Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Matija Rezar

Sestavljanje genoma iz odčitkov zaporedja

magistrsko delo

študijski program druge stopnje
računalništvo in informatika

Mentor: dr. Andrej Brodnik

Ljubljana, 2016

University of Ljubljana
Faculty of Computer and Information Science

Matija Rezar

Genome assembly from sequence reads

Master's Thesis

2nd cycle Masters Study programme
Computer and Information Science

Supervisor: dr. Andrej Brodnik

Ljubljana, 2016

Povzetek

V grobem lahko postopek sestavljanja genoma opišemo kot iskanje Eulerjevih obhodov v de Bruijnovih grafih. V nalogi povzamemo teorijo podatkovnih struktur za predstavitev (de Bruijnovih) grafov in opišemo nekaj implementacij. Nato predstavimo kBWT, novo deterministično podatkovno strukturo za predstavitev de Bruijnovih grafov, ki uporablja skoraj optimalnih $n \cdot \sigma + o(n)$ bitov pomnilnika, kjer je n število k-merov v grafu in σ velikost abecede. Podatke o sosednosti vozlišča lahko v njej najdemo v $\Theta(\sigma \cdot k)$ časa. Strukturo primerjamo z obstoječo podatkovno strukturo ogrodja GATB, ki je osnovana na Bloomovih filtrih. Preizkusi kažejo, da je deterministični kBWT pričakovano počasnejši od podatkovne strukture ogrodja GATB. Pokazalo pa se je, da kBWT bolj učinkovito izrablja predpomnilnik, vendar to ni nadomestilo porabe procesorskega časa.

Ključne besede: genom, sestavljanje genoma, teorija grafov, de Bruijnov graf, Eulerjev obhod.

Abstract

Title: Genome assembly from sequence reads

Assembling genomes can be roughly described as finding Eulerian tours in de Bruijn graphs. We explain the theory behind (de Bruijn) graph data structures and describe some of the implementations. Then we present kBWT, a new space-efficient deterministic data structure for storing a de Bruijn graph, which uses near-optimal $n \cdot \sigma + o(n)$ bits of memory, where n is the number of k-grams in the graph and σ is the size of the alphabet. It can retrieve neighborhood information for a given node in $\Theta(\sigma \cdot k)$ time. We also compare it to an existing data structure found in the GATB framework, which is based on Bloom filters and therefore probabilistic. Benchmarks of the deterministic kBWT show it is slower in practice, compared to GATB's data structure. Evaluation showed kBWT had better cache efficiency, which did not make up for the number of processor cycles used for executing the algorithm.

Keywords: genome, genome assembly, graph theory, de Bruijn graph, Eulerian tour.

Zahvaljujem se svoji družini, ki me je pri študiju podpirala, svojemu mentorju, ki sem mu življenje skrajšal vsaj za kakšno leto, Matevžu ker mi je pomagal pri bolj nerodnih stavkih tikparskin napakah, in vejicah ter Gašperju za splošno moralno podporo.

Contents

Sestavljanje genoma iz odčitkov zaporedja

Uvod

Genom je zaporedje, ki opisuje razvoj in lastnosti živih bitij. Obstoječe tehnologije ne omogočajo, da bi genom prebrali od začetka do konca v enem kosu, zato ga beremo po delih. Slika 1.1 na strani 3 prikazuje postopek branja in sestavljanja. Dano DNK molekulo (Slika 1.1a) večkrat kopiramo (Slika 1.1b). Nato kopije razbijemo na krajše odseke (Slika 1.1c). Ti odseki so dovolj kratki, da jih lahko preberemo v t.i. odčitke (Slika 1.1d).

Odčitke sedaj s pomočjo računalnika sestavimo v zaporedje, ki predstavlja celoten genom. Najprej odčitke presejemo in očistimo: odstranimo nizko-kvalitetne odčitke, odrežemo nizko-kvalitetne dele odčitkov, itd. Nato med odčitki poiščemo prekrivanja (Slika 1.1e) in jih glede na ujemanje poravnamo. Za predstavitev prekrivanj se navadno uporabljajo grafi. V preteklosti je bil to graf prekrivanja [26], danes pa se uporablja de Bruijnov graf. Graf lahko prebolikujemo na različne načine, ki običajno poskušajo popraviti napake, ki so se zgodile v postopku branja (za kratek opis postopkov preoblikovanja glej stran 34). Na koncu vse skupaj združimo v eno predstavitev celotnega genoma (Slika 1.1f).

De Bruijnov graf je definiran kot usmerjen graf, kjer so vozlišča označena z nizi dolžine k. Povezave potekajo od vozlišč s *pripono* dolžine $k-1$, proti vozliščem z enako *predpono* dolžine $k-1$. Oznake povezav so zadnje črke ponornih vozlišč.

Iz genoma lahko sestavimo delen de Bruijnov graf tako, da najprej v genomu poiščemo vse podnize dolžine k, imenovane tudi k-meri, ki se v njem pojavljajo. Vrednost k moramo izbrati tako, da se vsak podniz dolžine $k+1$ v genomu, ki je navadno veliko daljši kot k, pojavi največ enkrat[1]. To pomeni, da vsaka povezava v grafu predstavlja natanko en znak v genomu. Nato iz k-merov naredimo vozlišča in jih povežemo kjer se pripone in predpone ujemajo. Genom sedaj predstavlja Eulerjev obhod grafa. Če sledimo povezavam v enakem vrsten redu, kot si njihove oznake sledijo ena drugi v genomu, bomo obšli vse povezave. Intuicija postopkov za sestavljanje gre v nasprotni smeri. Iz odčitkov razberemo, kateri k-meri se pojavljajo v genomu, nato iz njih zgradimo graf in v njem poiščemo Eulerjev obhod, torej izvorni genom.

Velikosti genomov so različne. Virusni so sestavljeni iz nekaj tisoč baznih parov (bp), medtem ko rastlinski dosegajo tudi več kot sto milijard baznih parov (tudi do 150 Gbp). Človeški genom je sestavljen iz 3 do 4 Gbp, med tem ko se bakterijski gibajo okrog 5 Mbp.

Posledično graf genoma, kot je na primer človeški, vsebuje približno 3 milijarde vozlišč. Če vsako vozlišče predstavimo kot svoj objekt, za to potrebujemo $2 \cdot k$ bitov za predstavitev njegovega k-mera in $8 \cdot 64$ bitov za predstavitev njegovih povezav. Za $k = 31$ človeški genom tako zaseda $(2 \cdot 31 + 8 \cdot 64) \cdot 3 \cdot 10^9$ bitov ali okrog 72 GB. Že to je preveč za glavni pomnilnik običajnega računalnika, za kvalitetno sestavljanje pa potrebujemo še dodatne podatke o vozliščih, kot so na primer frekvence k-merov v vhodnih podatkih. Očitno je, da za sestavljanje potrebujemo bolj učinkovite podatkovne strukture.

De Bruijnov graf

De Bruijnov graf je usmerjen graf, kjer so vozlišča nizi dolžine k sestavljeni iz znakov neke abecede. Na primer, za abecedo $\Sigma = (\texttt{A}, \texttt{C}, \texttt{T}, \texttt{G})$, velikosti

[1]Zaradi nadaljnih postopkov želimo imeti graf in ne hipergrafa.

$|\Sigma| = \sigma = 4$, ki se uporablja pri genomih, in $k = 2$ so vozlišča regularnega de Bruijnovega grafa označena z:

$$V = \{\texttt{AA}, \texttt{AC}, \texttt{AT}, \texttt{AG}, \texttt{CA}, \texttt{CC}, \texttt{CT}, \texttt{CG}, \texttt{TA}, \texttt{TC}, \texttt{TT}, \texttt{TG}, \texttt{GA}, \texttt{GC}, \texttt{GT}, \texttt{GG}\} \quad .$$

Kasneje bomo spoznali še delni de Bruijnov graf. Na Sliki 2.1 na strani 10 je primer regularnega de Bruijnovega grafa za abecedo $\Sigma = \{0, 1\}$ in $k = 2$.

Kot smo omenili prej, povezave grafa tečejo od vozlišč z neko pripono dolžine $k-1$ proti vozliščem z enako predpono dolžine $k-1$. Oznake povezav so zadnji znaki ciljnih vozlišč. Na primer, iz vozlišča ACAGT poteka povezava v vozlišče CAGTG in je označena z znakom G. Če poznamo oznako nekega vozlišča in oznako izhodne povezave, lahko vnaprej ugotovimo, kakšna bo oznaka vozlišča, do katerega bomo prišli po dani povezavi. Graf je regularen (vsa vozlišča imajo enako vhodno in izhodno stopnjo), saj imajo vsa vozlišča natanko σ vhodnih in izhodnih povezav.

Eulerjev obhod takega grafa nam omogoča, da sestavimo niz iz dane abecede, ki kot podnize vsebuje vse nize dolžine $k + 1$. Tak niz je dolg σ^{k+1}, če ga obravnavamo kot krožen niz, oziroma $k + \sigma^{k+1}$, če želimo običajen niz.

Za namene sestavljanja genomov uporabljamo delni de Bruijnov graf. Kot vozlišča uporabimo vse k-terice, ki jih najdemo v vhodnih podatkih. Povezave ustvarimo na enak način, kot pri regularnem de Bruijnovem grafu. Na Sliki 2.2 na strani 11 je primer delnega de Bruijnovega grafa za genom CAGGAGGATTA. Opazimo lahko, da manjka večina od 256-ih vozlišč, ki bi bila prisotna v regularnem de Bruijnovem grafu za $k = 4$.

Razlog za uporabo de Bruijnovega grafa je naslednji: če bi imeli polni genom in bi iz njega sestavili de Bruijnov graf, bi nam genom predstavljal Eulerjev obhod grafa. Če gremo v nasprotno smer, torej sestavimo graf iz odčitkov genoma in poiščemo obhod tega grafa, bi morali dobiti nazaj genom. V praksi le redko dobimo Eulerjev obhod in iz grafa lahko sestavimo le dele genoma. Vseeno je bila ta ideja dobra odskočna deska za vse sodobne sestavljalnike.

Eulerjev obhod

Ideja Eulerjevega obhoda je pomembna za sestavljanje genomov, zato smo raziskali algoritme za iskanje Eulerjevih obhodov v usmerjenih grafih.

Zaporeden algoritem je znan že zelo dolgo. Deluje tako, da poišče osnoven obhod, nato pa iz preostalih povezav sestavlja obhode in jih priključuje obstoječemu, dokler niso vse povezave porabljene. Ideja vzporednega algoritma je podobna. Najprej poišče povezavno-disjunktne cikle, ki jih nato združi v en obhod.

Zanimiva težava se pojavi, ko si dva cikla v grafu delita vozlišče. Za namen Eulerjevega obhoda ni pomembno v kakšnem vrstnem redu obiščemo ta dva cikla. Pri genomu pa je pomembno kakšno je zaporedje, saj le eno zaporedje da pravilno rešitev. To lahko rešimo tako, da pogledamo kam se v grafu usmerjajo odčitki. Odčitek lahko preslikamo v pot v grafu, glede na zaporedje k-merov v njem. Če ta pot zavije na neko vejo v grafu, mora tako pot ubrati tudi Eulerjev obhod, iz katerega nameravamo sestaviti genom.

Ker se pri branju genoma dogajajo napake, se lahko zgodi, da graf ni Eulerjevski. V tem primeru Eulerjevega obhoda ne moremo najti in moramo genom izluščiti na drugačen način. Metode za reševanje tega problema navadno uporabljajo idejo algoritmov za iskanje Eulerjevega obhoda in iščejo dolge poti v grafu.

Obstoječe rešitve

V splošnem se strukture za predstavitev de Bruijnovih grafov delijo v dve kategoriji: navigacijske podatkovne strukture, ki podpirajo sprehajanje po vozliščih grafa, in slovarske podatkovne strukture, ki omogočajo poizvedbe o prisotnosti vozlišč v grafu. Mogoče je pokazati, da so slovarske podatkovne strukture tudi navigacijske. Definicija de Bruijnovega grafa omogoča ugibanje sosednjih vozlišč, katerih prisotnost lahko preverjamo v slovarski strukturi.

Najbolj preprost način predstavitve grafa je običajen objekt ali C-jevska eksplicitna struktura, ki vsebuje niz za opisovanje k-terice ter 8 referenc na sosednja vozlišča. Tak zapis je prostorsko relativno neučinkovit in ni primeren za večje genome, saj je prostor, ki ga po eni strani zasedejo reference in po drugi oznaka vozlišča, zelo velik.

Večina grafa bo zgrajena iz preprostih zaporedij vozlišč brez vejitev, ki jih lahko stisnemo v eno dolgo vozlišče. Tak način zapisa grafa uporablja sestavljalnik Velvet [38]. Sestavljalnik SOAPdenovo2 [25] gre korak dlje in namesto enega znaka na povezavo povezavam dodeli daljša zaporedja in s tem prihrani prostor [37].

Učinkovitejšega načina se poslužujeta ogrodje GATB [13] in sestavljalnik Minia [10]. Uporabljata Bloomov filter, ki omogoča hitro in učinkovito preverjanje pripadnosti množici, v kombinaciji z množico lažnih pozitivnih elementov.

Nekateri drugi načini zapisa grafa so osnovani na transformaciji Burrows-Wheeler (BWT) in sorodnem kazalu FM, ki besedilo preslika v obliko, ki je bolj stisljiva, hkrati pa omogoča hitro iskanje po besedilu [16]. Tak način zapisa uporablja implementacija DBGFM [11]. Podobna je tudi struktura opisana v [5], ki je osnovana na sorodni transformaciji XBW.

Na področju praktičnih algoritmov za gradnjo sosesk je vredno omeniti BCALM [11], ki iz odčitkov gradi daljše verige, in Omnitigs [36], ki iz grafa sestavi tako imenovane *varne nize*. Varni nizi so nizi, ki se glede na graf zagotovo pojavljajo v vseh rekonstrukcijah genoma. Omnitigs varne nize išče tako, da sestavlja poti, ki se ne vračajo nazaj na ista vozlišča. Tako se izogne težavam s cikli.

