižnica također su opisane u nastavku.

### 2.2.1 SA (Sufiksno polje)

Generiranje sufiksnog polja u ovoj se programskoj implementaciji ostvaruje korištenjem gotove knjižnice za izradu sufiksnog polja. Sufiksno polje predstavlja sortirano polje indeksa svih sufiksa ulaza. Sufiksi su poredani od leksikografski najmanjeg do leksikografski najvećeg (tako je primjerice uvijek prvi član SA indeksa znaka '\0').

### 2.2.2. LCP (Longest common prefixes)

Za generiranje LCP polja također se koristi gotova knjižnica za izradu LCP polja, ali gotova implementacija nema jednak oblik kao onaj opisan u pseudokodu osnovne implementacije koji se koristi u ovom projektu. Iz tog se razloga radi dorada rezultata te se na kraj LCP polja dodaje -1. Budući da se rezultat gotove implementacije ne može direktno mijenjati rezultat dodavanja sprema se u novi vektor nazvan *lcpfull*. LCP polje općenito sadrži duljine najdužih zajedničkih prefiksa međusobno uzastopnih sufiksa iz SA polja.

### 2.2.3. BWT (Burrows-Wheeler transformacija)

Za generiranje BWT također je korištena gotova knjižnica no prvo se ulazni niz trebao promijeniti tako da su, kao što je ranije napomenuto, sekvence odvojene znakom „%“, a kraj naznačen znakom „\n“. Unutar pseudokoda naznačeno je da se indeksi BWT-a kreću od 1, ali gotova knjižnica koristi indekse od 0 pa se umjesto pomicanja cijelog polja udesno korigiralo korištenje indeksa danog pseudokoda. BWT je općenito reverzibilna transformacija koja mijenja raspored simbola u nizu te ne čuva dodatne podatke. Koristi se u oba algoritma, a općenito vrijedi da je BWT[i] = s[SA[i]-1].

### 2.2.4. C-mapiranje

Za ostvarivanje C mapiranja nije korištena gotova implementacija već je C polje izgrađeno unutar programske implementacija. Općenito u C polju za za svaki znak abecede postoji informacija o tome koliko se ukupno puta u nizu pojavljuju znakovi koji su leksikografski strogo manji od znaka čiju vrijednost gledamo.

### 2.2.5. WT (Wavelet tree)

Za konstrukciju WT unutar ovog programskog rješenja korištena je gotova knjižnica. Wavelet tree općenito je struktura podataka koja se koristi za komprimirano spremanje nizova. Listovima ovog stabla odgovaraju znakovi abecede koja je korištena, a u svakom čvoru spremljen je bit vektor koji govori kojem podskupu pripada znak niza. Općenito stablo rekurzivno dijeli korištenu abecede u parove podskupove.

# 3. Algoritam1: Konstrukcija bit vektora

## 3.1 Opis implementacije algoritma

Prvi algoritam koji je ostvaren u ovom projektu koristi se za konstrukciju 2 bit vektora, Bl i Br. Funkcija nazvana *createBitVectors* kao ulazne parametre prima k, BWT, adresu od implicitnog grafa *graph*, adresu reda Q te adresu bit vektora Bl i Br. Ulazni parametar k je cijeli broj koji predstavlja duljinu k-mera koja će se koristiti, BWT je niz znakova koji predstavlja Burrows-Wheeler transformaciju objašnjenu u prethodnom poglavlju. Ostali parametri pokazuju na podatke koji se u algoritmu koriste budući da je funkcija korištena za implementaciju ovog algoritma tipa void. Općenito bit vektor Br koncipiran je tako da je vrijednost njegove i-te pozicije 1 ako je S[SA(i), SA(i)+k-1] najveći desno ponavljajući niz te je S[SA(i)] leksikografski najveći ili najmanji sufiks početnog niza takav da mu je S[SA(i), SA(i)+k-1] sufiks. Kod bit vektora Bl situacija je malo drugačija pa bit vektor Bl na i-toj poziciji ima 1 ako i samo ako S[SA(i), SA(i)+k-1] nije najveći desno ponavljajući niz te je S[SA(i)] leksikografski najveći sufiks početnog niza kojem je S[SA(i), SA(i)+k-1] prefiks. Budući da je ulaz u ovu funkciju BWT, u *main* funkciji izračunava se prvo sufiksno polje SA koje je potrebno za izračunavanje vrijednosti BWT koja se koristi u danjem računu. Također u *main* dijelu programa računa se i BWT koji se koristi u ovom algoritmu. U ovoj programskoj implementaciji sama funkcija koja ostvaruje algoritam1 poziva se iz funkcije za ostvarivanje algoritma2. Na početku algoritma izračunavaju se WT te LCP korištenjem gotove knjižnice. Također računa se vektor C za sve znakove te se kreira vektor *lcpFull* iz glavnog vektora kako je opisano u prethodnom poglavlju. U ostatku algoritma pojavljuju se dvije for petlje. Prva for petlja kreće se kroz sve vrijednosti promijenjenog LCP vektora. Ako se 2 uzastopna sufiksa podudaraju u više od k znakova, odnosno ako je vrijednost LCP veća od k to nam je znam da u idućim koracima možemo dobiti vrijednost 1 za Bl ili Br te se taj slučaj prati podizanjem zastavice *open*. Ako je vrijednost LCP-a u nekom koraku točno k, odnosno 2 uzastopna sufiksa podudaraju se u točno k znakova prati se indeks varijablom *kIndex* jer postoji mogućnost da je vrijednost Br(i) jednaka 1. Ako ovo nije slučaj, odnosno 2 uzastopna sufiksa se podudaraju u manje od k znakova, provjeravamo možemo li dodati jedinice u bit vektore. Ako je trenutna vrijednost *kIndex* veća od vrijednosti lijeve granice, odnosno indeksa najmanjeg sufiksa koji se podudara sa svojim sljedbenicima u najmanje k znakova, postavljamo vrijednost Br na 1 za lijevu i desnu granicu. Ovdje također počinje i izgradnja implicitnog grafa kojem postavljamo početne čvorove. Također u red Q dodajemo indeks čvora. Zatim se provjerava može li se postaviti vrijednost bit vektora Bl na 1. To se radi ako je najveći indeks u kojem se razlikuju 2 uzastopna BWT znaka veći od lijeve granice te tada sve vrijednosti Bl(C[BWT[j]]), gdje se j kreće od lijeve granice pa sve do trenutne vrijednosti i,postavljamo na 1 ako trenutna vrijednost znaka nije znak % ili \0. Na kraju petlje provjerava se je li neki od indeksa za kojeg je vrijednost bit vektora Bl 1 ujedno i indeks najveće desno ponavljajućeg niza te ako je postavljamo vrijednost vektora Bl za taj indeks u 0.