Prostorsko-učinkovita podatkova struktura za predstavitev de Bruijnovega grafa

V nalogi predstavljamo novo podatkovno strukturo, imenovano kBWT. Podatkovna struktura omogoča iskanje po množici k-merov iz besedila v $O\left(k\right)$ času. Pri tem porabi $\sigma \cdot n + o(n)$ bitov prostora, kjer je σ velikost abecede in n število k-merov v grafu.

Genom je besedilo z abecedo $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ velikosti $\sigma = 4$. Besedilo je dolgo nekaj tisoč, do več milijard zankov (odvisno od velikosti genoma).

Pri gradnji strukture najprej k začetku besedila T pripnemo znak `$`, ter koncu dodamo k znakov `$`, da dobimo T'. Nato naredimo množico vseh $k+1$-teric v T'.

Elemente množice uredimo v seznam glede na pripono dolžine k, oziroma, kjer je ta enaka, glede na prvi znak. Nato zaporedja $k+1$-teric, ki imajo enako pripono dolžine k, združimo v eno $k+1$-terico. Prvi znak dobljene $k+1$-terice je iz nove abecede in vsebuje podatke o tem, kateri prvi znaki $k+1$-teric so bili prisotni. To lahko predstavimo kot bitni vektor v velikosti abecede, kjer vsak bit predstavlja en znak. Znak `$` se pretvori v vektor ničel, zaporedje $k+1$-teric, kjer sta prisotna `A` in `T` dobi kot prvi znak sedaj `0101`[2], itd. Sedaj $k+1$-terice predstavljajo vozlišča v grafu. Prvi znak predstavlja vhodne povezave vozlišča, zadnjih k znakov predstavlja oznako vozlišča.

Zadnja sprememba je, da prve znake zaporedij $k+1$-teric, ki imajo enakih srednjih $k-1$ znakov, zapišemo v prvi člen zaporedja. Ostalim prvi znak nastavimo na `0000`. To storimo, ker so vse k-terice, ki so povezane na eno od k-teric v zaporedju, povezane tudi na ostale, torej imajo vozlišča enake sosede glede na vhodne povezave.

Prvi znaki naših $k+1$-teric sedaj tvorijo preoblikovano besedilo. Po njih lahko iščemo z Algoritmom 4.2 na strani 46, ki je podoben algoritmu v [16].

Zaključki

Našo podatkovno strukturo smo primerjali s podatkovno strukturo ogrodja GATB, ki je osnovana na Bloomovih filtrih. Algoritem, ki ga uporablja kBWT, opravi k obhodov zanke. Ob vsakem obhodu mora opraviti dve operaciji *rank*, za kateri je potrebno iz bitnega vektorja rekonstruirati dva

[2]Celoten vektor skupaj jemljemo kot en znak. Predstavitev v obliki bitov je le za boljši prikaz.

bloka. Po drugi strani Bloomov filter izračuna d zgoščevalnih funkcij in poišče teh d bitov v svojem bitnem vektorju. Ker je $d \ll k$ in ker so operacije *rank* zelo drage, se je pričakovano naša podatkovna struktura odrezala slabše.

Pri preizkušanju s Cachegrind-om se je pokazalo, da naša stuktura povzroči manj zgrešitev v predpomnilniku. To je v skladu s pričakovanji, saj kBWT iskanje vedno začenja na dveh od petih mogočih točk, medtem ko Bloomov filter skoraj zagotovo povzroči zgrešitev ob vsakem vpogledu v bitni vektor. Boljša izraba predpomnilnika žal ni nadomestila večje porabe procesorskega časa, ki ga je potreboval kBWT.

Struktura kBWT porabi tudi več prostora, kot struktura ogrodja GATB. Slednja bolj učinkovito izloča nepotrebne k-mere, kar je verjeten razlog za povečano porabo pomnilnika in bi ga tudi veljalo izboljšati. Ob gradnji grafa se želimo znebiti manj pogostih k-merov, saj gre verjetno za napake, hkrati pa ti k-meri zavzemajo prostor. Pri Bloomovih filtrih je to preprosto, saj manj pogostih k-merov enostavno ne dodamo v množico. Pri kBWT se pojavi težava, saj naš algoritem za iskanje zahteva, da so v strukturi prisote vse pripone vseh k-merov. Navadno za pripone danega k-mera poskrbijo predpone ostalih k-merov. Najbolj enostavno je, da obdržimo manj pogoste k-mere za namene iskanja in jih označimo, kar poveča podatkovno strukturo za 1 bit na k-mer. Na žalost s tem ne prihranimo prostora. Druga možnost je, da dodamo vse predpone k-mera, ki ga želimo odstraniti, in jih dopolnimo do dolžine k z znaki `$`. Tudi tu verjetno ne bi prihranili veliko prostora. Zadnja možnost je, da preverimo katerim k-merom je potrebno priskrbeti pripone. To bi bilo računsko zelo zahtevno. Verjetno bi potrebovali nekakšen slovar k-merov, kar pa je natanko problem, ki ga kBWT poskuša rešiti. V naši implementaciji smo redke k-mere obdržali. Posebej smo obravnavali le k-mere na koncu odčitkov, ki so kandidati za k-mere brez prisotnih pripon.

Učinkovita uporaba predpomnilnika običajno močno prispeva k praktični hitrosti delovanja algoritma, zato je strukture osnovane na BWT vredno razvijati naprej. Učinkovitejše strukture omogočajo uporabo bolj kompleksnih algoritmov, kar lahko pripelje do bolj kvalitetno sestavljenih genomov.

Chapter 1

Introduction

DNA molecules are the blueprints for life on Earth [1]. They are found in every organism and control its development and functioning from before it is born, until death. Some parts of DNA encode sequences of amino acids which are assembled into various proteins, the building blocks of living organisms. Other parts of the DNA control which proteins get synthesized and when [3].

The DNA strand consists of a sequence of smaller molecules called nucleotides or base pairs (bp). The precise sequence of nucleotides is unique to each living organism, but many may share large portions of it. Parts that represent common proteins, for instance, are shared even between different species. Other sections, e.g., those controlling cell division, may be specific to a small group, making them more susceptible to changes and consequently cancer. A portion of a genome that encodes a specific function is called a GENE. The complete genetic information of an individual is that individual's GENOME [1]. Genomes vary in size, from ∼2000 bp virus genomes to more than 100 Gbp (100 billion base pair) plant genomes. Human genomes clock in at 3 Gbp to 4 Gbp and bacterial genomes at around 5 Mbp [1, 24].

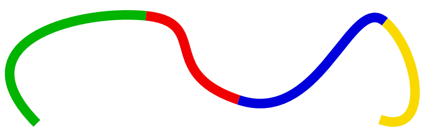
1.1 From genome to genome—the theory

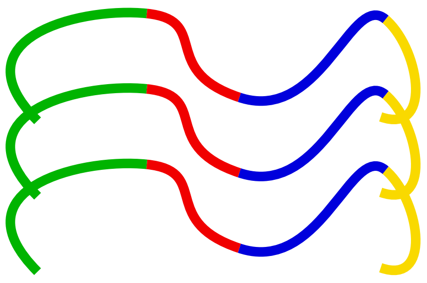
No currently available technology can read a complete genome from start to finish. The best machines may manage a small virus, but most can read around 100 bp at a time. To create a representation of the whole genome, the genome is read piecemeal and then assembled back together. We start with a genome in the form of a DNA molecule (Figure 1.1a). The genome is cloned (Figure 1.1b) tens to hundreds of times, sometimes even more. The clones are then split at various points into short pieces (Figure 1.1c). The overlap of those pieces from multiple copies of the genome allows us to later reassemble the genome, using a computer. The pieces are read, producing so called reads (Figure 1.1d), which the computer can process. Next, we algorithmically align those pieces on the parts that overlap (Figure 1.1e). Finally, we merge the reads into the electronic representation of the genome (Figure 1.1f). This process can be very demanding due to the large amount of data involved.

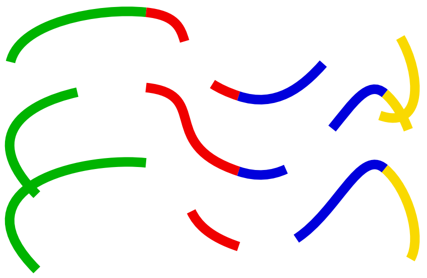
To help with the alignment process, a type of directed graph, called a de Bruijn graph, is used to map the pieces and their relations to each other. A de Bruijn graph's vertices are labeled with fixed-length strings, called k-mers, that were found in the reads of the genome. The edges connect vertices where the last $k-1$ characters of the source vertex label matches the first $k-1$ characters of the destination vertex label. The edges are labeled using the last character of the destination vertex label. Consequently, the label of the destination of an edge can be computed, by appending the edge label to the label of the source vertex.

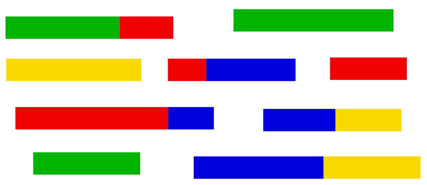
If we construct a de Bruijn graph with all 4^k possible k-mers[1] and find an Eulerian tour in it, the concatenation of the labels of edges along the tour produces the shortest string, that contains every possible $k+1$-mer as a substring. On the other hand, if we construct a de Bruijn graph from k-mers found in a single genome, we also automatically get it's Eulerian tour: it

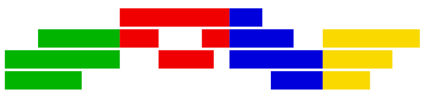
[1]We use 4 here as there are four possible nucleotides.


(a) Original DNA strand.


(b) DNA is cloned.


(c) Clones are divided into fragments.


(d) Fragments are read.


(e) Reads are aligned according to their overlaps.

(f) Genome is reconstructed from the alignment.

Figure 1.1: Illustration of the genome assembly process.

is the walk we traverse by starting in the first k-mer of the genome, then following edges as their labels appear in the genome. The idea of a genome assembly using de Bruijn graphs is thus: if we build a de Bruijn graph from k-mers found in the reads of a genome, we can reconstruct the genome by finding the Eulerian tour of the graph.

1.2 From genome to genome—the practice

Although in theory, the genome in the form of a de Bruijn graph can be reconstructed using an Eulerian tour of the graph, in practice, the data obtained from the sequencer usually contain errors, which deform the graph in such a way that it does not contain an Eulerian tour anymore. Consequently, various techniques have been developed to remove those errors from the graph. They generally can not remove all the errors and often remove non-errors. The most crude methods merely involve removing parts of the graph, that seldom appear in the input data. More sophisticated algorithms look for certain subgraphs in the graph and attempt to transform them to remove the errors.

Another issue of a more technical nature is the sheer amount of data involved. For instance, a human genome multiplied 30 times consists of $30 \cdot 4 \cdot 10^9$ bases. With 2 bits required to encode a base this means $120 \cdot 10^9$ bits or 15 GB of raw data has to be stored in the main memory of a computer. To represent that data as a graph, additional memory is required.

1.3 Our contribution

In the thesis we present a new Burrows-Wheeler transform (BWT) based data structure, which we call kBWT. It is constructed by sorting a set of all k-mers that appear in the genome and indexing them. Due to the way they overlap, they behave in much the same way as rotations in BWT, which allows us to search through them in $\Theta(k)$ time.

We considered implementing our data structure into Velvet [38], but eventually decided to implement it for GATB [13] instead. GATB is a bioinformatics framework designed to be modular and as such presented a much simpler interface for implementing de Bruijn graph data structures.

1.4 Thesis structure

First, we describe the theory behind the algorithms and discrete structures used in genome assembly, specifically: de Bruijn graphs and Eulerian tours in § 2.

In § 3 we look at how genomes are sequenced and the challenges associated with genome assembly. We will approach the subject mostly from the point of view of computer science, but some domain-specific knowledge is required. We briefly describe what genomes are, some terminology used when working with them and how data about them is acquired. We then outline the basic process of genome assembly and show how it combines de Bruijn graphs and Eulerien tours.

In § 4 we present a new data structure used to represent a de Bruijn graph, kBWT. We also perform benchmarks and compare it to the original data structure used in GATB.

Chapter 2

De Bruijn graph and Eulerian tour

A de Bruijn graph is the primary discrete structure used in contemporary genome assemblers. In this chapter we will first define some terms, then we will talk about de Bruijn graphs and describe how they are used in genomics. Lastly we will examine some examples of different data structures used to represent it.

2.1 Definitions

Graph We use the term GRAPH to either mean a DIRECTED GRAPH or specifically a de Bruijn graph, unless explicitly stated.

A directed graph $G\,(V, E)$ consists of a set of VERTICES (also called NODES) V and a set of EDGES E. Each vertex $v \in V$ has a label $\text{label}(v)$. Each edge is an ordered pair (u, v), meaning there is a directed connection going from vertex u to vertex v. Every edge also has a $\text{label}((u, v))$.

We define $\text{indegree}(v) = |\{e \in E \,|\, e = (u, v)\}|$ to be the number of edges going into v and $\text{outdegree}(v) = |\{e \in E \,|\, e = (v, u)\}|$ the number of edges leaving v.

Alphabet An ALPHABET is a finite set of characters that form a TEXT. Further, a text or a STRING is a finite series of characters from an alphabet. We use Σ to denote the alphabet and $\sigma = |\Sigma|$. With Σ^k we denote a set of all strings of length k from an alphabet Σ, also called k-GRAMS. We use k-grams to represent k-mers of the genome. A special case is $\Sigma^0 = \{\epsilon\}$, which represents an empty string. Finally, $\Sigma^* = \bigcup_{k=0}^{\infty} \Sigma^k$ is the set of all texts with alphabet Σ.

Lower case letters from the beginning of the English alphabet $(a, b, c, \dots)$ denote characters from an alphabet, while lower case letters from the end of the alphabet $(w, x, y, \dots)$ denote strings. If we want to concatenate characters, we write them one after another $w = ab$. Same goes for concatenating strings and characters, $x = aw$ or $y = wa$, and concatenating strings: $z = xy = aababa$. Sometimes, to make it clearer, we use $\cdot$ to explicitly denote concatenation, e.g., $a \cdot b$. Operator $\mathrm{len}(w)$ is used to describe the length of a string. Function $\mathrm{prefix_k}(w)$ represents the first k characters of w and $\mathrm{suffix_k}(w)$ the last k characters of w.

In our solution we require the characters of the alphabet to be TOTALLY ORDERED, which means there exists a relation $\leq$ such that the following statements are true:

- $a \leq a$: reflexivity; every element is in relation with itself
- $(a \leq b) \wedge (b \leq a) \implies a = b$: antisymmetry; an element is in both relations only with itself
- $(a \leq b) \wedge (b \leq c) \implies a \leq c$: transitivity; if a is smaller than b and b is smaller than c, then a is smaller than c
- $(a \leq b) \vee (b \leq a)$: totality; two elements are always ordered in one or the other way.

The total order of characters can be used to define the ordering between strings in the following way:

$$w \leq x \iff (w = ay) \wedge (x = bz) \wedge (a \leq b \vee ((a = b) \wedge (y \leq z)))$$

Character	Binary code
A	0100 0$\underline{00}$1
C	0101 0$\underline{01}$1
T	0100 0$\underline{10}$0
G	0100 0$\underline{11}$1
a	0110 0$\underline{00}$1
c	0110 0$\underline{01}$1
t	0111 0$\underline{10}$0
g	0110 0$\underline{11}$1

Table 2.1: Encoding of `A`, `C`, `T`, and `G` using ASCII codes.

and

$$\forall w : \epsilon \leq w \quad .$$

We will use an alphabet $\Sigma = \{\$, A, C, T, G\}$, where $\$ \leq A \leq C \leq T \leq G$. The character \$ is used to denote ends of strings when needed. The reader might also notice we swapped T and G compared to their usual order. This is due to their transcoding from the ASCII character set to bring down their space requirements from 8 to 2 bits. The encoding uses only their second and third bits (see Table 2.1). We can change the order because, semantically, characters have no order on their own and we may impose any order on them.

2.2 Regular de Bruijn graph

A de Bruijn graph [8] over alphabet Σ with $k \in \mathbb{N}$ is defined as $G(V, E)$ where every k-gram in Σ^k is represented by a vertex v. This means that $\text{label}(v) \in \Sigma^k$ and for $u, v \in V$ where $u \neq v$ it holds that $\text{label}(u) \neq \text{label}(v)$. Consequently, $|V| = \sigma^k$.

Edges of G are defined as:

$$E = \{(u, v) \in V \times V \mid \text{suffix}_{\text{k-1}}(u) = \text{prefix}_{\text{k-1}}(v)\} \quad .$$

In other words, if the $k-1$-suffix of vertex u matches the $k-1$-prefix of vertex v, they are connected. Consequently, for each vertex v, outdegree$(v) =$ indegree$(v) = \sigma$ and the number of edges in the whole graph is σ^{k+1}. We label the edges with the last character of the label of their destination vertex, that is, if label$(v) = wa$, then label$((u, v)) = a$. The edges tell us which character we must append to the source node's label to get the label of the destination node. The concatenation of the labels of the first vertex and all the edges of an Eulerian tour of the graph gives us a string that contains every $k+1$-gram.

A REGULAR GRAPH is a graph where

$$\forall v \in V\ :\ \text{indegree}(v) = \text{outdegree}(u)$$

and

$$\forall u, v \in V\ :\ \text{indegree}(u) = \text{indegree}(v) \land \text{outdegree}(u) = \text{outdegree}(v)\ .$$

Every vertex in a de Bruijn graph as defined above has both indegree and outdegree σ, therefore de Bruijn graphs are regular.

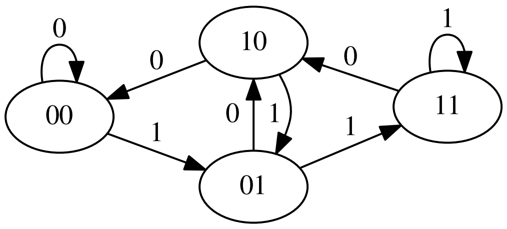


Figure 2.1: Regular de Bruijn graph with alphabet $\Sigma = \{0, 1\}$ and $k = 2$.

Figure 2.1 shows an example of a de Bruijn graph with $k = 2$ and the alphabet $\Sigma = \{0, 1\}$. Hence, $V = \{00, 01, 10, 11\}$. The nodes are connected so that if we remove the first character of the node and append the label of an outgoing edge we get the label of the node the edge is directed to. If we follow an Eulerian tour of the graph from the node `00`, we can assemble a string `$0001011100`, which contains every binary string of length $k+1=3$. Usually, there is more than one tour.

2.3 De Bruijn graph of a text

If we have a text T with alphabet Σ, we can represent it with a de Bruijn graph. For any k we can construct a set of all k-grams present in T. We then create a vertex for each k-gram and connect them following the same rule as in regular de Bruijn graphs. What we get, is a graph that shares many of the properties of a regular de Bruijn graph, but with some of the vertices and edges missing. Such a graph is not necessarily regular.

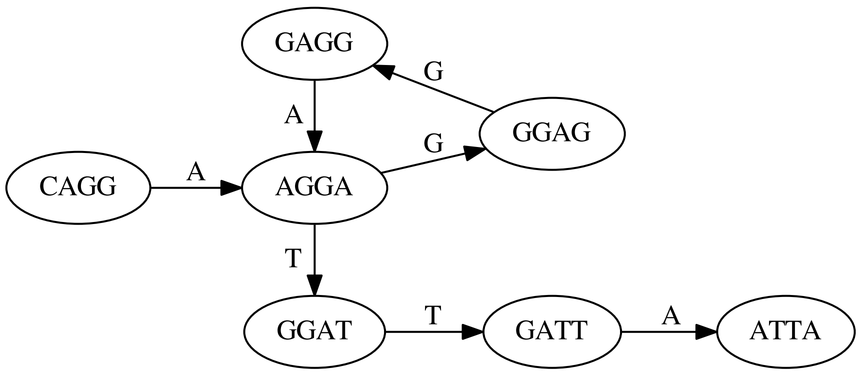


Figure 2.2: De Bruijn graph for a genome sequence `CAGGAGGATTA` for $k=4$.

For the alphabet $\Sigma = \{A, C, T, G\}$ Figure 2.2 contains the de Bruijn graph for text $T =$ `CAGGAGGATTA`. From T we form all 4-grams

$$[\texttt{CAGG}, \texttt{AGGA}, \texttt{GGAG}, \texttt{GAGG}, \texttt{AGGA}, \texttt{GGAT}, \texttt{GATT}, \texttt{ATTA}] \ ,$$

of which the unique ones are $\{\texttt{CAGG}, \texttt{AGGA}, \texttt{GGAG}, \texttt{GAGG}, \texttt{GGAT}, \texttt{GATT}, \texttt{ATTA}\}$. This set contains all our nodes which is less than 3% of the $4^4 = 256$ possible nodes of a regular de Bruijn graph with $k=4$. Also a number of edges from a regular de Bruijn graph are missing. For example, `AGGA` and `GGAT` are connected with an edge labeled `T` because they share `GGA`, but an edge labeled `C` out of `GGAT` does not exist because there is no node `GATC`. Using a crude algorithm, which looks for trails, we can extract pieces `CAGGA`, `GGAGGA` and `AGGATTA` from this graph. However, this graph has an Eulerian tour, so we can follow it and reassemble the full "genome" `CAGGAGGATTA`.

If k is chosen so that $k+1$-mers are not unique, we can no longer reconstruct the genome from the graph. Figure 2.3 shows the graph for the

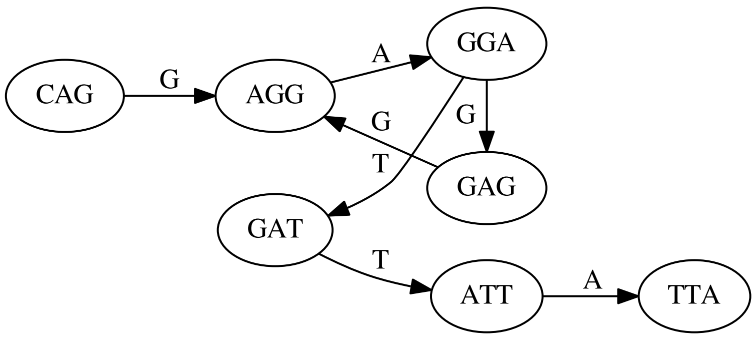


Figure 2.3: De Bruijn graph for a genome sequence `CAGGAGGATTA` for $k = 3$.

same genome as before, but with $k = 3$, which is too small, as `AGGA` is not unique. The new graph is no longer Eulerian and we can not reconstruct the genome from it. However, we can still extract some useful sequences from it, by following walks between branches. We can reconstruct: `CAGGA`, `GGAGG` and `GGATTA`. All of these appear in to source text.

2.4 Data structures

2.4.1 Data structure types

Navigational data structure A NAVIGATIONAL DATA STRUCTURE is a structure that allows us to navigate the de Bruijn graph. That is, given a vertex v, it supports the following operations:

next(v, a): returns vertex u such that label$((v, u)) = a$; if it does not exist it returns $\perp$;

previous(v, a): returns the vertex u such that label$((u, v)) = a$; if it does not exist it returns $\perp$;

neighbors(v): returns

$$S = \{u \in V \mid \exists a : \text{next}(v, a) = u \vee \text{previous}(v, a) = u\} \quad .$$

To give a few examples from Figure 2.2:

$$\begin{aligned} \text{next}(\texttt{AGGA}, \texttt{G}) &= \texttt{GGAG} \\ \text{previous}(\texttt{AGGA}, \texttt{A}) &= \perp \\ \text{previous}(\texttt{AGGA}, \texttt{C}) &= \texttt{CAGG} \\ \text{neighbors}(\texttt{AGGA}) &= \{\texttt{CAGG}, \texttt{GAGG}, \texttt{GGAG}, \texttt{GGAT}\} \end{aligned}$$

Membership data structure MEMBERSHIP DATA STRUCTURE is in general a dictionary [12] that maps the value of a key to either true or false. In our case we need only a static dictionary that supports only the query operation member(w) and no update operations insert(w) and delete(w). Consequently, its implementation can be designed with a smaller memory footprint and lower time complexity (cf. [7, 18]). To wrap up, our data structure needs to support only a single operation member(w), which returns v, iff there is vertex v such that label$(v) = w$ and $\perp$ otherwise.

Theorem 2.1 *A membership data structure is also a navigational data structure.*

To prove the theorem, we have to show how to implement operations from the navigational data structure using a membership data structure.

Proof. We use the membership data structure to store labels of vertices of a graph. We also know that the edges of a de Bruijn graph are labeled by characters from Σ. Finally, there is a well defined relationship between labels of u and v, if $(u, v) \in E$. Consequently, $\mathrm{next}(v, a)$ can be defined as

$$\mathrm{next}(v, a) = \mathrm{member}(\mathrm{suffix}_{k-1}(\mathrm{label}(v)) \cdot a) \ ,$$

and

$$\mathrm{previous}(v, a) = \mathrm{member}(a \cdot \mathrm{prefix}_{k-1}(\mathrm{label}(v))) \ .$$

If the query returns a vertex the output is correct; if the query returns $\perp$, it indicates that the string is not in the dictionary and consequently works as defined.

Because the alphabet is finite, we can construct the neighboring vertices in $\Theta(\sigma)$ time by trying all σ possible neighbors

$$\mathrm{neighbors}(v) = \bigcup_{a \in \Sigma} \mathrm{next}(v, a) \cup \bigcup_{a \in \Sigma} \mathrm{previous}(v, a).$$

□

In the case of DNA where $\sigma = 4$, the $\mathrm{neighbors}(v)$ requires eight membership queries.

Chikhi et al. showed that the lower bound for a navigational data structure of a genome is roughly 2.25 bit to 3.25 bit per k-mer [11]. This is less than 1 bit per incoming edge.

2.4.2 Naïve data structure

The most basic data structure is explicit and uses ordinary objects (or C structures) to represent vertices, as shown in Listing 2.1. It uses $\lceil \lg \sigma \rceil k$ bits to represent the k-mer and $2\sigma \cdot 64$ bits[1] for references to the next and previous vertices. For $k = 51$ and $\sigma = 4$ this gives us $51 \cdot 2 + 2 \cdot 4 \cdot 64 = 614\,\mathrm{bit} = 77\,\mathrm{B}$ per vertex.

[1]Assuming a 64-bit processor architecture.

Listing 2.1: An example of a node implemented in C.

```
typedef struct Node {
        char kmer[(KMER_SIZE * SIGMA)/BYTE_SIZE + 1];

        struct Node* forward[SIGMA];
        struct Node* backward[SIGMA];
} Node;
```

For a human genome this means approximately $77\,\mathrm{B} \cdot 4 \cdot 10^9 = 308\,\mathrm{GB}$. If we also want to store satellite data, we could easily run out of main memory on an average computer.

Chain optimization

First, we observe that, since we are using an explicit data structure, the majority of the space is taken up by the references, and, second, most of the k-mer information is available in neighboring vertices. Moreover, we can also assume that a genome will for the most part resemble a single long chain. Using these assumptions, we can compress the nodes that form a chain into a single node, with the outgoing connections copied from the last node and the incoming connections copied from the first node in the chain. The label of the node now has variable length. This technique uses space much more efficiently and is used in the Velvet family of assemblers [38].

SOAPdenovo2 [25] takes chain optimization a step further and uses a data structure described in [37]. The data structure is not a true de Bruijn graph, since the labels of its edges are not single characters, but chains of characters. This brings savings in terms of memory.

In the rest of this chapter we will describe implicit data structures, which use the second observation to avoid the issue of space usage by references. The navigation through the graph can be used to implicitly store vertex labels, thus avoiding explicitly storing k-grams.

2.4.3 Bloom filter

Bloom filter [4] is a data structure that allows us to fuzzily test set membership. It consists of a bit vector of size m and d different hash functions. Each element is hashed using the d hash functions and the bits at those d locations in the vector are set to 1.