## 3.2 Primjer rezultata

# 4. Algoritam 2: Konstrukcija implicitnog grafa

## 4.1 Opis implementacije algoritma

Rezultat drugog algoritma je konstruirani implicitni graf koji služi za reprezentaciju de Bruijnovog grafa. Za konstrukciju implicitnog grafa koriste se bit vektori dobiveni algoritmom 1 opisanom u prethodnom poglavlju. Kao što je naznačeno algoritam 1 osim konstrukcije bit vektora dodaje i inicijalni čvor u graf. U ovom algoritmu potrebno je najprije odrediti koliko je konačni broj čvorova te dodati završne čvorove u implicitni graf. Graf je kako je već napomenuto vektor elemenata tipa *Node* čija je struktura objašnjena u poglavlju 2.1. Funkcija u kojoj je u ovom projektnom rješenju ostvarena implementacija ovog algoritma naziva se *createCompressedGraph* te prima 3 ulazna parametra k, BWT i zastavica ispisa. Značenje i vrijednosti parametara k i BWT jednake su kao u algoritmu 1 te su opisani u prethodnom poglavlju. Posljednji parametar služi kao zastavica te ako je njena vrijednost jednaka *originalPrint* na ekran se ispisuje prve 2 vrijednosti rezultata, ako je njena vrijednost jednaka *lastYearPrint* na ekran se ispisuje sve 4 vrijednosti rezultata. Ako ne postoji niti jedna od vrijednosti zastavica rezultat se ne ispisuje na ekran već se sve 4 vrijednosti ispisuju u datoteku. Posljednji parametar primarno se koristi u svrhu testiranja programskog koda. Sam algoritam računa intervale sufiksnog polja za sve desno maksimalne podnizove duljine k. Nakon toga za svaki od tih podnizova koje možemo označiti kao *w* računa *cw* intervale sufiksnog polja u kojem *c* predstavlja element abecede korištene u ovom programskom rješenju. Koristi se sada niz *u* koji predstavlja vrijednost *cw* te se računa *cu* interval te tako dalje redom. Cijela procedura tako započinje sa svim desno maksimalnim podnizovima duljine k te ih proširuje koliko god može, na sve moguće načine, znak po znak. Kao što je naznačeno na početku, algoritam 1 pronalazi sve desno maksimalne podnizove duljine k te dodaje početne čvorove u graf G koji predstavljanju njihove reprezentacije, a također se u red Q dodaju identifikatori za pojedini dodani čvor. Algoritam 2 koji gradi implicitni graf redom uzima sve te čvorove prema indeksima pohranjenim u redu Q te ih proširuje ulijevo. Funckija koja je korištena za pronalazak *cw* intervala naziva se *intervals\_symbols* te je to gotova funkcija iz knjižnice sdsl-lite. Zatim se za svaki niz *cw* dalje testira je li njegov prefiks duljine *k* desno-maksimalni ponavljajući niz. Provjera se izvršava korištenjem desnog bit vektora Br u kojem su označene granice sufiksnih intervala snih desno maksimalno ponavljajućih nizova. Ako test pokaže da taj prefiks je desno maksimalan, račun ovdje staje te algoritam prelazi na idući čvor koji je određen idućom vrijednosti reda Q. Za slučaj kada funkcija *intervals\_symbols* pronađe više od jednog znaka c, niz *w* je lijevo maksimalni podniz te se u tom slučaju u graf dodaje novi čvor koji reprezentira prefiks duljine k niza *cw* s kojim se dalje nastavlja izvođenje algoritma. Zadnji slučaj, ako *w* nije lijevo maksimalni podniz, njegov pripadni čvor se proširuje, odnosno povećava se vrijednost duljine niza te se nastavlja istraživanje pronađenog *cw* intervala.

## 4.2 Primjer rezultata

# 5. Analiza rezultata

Ovdje oni lukini grafovi za naš i originalan i usporediti koji je brži bolji i to, kolike su razlike blabla

# 6. Zaključak

Kako je prošlo testove, možemo zaključiti da je brutalno i da smo sve riješili i amen

# 7. Literatura

[1] Timo Beller and Enno Ohlebusch. A representation of a compressed de bruijn

graph for pan-genome analysis that enables search. Algorithms for Molecular

Biology, 11(1):20, 2016.