When we want to test set membership, we hash the query with the d hash functions and check the bits at those d addresses. If all of them are set, then our query *may be* a member of the set. On the other hand, if a single of the hashed bits is not set, the query is definitely not in the set. It is possible, due to a hash collision, that all d bit are set, yet the query still is not in the set and we get a *false positive*.

Since we assume that hash functions hash to each location with equal probability, the probability for a certain bit not to be set when an element is inserted is

$$\left(1 - \frac{1}{m}\right)^{d} ,$$

and after inserting n elements the probability becomes

$$\left(1 - \frac{1}{m}\right)^{dn} .$$

To calculate the probability of a false positive, we need the probability that any d random bits are set. For one bit the probability is

$$1 - \left(1 - \frac{1}{m}\right)^{dn} ,$$

and for d bits the probability of a false positive becomes

$$\left(1 - \left(1 - \frac{1}{m}\right)^{dn}\right)^{d} .$$

For a filter with $m = 10000$, $d = 3$ and $n = 2000$ elements, the probability of a false positive becomes 9.2%.

If we still want an exact membership test we can now use a more expensive data structures like real hash tables. Bloom filters become very useful when membership tests are expensive and we expect most of them to fail.

The bioinformatics framework GATB [13] uses Bloom filters with a slight twist based on the earlier work in [10]. First it uses multiple levels of bloom filters instead of just one (cf. perfect hashing [18]). Then, instead of testing the elements that pass the filter for membership on the complete set, it builds a set of false positives, called critical false positives or cFP and tests against those. So, if the query passes the Bloom filter and *is not* in cFP, then it is a member of the set.

2.4.4 BWT-derived data structures

Burrows-Wheeler transform [9, 16] (or BWT for short) is an algorithm that losslessly transforms a text to improve run-length encoding. A side effect of the transformation is also the ability to search the text for substrings [16], which is used in the FM-index [16] data structure. The algorithm used for the queries is described in a later section (Algorithm 4.1).

The FM-index of a genome is essentially a membership data structure. Any k-mer can be checked for membership in $O(k)$ time and $O(n)$ space. Chikhi et al. used this in DBGFM, their de Bruijn graph [11] data structure implementation.

Bowe et al. presented a data structure based on XBW-transform [17]. It shares many similarities with BWT, but uses additional data to allow for fast queries to determine a node's neighborhood [5].

2.4.5 Distributed data structure

Because genome assembly is very computationally demanding, it is reasonable to employ clusters of computers, which requires the use of distributed data structures. ABySS is an assembler that distributes k-mer information across multiple nodes of a cluster according to their hashes (cf. peer-to-peer structures like [28]). This means it is possible to deterministically calculate which node has information about a given k-mer. The data structure stores neighborhood information in 8 bits by recording which edges exist in both directions, making it a membership data structure [35].

2.5 Eulerian tour

As we have mentioned in the Introduction, the genome can be extracted from the de Bruijn graph using an Eulerian tour. In this section we present algorithms used for constructing Eulerian tours of graphs. Later we will use them as intuition behind actual practical algorithms. The presented algorithms are general and work on all Eulerian graphs, including de Bruijn graphs.

Before we continue we must define the following terms:

walk in a graph $G(V, E)$ is an alternating sequence of vertices and edges $(v_1, e_1, v_2, e_2, v_3, \ldots, v_{n-1}, e_{n-1}, v_n)$ where $v_i \in V$, $i = 1, 2, \ldots, n$ and $e_i = (v_i, v_{i+1}) \in E$, $i = 1, 2, \ldots, n-1$;

closed walk $(v_1, e_1, v_2, \ldots, v_n, e_n, v_1)$ is a walk that starts and ends at the same vertex;

trail is a walk where $e_i \neq e_j$, iff $i \neq j$ (i.e. each edge is traversed at most once);

tour is a trail that is also a closed walk;

subgraph is a graph $G'(V', E')$ inside graph $G(V, E)$ where $V' \subseteq V$ and $E' \subseteq E \cap V' \times V'$;

strongly connected component is a subgraph G' of G where for every pair of vertices $u, v \in V'$ there exists a walk $(v, \ldots, u)$.

An Eulerian tour in a directed graph is a walk that goes through every edge of the graph exactly once and finishes at the same vertex it started from. Some graphs contain an Eulerian trail, but not a tour. In those cases we can add a virtual edge to link the last vertex in the trail to the first, thus creating a tour. We will henceforth concentrate on tours. We call a graph containing an Eulerian tour an EULERIAN GRAPH.

The requirements for a graph to contain an Eulerian tour or trail are:

1. graph $G(V, E)$ contains a strongly connected component $G'(V', E')$ and for every vertex v either $v \in V'$ or $\neg\exists u \in V : (u, v) \in E \vee (v, u) \in E$;

2. $\forall v : \text{incoming}(v) = \text{outgoing}(v)$ or, if the graph contains only an Eulerian trail, there exist nodes u and v, such that

$$\text{incoming}(u) - \text{outgoing}(u) = -1$$
$$\text{incoming}(v) - \text{outgoing}(v) = 1.$$

Vertices u and v are the starting and the finishing vertices of the trail.

In the rest of this chapter we use $n = |V|$ and $m = |E|$.

2.5.1 Sequential algorithm

Finding Eulerian tours of a graph is fairly simple. The procedure follows these steps:

1. Test whether the graph is Eulerian.

2. Start from vertex v and follow unvisited edges until we get back to v.

3. If there is a vertex u in the tour with an unvisited edge, start from u and follow unvisited edges until we get back to u.

4. Join the new trail to the main trail by directing it from u along the new trail then proceeding along the main trail.

5. If there are still unvisited edges, go to 3.

The first step is simple and takes $O(n)$ time. It consists of verifying that each vertex has the same number of incoming and outgoing edges.

In the second step we construct the first basic tour. There is no wrong direction to go, since the tour starts and ends in v *and* includes every edge. It is possible that the walk terminates before visiting all edges, but we will fix that in the next step.

Third step looks for remaining trails. If we can leave u through one edge, there must be at least one free edge returning to u which we must still visit, thus creating a tour.

Lastly we join the newfound tour to the main trail we have constructed so far. If we have $t_1 = (v \dots u \dots v)$ and $t_2 = (u \dots u)$ we can merge them into $(v \dots t_2 \dots v)$.

We repeat this until all edges have been added to the tour. Each edge is visited exactly once giving us the time and space complexity $O(m)$ [23]. Consequently, the complexity of the algorithm is $O(m + n) = O(m)$.

2.5.2 PRAM algorithm

The PRAM algorithm described by Atallah and Vishkin [2] uses the same idea as the sequential algorithm. It finds tours in the graph and merges them together into one tour.

The following is an outline of the algorithm from [2]:

1. Partition the edges of the graph $G = (V, E)$ into pairwise disjoint tours.

2. Construct an auxiliary undirected graph $G_1 = (V_1, E_1)$ with two types of vertices: "real-vertices" taken from V and "tour vertices" with one vertex per tour found in step 1. E_1 consists of edges going from each vertex to each tour it belongs to.

3. Find a spanning tree $T = (V_1, E_1')$ of G_1. Replace each edge in T with two antiparallel edges to form a directed Eulerian graph T'.

4. Find the Eulerian tour of T' and use it to connect the tours found in step 1 into one Eulerian tour of G.

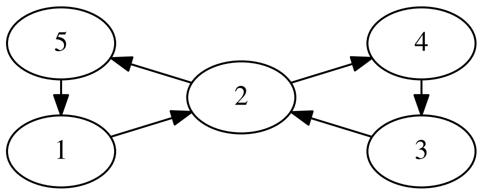


Figure 2.4: Simple directed graph.

We will use the graph in Figure 2.4 as an example, another example is available in Appendix A. The first step is achieved by pairing each vertex's edges together. Throughout the algorithm we will hold the succession of edges in an array called $\texttt{succ}[m]$ where $m = |E|$, so at the end when we extract the order of edges in the tour, e will be followed by $\texttt{succ}[e]$. We enumerate all incoming and outgoing edges of each vertex, so that $\texttt{incoming}(v, k)$ is the k^{th} incoming and $\texttt{outgoing}(v, k)$ is the k^{th} outgoing edge of v. Now we can pair them up:

$$\texttt{succ}[\texttt{incoming}(v, k)] = \texttt{outoging}(v, k).$$

This pairing operation can be done in $O\left(\frac{m}{p}\right)$ time, where p is the number of processors. Table 2.2 shows the state of `succ` after the first step.

Edge	$succ[e]$
$(1, 2)$	$(2, 5)$
$(2, 4)$	$(4, 3)$
$(2, 5)$	$(5, 1)$
$(3, 2)$	$(2, 4)$
$(4, 3)$	$(3, 2)$
$(5, 1)$	$(1, 2)$

Table 2.2: The state of $\texttt{succ}[e]$ after the first assignment.

Edge	$\mathtt{D}[e]$	Tour name
$(1,2)$	$(1,2)$	A
$(2,4)$	$(2,3)$	B
$(2,5)$	$(1,2)$	A
$(3,2)$	$(2,4)$	B
$(4,3)$	$(2,4)$	B
$(5,1)$	$(1,2)$	A

Table 2.3: After tours are found and named.

Next we need to identify the tours. We can achieve that by the pointer jumping technique [22]. We will use $\mathtt{D}[m]$ initialized to $\mathtt{D}[e] = e$ to store an edge that is a unique identifier for the tour which edge e belongs to. We also use a temporary array $\mathtt{next}[m]$, which starts as a copy of `succ`. For each edge we do:

$$\begin{aligned} \mathtt{D}[e] &= \mathtt{min}(\mathtt{D}[e], \mathtt{D}[\mathtt{next}[e]]) \\ \mathtt{next}[e] &= \mathtt{next}[\mathtt{next}[e]]. \end{aligned}$$

The function `min` returns the smaller edge according to some ordering. We repeat this $\lceil \log m \rceil$ times. This takes $O\left(\frac{m}{p} \log m\right)$ time overall. Now we have an array `D` of representatives for each edge's tour. For ease of understanding we named the tours of the example in Table 2.3.

Array `D` already yields the edges of G_1 and the vertices can easily be computed.

We define $\mathtt{certificate}[v, w]$ where v is a "real" vertex and w is a "tour" vertex in G_1. The map `certificate` contains an edge (i, v) from G for every v in G that lies on w and certifies that v does indeed belong to tour w. Assigning certificates takes $O(\log n)$ time on $n + m$ processors. Table 2.4 shows the chosen certificates for the example.

Now we compute the spanning tree T of G_1, which takes $O(\log n)$ time on $n + m$ processors. G_1 and its T for the example are in Figure 2.5. Next

Vertex	Tour	`certificate`$[e]$
1	A	$(5, 1)$
2	A	$(1, 2)$
2	B	$(3, 2)$
3	B	$(4, 3)$
4	B	$(2, 4)$
5	A	$(2, 5)$

Table 2.4: Edges that certify that a vertex belongs to a tour.

we produce a directed graph T' where we replace each edge in T with two antiparallel edges. T' is Eulerian, since each edge was replaced with an incoming and an outgoing edge therefore balancing the number of edges for each node. We show T' and G of the example combined in Figure 2.6.

The last part uses T' to merge tours from the first step into one tour. First we must find an Eulerian tour of T'. "Real" vertices v in T are connected to "tour" vertices through edges $\{v, u_1\}, \ldots, \{v, u_d\}$ where d is the degree of v. After we introduce directed edges, we can assign successors for "real" vertices in a circular order with $\texttt{succ}[(u_i, v)] = \big(v, u_{(i+1) \bmod d}\big)$.

For "tour" edges, we must first use $(i, v) = \texttt{certificate}[v, w]$ to define (v, j) to be $\texttt{succ}[(i, v)]$. Edge (v, j) is the edge following the certifying edge for v in w. For all certifying edges (i, v) we assign successors as follows:

$$\begin{aligned} \texttt{succ}[(i, v)] &= (w, v) \\ \texttt{succ}[(v, w)] &= (v, j)\,. \end{aligned}$$

If we look at `succ` we notice it does not contain a valid walk. However, if we follow it blindly, we notice that the edges of V spell out an Eulerian tour of G and the edges of T' produce an Eulerian tour of T'. The new `succ` for

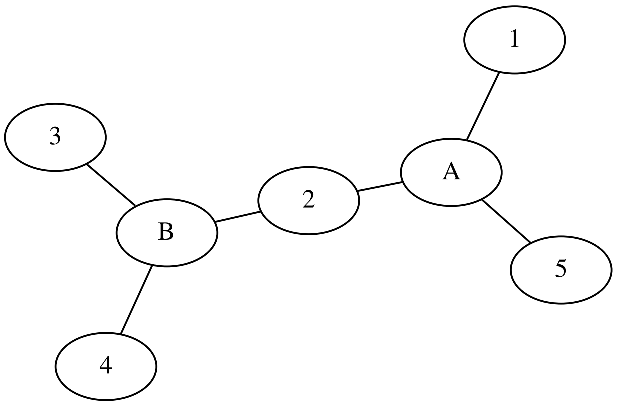


Figure 2.5: Graph G_1 for the example. The spanning tree is obvious.

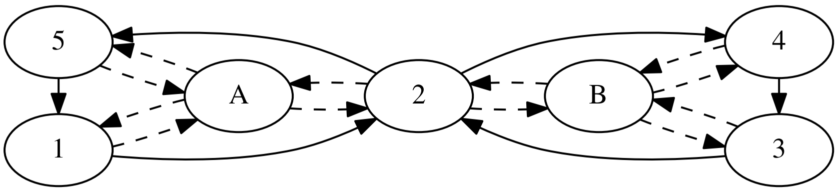


Figure 2.6: T' and G on the same graph. The edges of T' are dashed.

our example is in Table 2.5. It spells the following sequence:

$$((\mathbf{1},\mathbf{2}),(A,2),(2,B),(\mathbf{2},\mathbf{4}),(B,4),(4,B),(\mathbf{4},\mathbf{3}),(B,4),(4,B),(\mathbf{4},\mathbf{3}),(B,3),$$
$$(3,B),(\mathbf{3},\mathbf{2}),(B,2),(2,A),(\mathbf{2},\mathbf{5}),(A,5),(5,A),(\mathbf{5},\mathbf{1}),(A,1),(1,A),(\mathbf{1},\mathbf{2})).$$

The edges of G in bold are an Eulerian tour of G.

Now we must clean it up. There are no more than two successive edges of T' in the sequence defined by `succ`, so we perform the following operation twice:

```
if succ[$e$] $\in T'$ then
```

Edge	succ$[e]$	Edge	succ$[e]$
$(1,2)$	$(A,2)$	$(A,1)$	$(1,A)$
$(2,4)$	$(B,4)$	$(A,2)$	$(2,B)$
$(2,5)$	$(A,5)$	$(A,5)$	$(5,A)$
$(3,2)$	$(B,2)$	$(B,2)$	$(2,A)$
$(4,3)$	$(B,3)$	$(B,3)$	$(3,A)$
$(5,1)$	$(A,1)$	$(B,4)$	$(4,B)$
$(1,A)$	$(1,2)$		
$(2,A)$	$(2,5)$		
$(2,B)$	$(2,4)$		
$(3,B)$	$(3,2)$		
$(4,B)$	$(4,3)$		
$(5,B)$	$(5,1)$		

Table 2.5: The state of succ$[e]$ after computing the succession for T'.

```
    succ[$e$] ← succ[succ[$e$]]
end if
```

Now we can begin at any edge of E and follow succ to produce an Eulerian tour of G. The final state of succ for our example is in Table 2.6.

Edge	succ$[e]$
$(1,2)$	$(2,4)$
$(2,4)$	$(4,3)$
$(2,5)$	$(5,1)$
$(3,2)$	$(2,5)$
$(4,3)$	$(3,2)$
$(5,1)$	$(1,2)$

Table 2.6: The state of succ$[e]$ after cleanup.

The time complexity of the whole algorithm is $O(\log n)$ using $n+m$ processors and $O(n+m)$ space [2].

Chapter 3

Genome sequencing and assembly

A DNA molecule is a polymer, a chain of molecules that are similar to each other called monomers. DNA is composed of four different types of molecules called NUCLEOTIDES (nt): adenine, cytosine, guanine and thymine[1] [1]. Usually, when sequences of these molecules are described in text, the first letters of their names are used (usually capitalized): `A`, `C`, `G` and `T`.

Every chain of nucleotides comes with its pair and together they form the well known double helix. The pair is a negative of the chain; each nucleotide is linked to its opposite: adenine to thymine and cytosine to guanine. Each strand of the pair is oriented in the opposite direction of the other, thus forming REVERSE COMPLEMENTS of one another [1]. For instance, the sequence `ATGAGCATTCCGTTT` has `AAACGGAATGCTCAT` on the other side. When we are dealing with nucleotides in pairs, we refer to them as BASE PAIRS (bp).

Before sequencing, the genome is broken into manageable pieces. When each piece of DNA is sequenced, the data produced is called a READ. The read length is limited by both the physical DNA segment length and the sequencer's maximum configured reading length (see Figure 1.1).

Most current technologies produce so called SHORT READS, which are

[1]For the sake of brevity and without loss of generality we will ignore RNA.

around 100 bp long. Reads that are several hundreds of base pairs long are called LONG READS.

PAIRED-END READS are a category of reads, which are special in that they represent both ends of a single segment of DNA, while discarding the rest of the segment. In a way they are "cheating" the length limitation by ignoring the middle part.

3.1 Sequencing technologies

3.1.1 Sanger sequencing

Before the advent of Next-generaton sequencing, SANGER SEQUENCING [34] was the most popular method of sequencing genomes. It was developed in the 1970's and remains popular in certain applications until today.

Reads produced by the Sanger method are usually much longer than those produced by Next-generation methods, though the gap is getting narrower with newer refinements [31].

3.1.2 Sequencing by hybridization

SEQUENCING BY HYBRIDIZATION is an alternative technique that involves preparing an array of 4^k different sites where DNA can bind to its complimentary sequence. Then, the genetic material is allowed to bind to those sites. Finally, the array is analyzed, to see which sites were bound and which remained empty. The different k-mers that are present in the sequence are thus determined [14].

The process of assembly usually assumes all k-mers are unique. We can expect a given k-mer to appear every $\frac{4^k}{2}$ bases, therefore we need a k large enough to guarantee uniqueness. This is hard to achieve for large genomes [14], since 4^k grows very quickly.

We can reduce the number of required binding sites, if we know which k-mers to expect [14], so this technology still finds its uses in niche applications.

3.1.3 Next-generation sequencing

The Next-generation sequencing (NGS) is characterized by very short read lengths, around 100 bp, though newer techniques can go beyond that. It compensates for it, however, with high throughput. This creates large amounts of data and requires the use of advanced algorithms to filter and analyze them. The reads produced by this family of technologies are commonly referred to as short reads. There are many next-generation sequencing technologies being developed – we will mention only a few.

Illumina (formerly Solexa) sequencing is currently the most common technology used to produce short reads. The reads it produces are usually around 100 bp to 150 bp long. It also allows sequencing both ends of a segment of DNA, thus producing paired-end reads.

454 Life Sciences' technology uses pyrosequencing [32] and produces comparatively long reads, up to 1000 bp long [31].

We will also briefly mention Applied Biosystems' SOLiD method, which sequences and encodes two bases at a time. Every base is sequenced twice, once with each of its neighbors.

In the output it uses four "colors", shown in Table 3.1, to encode those transitions in addition to one single base per read, from which the other bases can be decoded. This improves error detection, since a single base that differs from the reference sequence induces change in two characters making the rest of the read continue as expected, whereas an error in a single character completely changes the meaning of the rest of the read. 2-base reads therefore work well when used with traditional reads for reference, but are not very good for assembling new genomes on their own.

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

Table 3.1: Colors used in 2-base encoding. First base is on the left, second base is on top.

3.2 Genome data

3.2.1 Properties

We will now look at reads from a data-centric point of view. First, each dataset can come from a different sequencer and may be processed in different ways before it makes its way to us. While formats are standardized, some of the properties of data may vary. If we obtain reads from a third party, like a project in the NCBI database, we must look at the files we receive carefully, including reading preparation methods, to interpret them properly.

The reads, while nominally the same length, might actually represent segments of the DNA strand that are shorter so the additional bases should be removed. It is possible that the reads still contain so called adapter sequences, artificially created known sequences, which were attached as part of the preparation process and are not actually part of the genome. The range of quality also has to be checked, since different sequencing technologies encode quality data differently.

We may receive quality information along with read data. This is usually in the form of a PHRED QUALITY SCORE:

$$Q = -10 \log_{10} P \ ,$$

where P is the probability that a certain nucleotide was misread.

Because of all these variable factors, genome analysis is often done using k-MERS. A k-mer is a fixed-length string, which helps normalize the data.

The set of unique k-mers from all the reads and the frequencies of those k-mers' appearances are commonly used to simplify the analysis.

3.2.2 File formats

We will talk about two most common genome data storage formats. They are both text-based, which comes in handy when we want to quickly examine their contents. It also makes them very portable; any computer with a text editor can be used to read them and simple analyses can be performed using generic command line programs.

The drawback, however, is their large size, since each nucleotide is encoded with 8 bits instead of 2 bits that would be required for four different characters, therefore they are usually compressed, most often using `gzip`.

The International Union of Pure and Applied Chemistry, IUPAC, devised a standard notation for displaying DNA in text, shown in Table 3.2. Along with codes for adenine, cytosine, guanine and thymine it includes codes for places in the genome sequence where any one of multiple bases may be present. These (aside from `N`) are not used in reads, since reads contain data from one physical DNA strand, but appear in reference sequences, which have been built as a consensus between multiple genomes.

FASTA is the simplest format for storing genome data. It includes a single line prefixed by a > character, followed by an optional descriptor. In the example in Listing 3.1 the descriptor is divided as follows: `gi` introduces a sequence identifier (`1002793874`); `gb` means that the sample comes from NCBI's GenBank database and has the identifier and version in the database `LRSR01000001.1`. The line ends with a general sequence description. The genetic data starts on the second line and continues until the next line prefixed by >. Long sequences are usually formatted into lines of constant width. In the example lines 2 through 4 are genome data, which has been split into lines of 70 characters.

Character	A	C	G	T
A	A			
C		C		
G			G	
T				T
W	A			T
S		C	G	
M	A	C		
K			G	T
R	A		G	
Y		C		T

Character	A	C	G	T
B		C	G	T
D	A		G	T
H	A	C		T
V	A	C	G	
N or -	A	C	G	T
Z				

Table 3.2: IUPAC nucleic acid notation. (Uracil omitted, see footnote on page 27)

Listing 3.1: Example of a FASTA entry.

```
>gi|1002793874|gb|LRSR01000001.1| Hypsibius dujardini strain Sciento
    H_dujardini_15454, whole genome shotgun sequence
GACAGACAGACAGACAGACAGACAGACATACATGCAGGTAGGCAGACAGGCAGACAGACGGAACGGTATA
TAACTAAAGATTAAGAACACACTTACTCGGTTTCAGTCGAACATCTACAGCCACAGAAAATGACTGATCG
CTCTGGGATCGACATCGGAACAAGCCCACACTAGCCGCCGAAGCCCTCTCGATTGTGAGCAGACCGCGAC
```

FASTQ format is used to store quality information along with genetic data. The format itself is simple: first line starts with `@`, followed by the sequence identifier and additional information. The example in Listing 3.2 has an identifier for the read set in the NCBI database (`SRR2052522`) and the exact read `8996868`, information about the sequencing equipment (in this case Illumina-specific `HWI-EAS390_0001:4:51:12456:6096`) and the length of the read. The second line contains a genome sequence. The third line starts with + and optionally repeats the identifier. The last, fourth, line contains quality data. This repeats as many times as there are reads stored in the file.

Quality Phred scores for each base pair are encoded as a sequence of characters from a range in the ASCII code tables. For example, in reads produced by Sanger sequencers, quality score 0 is represented by a `!`. The full range is `!` to `I` for scores 0 to 40. Older Illumina technologies used `@` through `h`. Newer Illumina machines use `!` to `J` to represent 0 to 41. We know our example comes from a newer Illumina sequencer, so we can interpret the fourth line as quality scores: 35, 39, 39, 39, 38, 39, 33, 39, 38, ...

Listing 3.2: Example of a FASTQ entry.

```
@SRR2052522.8996868 HWI-EAS390_0001:4:51:12456:6096 length=242
GTTTAGCAACATATGCGGCTTGCCCTGAACACCGGGCACCACTGTATTGATGCTCATTTCCCATAG
+SRR2052522.8996868 HWI-EAS390_0001:4:51:12456:6096 length=242
DHHHGHBHGHDHHHH@HHHHGB<G@;=?=?DG@8DEGGED@28?8CCECCGE3G<8?DB8EBCC<<
```

3.3 Assembly process overview

In general there are two types of genome assembly: *de novo* assembly, which is building a genome from scratch, and mapping or resequencing assembly, which uses a preexisting genome assembly as a guide to assembling the genome. We will focus on *de novo* assembly.

The process of *de novo* genome assembly is outlined as follows:

Sequencing First the genome is sequenced using one of the methods mentioned mentioned earlier. This produces the initial dataset usually in the form of large compressed FASTQ files (Figures 1.1a, 1.1b, 1.1c and 1.1d on page 3).

Data filtering and cleanup Records in these files can be cleaned up using simple techniques. These include removing low quality reads, removing low quality parts of reads, splitting paired-end reads to suite the particular input requirements of the assembler, etc.

The rest of the process usually takes place inside a single piece of software.

Graph construction A graph is used to represent the relations between different parts of the sequenced data. In the past the type of graph used may have been an overlap-consensus graph [26]; modern assemblers use a de Bruijn graph. This helps with aligning the reads (Figure 1.1e).

Graph transformation Various transformations can be performed on the graph. These usually perform more sophisticated forms of error correction.

For instance, a BUBBLE, part of the graph where a trail diverges and later converges again, may indicate a small error, or an SNP (single nucleotide polymorphism), a single differing base. The divergent trails can be merged into the one that appears more often in the input data, since the less frequent one is probably an error. In the case of SNPs, we want to choose one of the options and possibly analyze the alternative later, by mapping the reads back to the assembled sequence.

A dead end in the graph, also known as a TIP, may indicate an erroneous read, that completely diverges from the actual genome. The nodes of that part of the graph can be completely removed.

We can also prune the graph based on how many reads support each vertex. Vertices whose labels appear in few reads are probably results of errors.

Contiging A CONTIG is a contiguous area of genome we have managed to assemble. Contigs are extracted from the transformed graph using various algorithms, usually by finding long trails in the graph. Ideally, an Eulerian tour of the graph would be used, to extract the longest tour possible, but the graphs are rarely Eulerian.

Scaffolding Finally, we would like to know how contigs relate to each other. We want to know their relative positions and distances between them. At this point we need long reads and paired end reads. If we manage to anchor their ends into separate contigs they give us information about the position and distance between those contigs. The result is our genome (Figure 1.1f).

3.3.1 Result assessment

Since we are going into the process blindly it is hard to assess the quality of the result. We can not actually tell if the assembled genome is the same one we were trying to sequence.

The most common method of assessing quality is N50, the length of the shortest contig such that the sum of its length and the lengths of longer contigs reaches 50% of the sum of length of the genome, or, if that is not available, the sum of the lengths of all contigs [15]. The definition can be expanded to other percentages such as N90 or N10. These metrics show what fraction of the assembly consists of few large contiguous areas, as opposed to many small fragments.

Another metric is E-size [33], which attempts to answer the question: given a random position in the genome, what is the expected size of the contigs covering it? It is computed as

$$E = \sum_{C} \frac{L_C^2}{G} \ ,$$

where L_C is the length of contig C, and G is the estimated length of the genome.

If, on the other hand, we are just testing assemblers on known genomes, we can use many other metrics described in [15].

3.4 Contiging

Every regular de Bruijn graph is Eulerian, since each node has as many incoming and outgoing edges as there are characters in the alphabet. If we follow the Eulerian tour of such a graph, we can compose a string by concatenating the edge lables. That string is the shortest string that contains every $k+1$-gram of that alphabet, where k is the size of the nodes in the de Bruijn graph. Note, there are possibly several such strings.

A de Bruijn graph composed of the k-mers of a given string is also Eulerian, as long as k is large enough that each $k+1$-gram would appear in the string only once, since the string already gives us one of the Eulerian tours of the graph. Due to the properties of the Eulerian tour certain parts of the string can be reordered and still produce a valid tour [21]. However, one of the orders is certain to be the input string. This is usually the case when there are tours inside the Eulerian tour, which can be traversed in any order. The correct order can be determined by mapping reads to the graph. A read can be represented by a trail in the graph. If we look at which direction a read goes at a crossroad in the graph, we can determine the proper order[2] of traversal for that part of the graph. Figure 3.1 shows the de Bruijn graph

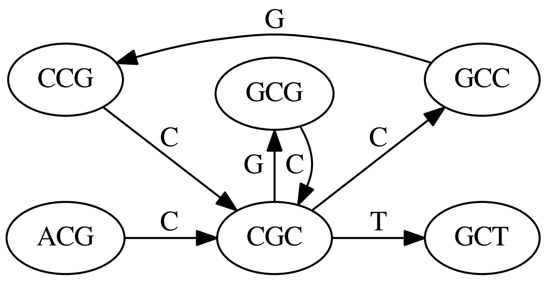


Figure 3.1: De Bruijn graph for a genome sequence `ACGCCGCGCT` for $k = 3$.

for a sequence `ACGCCGCGCT` and $k = 3$. The Eulerian trail contains two subtours, which we could traverse in either order and produce both `ACGC CGC GC T` and `ACGC GC CGC T` (spaces have been added to show which parts of the string can exchange places). However, only one of those is a correct reassembly of the "genome". The reads from that genome will probably contain `ACGCC`, which hints at the correct direction to go at vertex `CGC`.

Due to the errors of the procedure and natural properties of genomes, the graph produced from reads rarely has an Eulerian tour available. The Eulerian tour technique also presupposes that every k-mer appears in the genome

[2]For an Eulerian tour, any order of subtours that share a vertex is admissible. For a genome on the other hand, only one order is correct.

exactly once. In practice, some may not appear in the genome at all, but are present in the graph due to errors, others may appear in the genome multiple times, which can be both due to repeats in the genome itself or because the chosen value of k was too small. Contig building algorithms therefore focus on finding long trails in graphs, that are not necessarily Eulerian.

Before we proceed, we must describe the notion of SAFE STRINGS. A safe string is a string that appears in any reconstruction of the genome from a given graph [36] and is therefore "safe" to return as an output of the contiging algorithm. As we have seen above, the order of traversal of certain parts of the graph can change the output even in a perfect graph. While reads can be used as a guide at those points, they can contain errors and thus do not guarantee correctness.

The simplest safe strings are simple trails in the graph. Series of vertices between two branches in the graph form so called UNITIGS. They are guaranteed to appear in any reconstruction, since any algorithm will traverse those vertices one after another.

Tomescu and Medvedev describe an algorithm that finds safe strings they call OMNITIGS [36]. The general idea is to follow trails that have no chance of turning in on themselves. In terms of an Eulerian tour, this means they are not involved in a tour, which resolves the aforementioned issue with tour order.

In contrast with other techniques, BCALM [11] uses MINIMIZERS [30] to produce longest trails of a set of reads. Minimizers are a way of finding a k-mer that represents a given string, allowing it to be classified with other strings that belong together. In the case of BCALM, the starts and ends of reads are taken into consideration. The reads are grouped according to the lower[3] of either the start or the end of the string. The reads in the lowest group are then joined where beginnings and ends match. The joined strings now belong to either the same or a different group that is higher in the order and lowest group is eventually emptied. These joins are repeated on an ever

[3]According to some total ordering, e.g., lexicographically or by frequency.

higher group until all groups are merged.

BCALM method does not directly use a graph. The algorithm for building DBGFM [11] uses BCALM as part of its own graph construction. However, it does perform the task of producing long strings from the genome, which is the purpose of contiging.

Chapter 4

Space-efficient representation of a de Bruijn graph

In this chapter we introduce kBWT, a new space-efficient membership data structure. It is based on BWT, but takes into account the fact that the queries will never be longer than k and that we do not care about the number of repetitions of the query, only whether it appears in the input text or not.

4.1 Burrows-Wheeler transform

Burrows-Wheeler transform is performed by taking all cyclic rotations of a text T and assembling them into a matrix. The rows correspond to rotations and columns correspond to characters at their respective position in each rotation. The rows are then lexicographically sorted. The last column now represents the transformed text L. An example of the transformation is shown in Figure 4.1.

4.1.1 Backward search

Using Algorithm 4.1 BWT allows for substring search in $O(k)$ time, where k is the length of the substring. It returns false, if the substring is not present or the range of positions in the transformed text where the substring

```
CAGGAGGATTA$
AGGAGGATTA$C
GGAGGATTA$CA
GAGGATTA$CAG
AGGATTA$CAGG
GGATTA$CAGGA
GATTA$CAGGAG
ATTA$CAGGAGG
TTA$CAGGAGGA
TA$CAGGAGGAT
A$CAGGAGGATT
$CAGGAGGATTA
```

(a) All rotations of a text.

```
$CAGGAGGATT A
A$CAGGAGGAT T
ATTA$CAGGAG G
AGGATTA$CAG G
AGGAGGATTA$ C
CAGGAGGATTA $
TA$CAGGAGGA T
TTA$CAGGAGG A
GAGGATTA$CA G
GATTA$CAGGA G
GGATTA$CAGG A
GGAGGATTA$C A
```

(b) Sorted rotations of the text. The last column represents the transformed text L (`ATGGC$TAGGAA`).

Figure 4.1: Example of BWT of a short text `CAGGAGGATTA`.

is present. Search starts by looking at the last character of the query and proceeds from there toward the start of the query, hence the name. Inputs are: the transformed text L and the query P, which is k characters long.

Algorithm 4.1 also uses C, which is a table of the counts of characters smaller[1] than a certain character. Table C for the example in Figure 4.1 in Table 4.1 uses # to store the total count of all characters. It is used in the first line of the algorithm when the query ends with `G`.

The other function and its associated structure used in the algorithm is rank(L, c, i). It returns the number of characters c in the transformed text L up to and including position i. The values for rank for the example are in Table 4.2. This function can be made to run in $O(1)$ time with $o(n)$ space of overhead. For details consult [16].

[1]According to the total order defined on page 8.

Algorithm 4.1 Algorithms for finding substrings in BWT encoded text.

```
function BACKWARD_SEARCH($L, P[1, k]$)
    $i \leftarrow k$, $c \leftarrow P[k]$, $First \leftarrow C[c] + 1$, $Last \leftarrow C[c + 1]$
    while $First \leq Last$ and $i \geq 2$ do
        $c \leftarrow P[i - 1]$
        $First \leftarrow C[c] + \mathrm{rank}(L, c, First - 1) + 1$
        $Last \leftarrow C[c] + \mathrm{rank}(L, c, Last)$
        $i \leftarrow i - 1$
    end while
    if $Last < First$ then
        return false
    else
        return $[First, Last]$
    end if
end function
```

Character	Count
$	0
A	1
C	5
T	6
G	8
#	12

Table 4.1: Table C for the example in Figure 4.1.

Index	$	A	C	T	G
1	0	1	0	0	0
2	0	1	0	1	0
3	0	1	0	1	1
4	0	1	0	1	2
5	0	1	0	1	2
6	0	1	1	1	2
7	1	1	1	1	2
⋮	⋮	⋮	⋮	⋮	⋮

Table 4.2: Values of rank(L, c, i) for the first few indices.

4.2 k-mer Burrows-Wheeler transform

κBWT is a variant of BWT used to query the presence or absence of substrings with length up to k in a text. The data structure is very similar to [5] with simplifications which reduce the overall size of the data structure.

To construct it, we prepend text T with \$ and also append k copies of \$ to the end of T. The obtained new text is T'. Then we construct a set of all unique $k+1$-mers in T'. In practice this can be done using an algorithm like [29].

We first sort $k+1$-mers by their k-suffixes and within each k-suffix by their 1-prefixes. This is similar to ordinary BWT, except that the transformed text will be in the first column instead of the last one without loss of generality. The $k+1$-mers at this point represent edges going into the vertex labeled with the k-suffix of the $k+1$-mer. Sorting can be performed in $O(n \log n)$ time or $O(\log n)$ in parallel [27].

Next, we replace every run of consecutive $k+1$-mers that share a k-suffix, with a $k+1$-mer where the first character of the $k+1$-mer is a composite character representing a set of all first characters of that run (example in Figure 4.2). For instance `0000` means the only first character was `$`, `1000` means there was only `A` or `A` and `$`, `1011` represents `A`, `G` and `T`, etc. Note that each binary string is a single character that is represented here in binary form for clarity. We will be using a composite character that contains a character a and the normal character a interchangeably. These $k+1$-mers represent the k-mers of our graph. Each $k+1$-mer represents a vertex, with the first character representing the incoming edges.

We can apply this merging operation because in our algorithm both *first* and *last* always point to the beginning and the end respectively of such a run of $k+1$-mers, so merging them into a single entry does not affect the execution of the algorithm.

The merging can be executed in parallel by having each processor check, if its $k+1$-mer is the first for that k-mer. If it is not, the processor stops. If it is, it proceeds along building the composite character. It will move at most σ $k+1$-mers, so the time complexity is $O\left(\frac{n+\sigma n}{p}\right) = O\left(\frac{\sigma n}{p}\right)$.

```
ACGTA
TCGTA                        1010CGTA
```

(a) $k+1$-mers before the merge (b) $k+1$-mer after the merge

Figure 4.2: Example of the first merge.

Lastly, we merge the first characters of all runs that share the middle $k-1$ characters into the first $k+1$-mer of the run and replace all other first characters with the empty set 0000 (example in Figure 4.3). If there is a k-mer in the graph connected to one of the k-mers represented by the $k+1$-mers in the run, it is also connected to all other k-mers in that run. Therefore, we must perform this second merging operation. The time complexity of this merge is the same as the time complexity of the previous merge: $O\left(\frac{\sigma n}{p}\right)$.

First characters of our $k+1$-mers, the composite characters we constructed with the two merging steps, now form our transformed string L, similar to ordinary BWT.

Definition 4.1 *Function* RANK(L, c, i) *returns the number of characters with c as part of their set in string L up to and including position i.*

Function `rank` is used in the algorithm to calculate the offset of a k-mer that is connected to a certain k-mer. It must therefore return the same result for all of the $k+1$-mers of a run that shares a the middle $k-1$ characters, which is why we performed the second merging step.

Definition 4.2 *Array $C[\sigma]$ contains in each field c the number of characters represented by the composite characters in L which are lexicographically smaller than c.*

Algorithm 4.2 starts with the last character of the query, then proceeds backward toward the first character, narrowing down the range in which the queried k-mer may appear.

```
1010CGTA
0010CGTC
0001CGTT
0001CGTG
```

(a) $k+1$-mers before the merge

```
1011CGTA
0000CGTC
0000CGTT
0000CGTG
```

(b) $k+1$-mers after the merge

Figure 4.3: Example of the second merge.

Intuitively, Algorithm 4.2 starts in all nodes of the de Bruijn graph that start with the last character of the query. These k-mers are contiguous in the sorted list of k-mers, so we can represent them all as a range between two indices. It then works its way back, in the opposite direction of appropriate edges, until all the trails meet. The place where they meet corresponds to our query.

Lemma 4.1 *During the execution of Algorithm 4.2 variable $First$ always points to the first $k+1$-mer whose k-suffix is prefixed by $P[i,k]$, if such $k+1$-mer exists. Otherwise it points to the position where such a prefix would appear in the sorted order.*

We will show that *First* is an invariant maintained throughout the execution of the algorithm.

Proof. In the base case ($i = k$) this is obviously true.

In the general case, before the update *First* points to the first index, where the k-suffix is of the form $P[i,k]w[1,k-i]$ for some string w. w is also the smallest (lexicographically) such string.

We define $c = P[i-1]$.

Now we are looking for the first index where the k-suffix is of the form $cP[i,k]w[1,k-i-1]$. We know it will be located at or after $C[c]+1$. The offset is equal to the number of $k+1$-mers with prefix that includes c and k-suffix lexicographically smaller than $P[i,k]w[1,k-i]$. The offset is therefore $rank(L,c,First-1)$.

Algorithm 4.2 Modified backward search, that only queries for presence or absence of a k-gram.

```
function K_BACKWARD_SEARCH($L, P[1, k]$)
    $i \leftarrow k$, $c \leftarrow P[k]$, $First \leftarrow C[c] + 1$, $Last \leftarrow C[c+1]$
    while $First \leq Last$ and $i \geq 2$ do
        $c \leftarrow P[i-1]$
        $First \leftarrow C[c] + rank(L, c, First - 1) + 1$
        $Last \leftarrow C[c] + rank(L, c, Last)$
        $i \leftarrow i - 1$
    end while
    if $Last < First$ then
        return false
    else
        return true
    end if
end function
```

We will prove this by contradiction. Let's assume there is such an index j that its $k+1$-mer is of the form $bcP'[i,k]w'[1,k-i-1]$ and such index l, that $j > l$, and its $k+1$-mer is $dcP[i,k]w[1,k-i-1]$. There are also such indices $j' < l'$ with their corresponding $k+1$-mers $cP'[i,k]w'[1,k-i]$ and $cP[i,k]w[1,k-i]$. l' is equal to *First* before the update.

We want to prove that j' and l' do not switch places after the update. First we look at what happens if $P' \neq P$.

If $P' < P$, then $j > l$ cannot be true.

If $P' > P$, then $j' < l'$ cannot be true.

Now let's assume $P' = P$.

If $w'[1,k-i-1] > w[1,k-i-1]$, then $j' < l'$ cannot be true.

If $w'[1,k-i-1] < w[1,k-i-1]$, then $w'[1,k-i] < w[1,k-i]$, which contradicts the definition $First$.

If $P' = P$ and $w'[1,k-i-1] = w[1,k-i-1]$, then $w'[1,k-i] < w[1,k-i]$, which contradicts the definition of *First*.

Therefore $j < l$ and all $k+1$-mers prefixed by c with $k-1$-infixes less than $P[i,k]w[1,k-i-1]$ appear before $cP[i,k]w[1,k-i]$. This is also the number of $k+1$-mers with second character c and suffix less than $P[i,k]w[1,k-i-1]$. This "less than" relationship also makes the operation "overshoot" in the case where the prefix does not appear.

All $k+1$-mers which share the middle $k-1$ characters share their predecessors, which is why we performed the second merge step when building L. This removes the excess characters and makes the *rank* operation report the correct number of $k+1$-mers with appropriate suffixes. □

The proof for *Last* is similar. The offset is the same as for *First* with the addition of all the $k+1$-mers between *First* and *Last*. In the case where the k-mer is not present *Last* points to the index *before* the gap where it would appear.

Theorem 4.1 *Function* k_backward_search(L,P) *returns, true if k-mer P was the suffix of any of the $k+1$-mers used to construct L and false otherwise.*

Proof. *First* is always updated to point to the first $k+1$-mer whose k-suffix is prefixed by $P[i,k]$, therefore after k steps it points to the first $k+1$-mer suffixed by P. Similar is true for *Last*. If the k-mer is absent, *First* points to the position where the k-gram should appear and *Last* points before that position, and the function returns false. □

4.2.1 Auxiliary Data Structures

While all aforementioned operations can be performed by a linear scan over L, that would be extremely inefficient. We use additional data structures, to reduce the query times to $O(k)$.

As we have implied before, we use an array C to hold the number of characters less than any given character. Its size is $(\sigma+1)\cdot\lg n$, or in our case, $5\cdot\lg n$. The $+1$ comes from our need to know the number of all characters for the first step of the algorithm. We can omit `$`, since we never actually query it.

To allow the rank function to complete in constant time, we use the data structure presented by Ferragina and Manzini. This adds $o(n)$ space overhead and can be constructed in parallel using prefix sum algorithms in $O(\log n)$ time [22].

4.2.2 Encoding of the transformed text

We are using approximately 4 bits per k-mer, which is still above the lower bound, but gets close.

Huffman coding is a method of generating prefix codes for a given alphabet, which relies on entropy for compression. As we have mentioned earlier in § 2.4.2, most k-mers will have a single neighbor on each side. This means that we should use fewer bits to encode them. Huffman coding presents a good method of achieving that.

4.2.3 Satellite data

We may want to process additional data associated with k-mers as keys. This so called satellite data can be k-mer frequency, quality information or which directions reads take when mapped to the graph. The nodes in our structure are already sorted into a list and are thus assigned simple unique contiguous indices. Consequently, any data associated with them can be put in a simple array.

4.3 Implementation

First we looked into implementation of our data structure as part of Velvet [38], which is known for its large memory requirements. However, due to a very complicated interface to replace a membership data structure, we settled instead for GATB, which presents a much simpler interface for membership data structures.

Therefore, we present our implementation of the data structure and compare it to a Bloom filter based data structure in GATB. The whole project was implemented in C++.

4.3.1 GATB

GATB is a bioinformatics framework [13] that grew out of MINIA [10], a genome assembler. It consists of five main components [19]:

operating system abstraction provides abstraction for memory, thread, time and filesystem management,

genomic banks read/write support supplies methods for iterating over FASTA, FASTQ and other types of collections of genomic data,

kmers management implements various k-mer models, including tools for working with reverse complements,

de Bruijn graph management presents an interface and data structures for constructing de Bruijn graphs and performing operations on them, and

collections and design patters, a set of classes frequently used in programming, like iterators and vectors, which have been tailored to function well with the rest of the framework.

GATB's implementation of the de Bruijn graph uses the membership data structure method and allows the specific data structure to be replaced, as long as it supports the `contains(kmer)` operation.

We implemented a simple version of the data structure in a way that is compatible with GATB. Due to time limitations it does not completely integrate into the framework, but enough to compare its performance to GATB's existing graph data structure.

4.3.2 Implementation notes

With memory usage being a major concern, GATB relies heavily on templates and chooses the smallest data structures that can hold the data. Some code relies on Boost's variants to choose between different types, depending on the size of the data.

We simplified the data structure used for rank queries so that it does not use a lookup table, which can be very large in practice. It computes the last part of the lookup with a linear scan over the block, which turns out to be at least as fast as a lookup [20]. Block size of the rank data structure itself was fixed to 16 characters, which is enough even for large genomes and exactly fits into a `uint64_t`. For simplicity, Huffman coding was not implemented.

4.4 Benchmarking

4.4.1 Methods

We separately measured the times for construction of the data structure and queries on that data structure. The benchmarking process is illustrated in Figure 4.4.

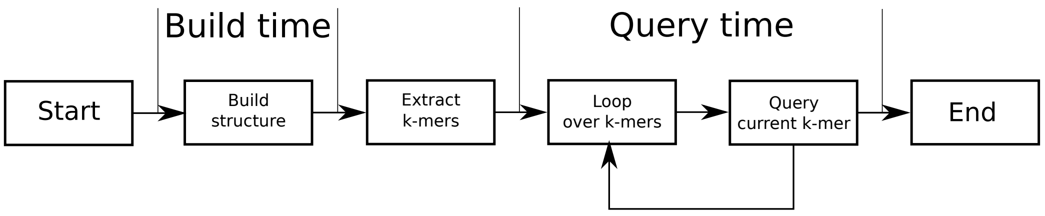


Figure 4.4: Benchmarking process flowchart.

All queries were k-mers found in the input data. This presents the worst case both for the original and for our data structure. For the data structure based on Bloom filters, it means a lot of checks to the false positive set, for kBWT, querying existing data means the algorithm cannot terminate early and must perform all k iterations.

The input data was a small subset of Potato virus Y NTN NIB V 151 VP1 (SRR1556761). Reads that contained unknown bases were removed and a small subset (400 000 reads) of the remaining data was used. Size of k-mers was set to 31. We timed how long does it take to build their respective data structures and perform queries for both the standard implementation and our implementation.

The test was performed on a computer with an Intel i7-6700 CPU, 16GB of RAM and a Western Digital Black 1TB HDD, running Funtoo Linux. Code was compiled with GCC 4.9.3 using `-O3` and `-funroll-loops`.

Time was measured using C++'s `<chrono>` library. It was measured for all the queries in the set of test k-mers together (see Figure 4.4). This was done because when we tried to measure individual queries separately, the compiler rearranged the order of calls as part of its optimization, which made timing inaccurate.

The average times were calculated by fitting a normal probability distribution to the measured times.

4.4.2 Results

Table 4.3 shows times required to build the data structure and then to perform the 718 queries (k-mers) determined by GATB's DSK [29] implementation to have high frequency. Results were averaged over 30 runs of both algorithms. Figure 4.5 shows a box plot of those times. During operation GATB reported memory usage of around 50 MiB for GATB's implementation and 500 MiB for kBWT.

4.5 Discussion

We created a cache-efficient data structure that encodes a de Bruijn graph in $\sigma \cdot n$ bits. When working with genomes it uses 4 bit per node, which approaches the theoretical lower bound. Only $o(n)$ additional data is required to bring the time complexity of queries from $O(k \cdot n)$ down to $O(k)$.

Algorithm	Part	Time $\pm$ 2σ $[ms]$
GATB	build	0.86 ± 0.014
	queries	3.17 ± 0.52
kBWT	build	29.96 ± 0.85
	queries	30.53 ± 5.71

Table 4.3: Times required to build a data structure and perform 718 queries.

Testing with Cachegrind revealed that kBWT causes fewer cache misses than Bloom filters. This can be attributed to the fact that all queries start at two of five possible points in the transformed text. These points are the boundaries between blocks with the same first character of the k-mer[2]. Bloom filters, on the other hand, are probabilistic and thus are not cache-efficient. However, having fewer misses did not in any way make up for the number of processor instructions required by the algorithm.

On other fronts our data structure did not prove to be very effective in practice with roughly 10 times slower queries and 10 times more memory usage than the original data structure used in GATB. This led us to recognize several disadvantages BWT-based structures have, compared to Bloom filter based ones.

The backward search used by kBWT performs k steps, each of which involves two rank operations. Each rank operation has to look up data in two separate arrays and reconstruct a block from a bit field. On the other hand, the Bloom filter performs d completely independent hashes and checks the obtained d addresses. Since $d \ll k$ and the rank operation is expensive, the slow performance was to be expected.

When it comes to memory usage, smaller graphs naturally take up less space. During the build phase, one of the easiest ways to reduce the number of k-mers, and thus nodes, is to remove less frequent k-mers, as those are usually the result of errors in the sequencing process. With Bloom filters this

[2]In the $k+1$-mers we used during construction, this was the second character.

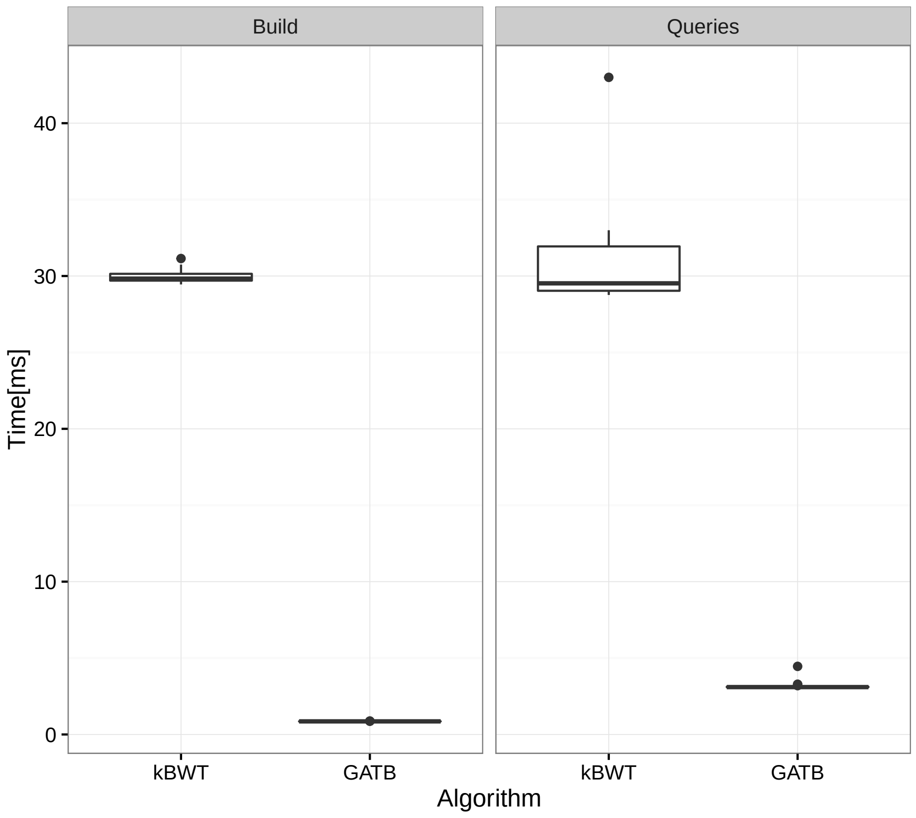


Figure 4.5: Box plot of measured times for two parts of both algorithms.

is simple: we do not insert the k-mers we do not want into the dictionary. GATB's implementation of DSK (disk streaming of k-mers) [29] takes care of removal. For kBWT the process is more complex, as our search algorithm depends on all suffixes of a k-mer to be present for it to work properly. The suffixes are normally provided by the prefixes of other k-mers. We can simply keep the less frequent k-mers and mark them as invalid (adding a 1 bit overhead), however, this does not yet save space. Another option is to add all prefixes of the removed k-mer, padded with `$`. This ensures all proper suffixes of other k-mers are provided. This is also unlikely to save space. We could also check which k-mers need replacement suffixes, but that would complicate the build process. It would probably require a k-mer dictionary of some sort, which is the exact problem we are trying to solve with kBWT.

In our implementation we didn't remove k-mers at all. The only k-mers that got special treatment were those at the end of reads, which are likely candidates for k-mers with missing suffixes.

One advantage BWT-derived algorithms have, is that they only define the maximal query length. The assembly process often benefits from performing the assembly using many different k' values, where $k' \leq k$, and combining the results. With BWT-based structures, assemblies using all k' values could be based off of the same data structure. This means the data structure would only have to be built once and could be reused many times.

Chapter 5

Conclusions

In this thesis we introduced a new space-efficient data structure, kBWT, which encodes the de Bruijn graph of a genome in near-optimal $\sigma = 4$ bits per k-mer. The size can be further reduced by exploiting the entropy of the connections present in the graph. With additional $o(n)$ of space it can be used to find a node's neighbors in $O(k)$ time. In practice the data structure did not prove to be as efficient as alternatives, though it used the cache more efficiently.

Biggest time- and space-wise improvements to the presented solution could be made in the build phase of kBWT, where the algorithm performs a lot of string comparisons during sorting. If these operations were implemented more efficiently, the build-time performance would be greatly improved. In our implementations we used some parts of the GATB framework, but its implementation of k-mers did not support all the required operations. An efficient implementation of these operations would improve both our algorithm and GATB as a framework in general.

Another improvement would be designing kBWT as a cache-aware data structure, which means that in the build phase, the algorithm processes data in chunks that fit into memory and only merges them into a complete data structure at the very end. Because k-mers can easily be partitioned into manageable sets based on their prefixes, each of those sets can be sorted individually. The merge operations could also be performed on those sets separately, since they would never cross set boundaries. This would also allow the building of the structure to be distributed over multiple computers or performed on one computer with less memory.

The process of genome assembly in general still has room for improvements in different directions. For example, the improvements in graph transformations include: better k-mer frequency threshold detection, especially in the area of repeats, more accurate bubble resolution to account for insertions or deletions, etc. The process of contiging could also be improved by combining multiple techniques like omnitigs [36] and long and paired-end reads at the same time. It could even use external data, like a set of known genes when resolving branches in the graph.

On a different note, little has been done in the area of parallel algorithms. A lot of the algorithms can trivially be made to work in parallel, but the authors seldom provide an analysis on the PRAM model. Such formalizations would allow the algorithms to easily be used on GPUs [6], vastly increasing their real-life performance.

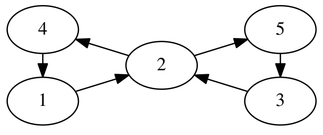
Currently, the design of most genome assemblers is monolithic; they implement a single new algorithm or idea. The algorithms are tested in isolation with, often, whole assemblers built around just one algorithm. Due to the imprecise nature of genome assembly, hybrid and modular approaches should be considered. The results from several algorithms can be combined into one assembly. This step could even employ machine learning techniques to more intelligently combine the different results.

To allow all these different approaches to graph transformations, contig assembly, algorithm design and the combinations of all three to be tested, we need a framework that is both fast and brings as little overhead as possible. Therefore, a continuation of work on minimizing the memory requirements is important. Our kBWT is only the tip of the iceberg in the area of BWT-based data structures. They seem to show promise in terms of cache efficiency and should be further researched.

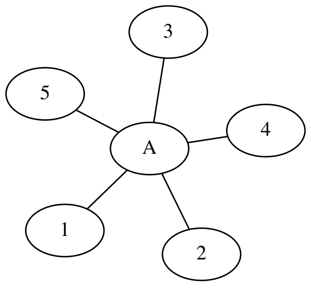
Appendix A

Example of the PRAM Eulerian tour algorithm

We show another example of the parallel Eulerian tour algorithm, this time with one tour found after the first part of the algorithm, to show how the certificates affect the table during the execution of the algorithm.



(a) Simple directed graph.



(b) Graph G_1 for the example. The spanning tree is obvious.

Figure A.1: Example graph and the corresponding G_1.

In Table A.1a we can see that the complete tour has already been formed. However, the algorithm does not yet know this. We perform the rest of the

Edge	`succ`$[e]$
$(1,2)$	$(2,5)$
$(2,4)$	$(4,1)$
$(2,5)$	$(5,3)$
$(3,2)$	$(2,4)$
$(4,1)$	$(1,2)$
$(5,3)$	$(3,2)$

(a) The state of `succ`$[e]$ after the first assignment.

Edge	`D`$[e]$	Tour name
$(1,2)$	$(1,2)$	A
$(2,4)$	$(1,2)$	A
$(2,5)$	$(1,2)$	A
$(3,2)$	$(1,2)$	A
$(4,3)$	$(1,2)$	A
$(5,1)$	$(1,2)$	A

(b) After tours are found and named.

Table A.1: Arrays `succ` and `D`.

steps as described earlier in § 2.5.2.

Table A.1b shows how the tour is named. Because there is only one tour it is "named" after the smallest edge. In our case, for clarity, we named it A.

Vertex	Tour	`certificate`$[e]$
1	A	$(4,1)$
2	A	$(1,2)$
3	A	$(4,3)$
4	A	$(2,4)$
5	A	$(2,5)$

Table A.2: Edges that certify that a vertex belongs to a tour.

We can see in Table A.2 that $(3,2)$ doesn't certify any of the vertices. This will affect what `succ` looks like before the final cleanup.

We can see in Table A.3a that the current sequence is:

$$((\mathbf{1},\mathbf{2}),(A,2),(2,A),(\mathbf{2},\mathbf{5}),(A,5),(5,A),(\mathbf{5},\mathbf{3}),(A,3),(3,A),(\mathbf{3},\mathbf{2}),(\mathbf{2},\mathbf{4}),$$
$$(A,4),(4,A),(\mathbf{4},\mathbf{1}),(A,1),(1,A),(\mathbf{1},\mathbf{2})).$$

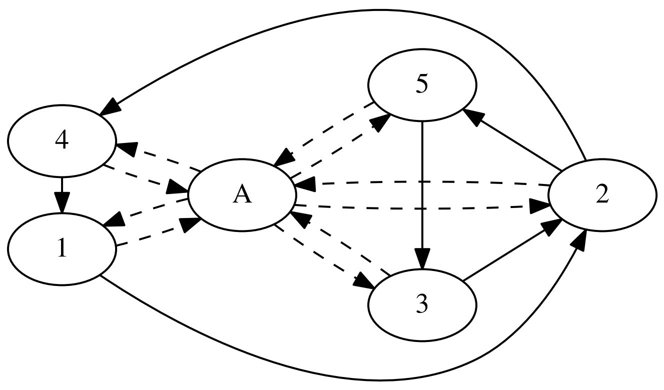


Figure A.2: T' and G on the same graph. The edges of T' are dashed.

$(3, 2)$ is followed by a "real" vertex. This is a consequence of the algorithm's use of certificates.

Finally, Table A.3b shows a complete tour, which is very similar to the one in our first example.

Edge	$\texttt{succ}[e]$
$(1,2)$	$(A,2)$
$(2,4)$	$(A,4)$
$(2,5)$	$(A,5)$
$(3,2)$	$(2,4)$
$(4,1)$	$(A,1)$
$(5,3)$	$(A,3)$
$(1,A)$	$(1,2)$
$(2,A)$	$(2,5)$
$(3,A)$	$(3,2)$
$(4,A)$	$(4,1)$
$(5,A)$	$(5,3)$
$(A,1)$	$(1,A)$
$(A,2)$	$(2,A)$
$(A,5)$	$(5,A)$
$(A,3)$	$(3,A)$
$(A,4)$	$(4,A)$

(a) The state of $\texttt{succ}[e]$ after computing the succession for T'.

Edge	$\texttt{succ}[e]$
$(1,2)$	$(2,5)$
$(2,4)$	$(4,1)$
$(2,5)$	$(5,3)$
$(3,2)$	$(2,4)$
$(4,1)$	$(1,2)$
$(5,3)$	$(3,2)$

(b) The state of $\texttt{succ}[e]$ after cleanup.

Table A.3: Array $\texttt{succ}$ with the Eulerian tour of T' and the final $\texttt{succ}$.

Bibliography

[1] B. Alberts, A. Johnson, J. Lewis, D. Morgan, M. Raff, K. Roberts, and P. Walter, *Molecular Biology of the Cell (6th edition).* Garland Science, 2014.

[2] M. Atallah and U. Vishkin, "Finding Euler tours in parallel," *Journal of Computer and System Sciences*, vol. 29, no. 3, pp. 330–337, 1984.

[3] E. Birney, J. A. Stamatoyannopoulos, A. Dutta, R. Guigó, T. R. Gingeras, E. H. Margulies, Z. Weng, M. Snyder, E. T. Dermitzakis, R. E. Thurman *et al.*, "Identification and analysis of functional elements in 1% of the human genome by the ENCODE pilot project," *Nature*, vol. 447, no. 7146, pp. 799–816, 2007.

[4] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[5] A. Bowe, T. Onodera, K. Sadakane, and T. Shibuya, "Succinct de Bruijn graphs," in *International Workshop on Algorithms in Bioinformatics*, 2012, pp. 225–235.

[6] A. Brodnik and M. Grgurovič, "Parallelization of ant system fpr GPU under the PRAM model," *Computing and Informatics*, to appear.

[7] A. Brodnik and J. I. Munro, "Membership in constant time and almost-minimum space," *SIAM Journal on Computing*, vol. 28, no. 5, pp. 1627–1640, 1999.

[8] N. G. de Bruijn, “A combinatorial problem,” in *Koninklijke Nederlandse Akademie v. Wetenschappen*, 1946.

[9] M. Burrows and D. J. Wheeler, “A block-sorting lossless data compression algorithm,” Digital Equipment Corporation, Tech. Rep., 1994.

[10] R. Chikhi and G. Rizk, “Space-efficient and exact de Bruijn graph representation based on a Bloom filter,” *Algorithms for Molecular Biology*, vol. 8, no. 1, p. 1, 2013.

[11] R. Chikhi, A. Limasset, S. Jackman, J. T. Simpson, and P. Medvedev, “On the representation of de Bruijn graphs,” in *International Conference on Research in Computational Molecular Biology*. Springer, 2014, pp. 35–55.

[12] T. H. Cormen, *Introduction to algorithms*, 3rd ed. MIT press, 2009.

[13] E. Drezen, G. Rizk, R. Chikhi, C. Deltel, C. Lemaitre, P. Peterlongo, and D. Lavenier, “GATB: Genome assembly & analysis tool box,” *Bioinformatics*, vol. 30, no. 20, pp. 2959–2961, 2014.

[14] R. Drmanac, S. Drmanac, G. Chui, R. Diaz, A. Hou, H. Jin, P. Jin, S. Kwon, S. Lacy, B. Moeur *et al.*, “Sequencing by hybridization (SBH): advantages, achievements, and opportunities,” in *Chip Technology*. Springer, 2002, pp. 75–101.

[15] D. Earl, K. Bradnam, J. S. John, A. Darling, D. Lin, J. Fass, H. O. K. Yu, V. Buffalo, D. R. Zerbino, M. Diekhans *et al.*, “Assemblathon 1: a competitive assessment of de novo short read assembly methods,” *Genome research*, vol. 21, no. 12, pp. 2224–2241, 2011.

[16] P. Ferragina and G. Manzini, “Indexing compressed text,” *Journal of the ACM*, vol. 52, no. 4, pp. 552–581, 2005.

[17] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan, "Compressing and indexing labeled trees, with applications," *Journal of the ACM*, vol. 57, no. 1, p. 4, 2009.

[18] M. L. Fredman, J. Komlós, and E. Szemerédi, "Storing a sparse table with O(1) worst case access time," *Journal of the ACM*, vol. 31, no. 3, pp. 538–544, 1984.

[19] "gatb.core-api-1.2.2: Design of the library," Web site, GATB, 2016, http://gatb-core.gforge.inria.fr/doc/api/design_page.html#coredesign, accessed 2016-10-06.

[20] R. González, S. Grabowski, V. Mäkinen, and G. Navarro, "Practical implementation of rank and select queries," in *In Poster Proceedings Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA'05) (Greece*, 2005, pp. 27–38.

[21] D. Gusfield, *Algorithms on strings, trees and sequences: computer science and computational biology.* Cambridge university press, 1997.

[22] J. JáJá, *An introduction to parallel algorithms.* Addison-Wesley Reading, 1992, vol. 17.

[23] D. Jungnickel and T. Schade, *Graphs, networks and algorithms.* Springer, 2008.

[24] D. Kutnjak, M. Rupar, I. Gutierrez-Aguirre, T. Curk, J. F. Kreuze, and M. Ravnikar, "Deep sequencing of virus-derived small interfering RNAs and RNA from viral particles shows highly similar mutational landscapes of a plant virus population," *Journal of virology*, vol. 89, no. 9, pp. 4760–4769, 2015.

[25] R. Luo, B. Liu, Y. Xie, Z. Li, W. Huang, J. Yuan, G. He, Y. Chen, Q. Pan, Y. Liu *et al.*, "SOAPdenovo2: an empirically improved memory-efficient short-read de novo assembler," *GigaScience*, vol. 1, no. 1, p. 1, 2012.

[26] M. Pop, "Genome assembly reborn: recent computational challenges," *Briefings in bioinformatics*, vol. 10, no. 4, pp. 354–366, 2009.

[27] D. M. W. Powers, "Parallelized QuickSort and RadixSort with optimal speedup," in *Proceedings of International Conference on Parallel Computing Technologies. Novosibirsk.*, 1991, pp. 167–176.

[28] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," in *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, vol. 31, no. 4, 2001.

[29] G. Rizk, D. Lavenier, and R. Chikhi, "DSK: k-mer counting with very low memory usage," *Bioinformatics*, p. btt020, 2013.

[30] M. Roberts, W. Hayes, B. R. Hunt, S. M. Mount, and J. A. Yorke, "Reducing storage requirements for biological sequence comparison," *Bioinformatics*, vol. 20, no. 18, pp. 3363–3369, 2004.

[31] "454 life sciences unveils new bench top sequencer, significant improvements to the Genome Sequencer FLX System including 1,000 bp reads for 2010," Press release, Roche, November 2009, http://www.roche.com/media/store/releases/med_dia_2009-11-19.htm, accessed 2016-09-19.

[32] M. Ronaghi, M. Uhlén, and P. Nyren, "A sequencing method based on real-time pyrophosphate," *Science*, vol. 281, no. 5375, p. 363, 1998.

[33] S. L. Salzberg, A. M. Phillippy, A. Zimin, D. Puiu, T. Magoc, S. Koren, T. J. Treangen, M. C. Schatz, A. L. Delcher, M. Roberts *et al.*, "GAGE: A critical evaluation of genome assemblies and assembly algorithms," *Genome research*, vol. 22, no. 3, pp. 557–567, 2012.

[34] F. Sanger and A. R. Coulson, "A rapid method for determining sequences in DNA by primed synthesis with DNA polymerase," *Journal of molecular biology*, vol. 94, no. 3, pp. 441–448, 1975.

[35] J. T. Simpson, K. Wong, S. D. Jackman, J. E. Schein, S. J. Jones, and I. Birol, "ABySS: a parallel assembler for short read sequence data," *Genome research*, vol. 19, no. 6, pp. 1117–1123, 2009.

[36] A. I. Tomescu and P. Medvedev, "Safe and complete contig assembly via omnitigs," *arXiv preprint arXiv:1601.02932*, 2016.

[37] C. Ye, Z. S. Ma, C. H. Cannon, M. Pop, and W. Y. Douglas, "Exploiting sparseness in de novo genome assembly," *BMC bioinformatics*, vol. 13, no. 6, p. 1, 2012.

[38] D. R. Zerbino and E. Birney, "Velvet: algorithms for de novo short read assembly using de Bruijn graphs," *Genome research*, vol. 18, no. 5, pp. 821–829, 2008.

Index