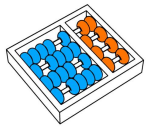
Universidade Estadual de Campinas
Instituto de Computação



Lucas Pansani Ramos

Draft genomes comparison with succinct de Bruijn graphs

Comparação de genomas incompletos usando grafos de de Bruijn sucintos

CAMPINAS
2022

Lucas Pansani Ramos

Draft genomes comparison with succinct de Bruijn graphs

Comparação de genomas incompletos usando grafos de de Bruijn sucintos

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Dissertation presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Computer Science.

Supervisor/Orientador: Prof. Dr. Guilherme Pimentel Telles
Co-supervisor/Coorientador: Prof. Dr. Felipe Alves da Louza

Este exemplar corresponde à versão final da Dissertação defendida por Lucas Pansani Ramos e orientada pelo Prof. Dr. Guilherme Pimentel Telles.

CAMPINAS
2022

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

R147d

Ramos, Lucas Pansani, 1998-
Draft genomes comparison with succinct de Bruijn graphs / Lucas Pansani Ramos. – Campinas, SP : [s.n.], 2022.

Orientador: Guilherme Pimentel Telles.
Coorientador: Felipe Alves da Louza.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Comparação de genomas. 2. Grafos de De Bruijn. 3. Filogenia. 4. Biologia computacional. I. Telles, Guilherme Pimentel, 1972-. II. Louza, Felipe Alves da, 1988-. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Informações para Biblioteca Digital

Título em outro idioma: Comparação de genomas incompletos usando grafos de de Bruijn sucintos
Palavras-chave em inglês:
Genome comparison
De Bruijn graphs
Phylogeny
Computational biology
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Guilherme Pimentel Telles [Orientador]
João Meidanis
Marinella Sciortino
Data de defesa: 05-07-2022
Programa de Pós-Graduação: Ciência da Computação

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0002-3699-2991
- Currículo Lattes do autor: http://lattes.cnpq.br/9921311673646745

Universidade Estadual de Campinas
Instituto de Computação

Lucas Pansani Ramos

Draft genomes comparison with succinct de Bruijn graphs

Comparação de genomas incompletos usando grafos de de Bruijn sucintos

Banca Examinadora:

- Profa. Dra. Marinella Sciortino
  Dipartimento di Matematica e Informatica - UNIPA
- Prof. Dr. João Meidanis
  Instituto de Computação - Unicamp
- Prof. Dr. Guilherme Pimentel Telles
  Instituto de Computação - Unicamp

A ata da defesa, assinada pelos membros da Comissão Examinadora, consta no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria do Programa da Unidade.

Campinas, 05 de julho de 2022

Acknowledgements

First of all, I would like to thank my advisors Guilherme Telles and Felipe Louza for the support, many teachings and friendship in the past two years. I really hope to work with you again in the future.

Second, thanks to my family and friends who never doubted my potential and were always very supportive when I needed.

Finally, I thank Prof. Nalvo Almeida for granting access to the machine used for the experiments and also, the internet, which made this graduation possible in these difficult times. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

Resumo

As melhorias contínuas em tecnologias de sequenciamento de DNA aumentaram a disponibilidade de genomas sequenciados. No entanto, devido às dificuldades para montar completamente um genoma, obtendo uma única cadeia para cada cromossomo, muitos genomas têm montagens incompletas (*draft*), em que cada cromossomo é representado por um conjunto de cadeias com informação parcial sobre a ordem relativa entre elas. Abordagens livres de alinhamento foram propostas recentemente para construir filogenias a partir da comparação de montagens draft de genomas, extraindo caminhos em grafos de *de Bruijn*. Neste trabalho investigamos métodos eficientes para comparar genomas incompletos por meio da comparação direta de grafos de *de Bruijn* coloridos representados de maneira sucinta usando a representação BOSS utilizando medidas baseadas na distribuição de similaridade de Burrows-Wheeler (BWSD). Nós propomos o algoritmo gcBB, este recebe um conjunto de genomas (em que cada genoma é representado por um conjunto de fragmentos) e sua saída são duas matrizes de distância derivadas da BWSD. Nós realizamos experimentos com conjuntos de genomas de Drosophilas e Vibrios, comparando as filogenias obtidas pelo gcBB com filogenias de referência.

Abstract

The continuous improvements in DNA sequencing technologies have increased the volume of available genomic data. Nevertheless, due to the difficulties in assembling a complete genome, obtaining a single sequence for each chromosome, many genomes are still incomplete (*draft*), in which each chromosome is represented as a set of sequences with partial information on their relative order. Recent alignment-free approaches have been proposed to construct phylogenies from the comparison between draft genomes, extracting paths from a de Bruijn graph. In this work we investigated efficient methods for directly comparing the succinct BOSS representation of colored de Bruijn graphs through measures based on the Burrows-Wheeler similarity distribution (BWSD). We propose the gcBB algorithm that takes a set of genomes (where each genome is a set of reads) and outputs matrices of distances derived from the BWSD. We performed experiments with sets of Drosophila and Vibrio genomes, comparing phylogenies obtained with gcBB to reference phylogenies.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

String comparison is an important task in Computer Science, being used in everyday text processing in editors like `Emacs` and `Vim` and in more involving tasks, for example, in WEB search engines, in the consolidation of non-structured databases, in plagiarism detection, in biological sequence comparisons and in the analysis of software variants.

We are often interested in computing some sort of similarity measure between strings. There exist many similarity measures for strings, which are computed in different forms, for example, as distance among vector representation of strings, as statistics calculated on groups of symbol co-occurrences, as edit distance, as alignment score, as substring tiling and others [37, 1].

The idea of comparing strings using compression-based measures such as the NCD [6] originates from the works of Kolmogorov and Chaitin on minimum algorithmic descriptions of strings. Similarity measures based on the Burrows-Wheeler Transform (BWT) [5], as the eBWT-based distances [28] and the Burrows-Wheeler Similarity Distribution (BWSD) [41], are particularly attractive because the BWT provides a self-index and can be computed in linear time on the string length.

In Bioinformatics, the standard form of calculating similarity between biological sequences is through alignments. Alignments, when used with amino acids and nucleotides evolution models, provide a similarity measure that reflects the evolutionary distance between molecules. The huge amount of currently available data and the quadratic cost of the algorithms to calculate alignments between strings ($O(nm)$ time for two strings of lengths n and m) triggered the development of faster alternatives such as heuristics, parallel algorithms and other distance measures, including alignment-free strategies [26].

In the process of sequencing a genome, a large amount of short strings (reads) from random fragments of the DNA is obtained and then assembled based on overlaps among the reads. The coverage information, that is, the number of times each DNA letter was read, can improve the ability of genome assembling software to handle ambiguities during the assembly process, either due to sequencing errors or due to repeated genomic regions. Representing the set of strings as a graph is an early stage of the genome assembly process, and a series of algorithms then extracts paths and connectivity information from the graph to obtain a tentative sequence or arrangement of sequences of the whole genome.

Many assemblers are based on the *de Bruijn graph* (*e.g.* [22, 21, 38]), that may be stored succinctly in the BOSS representation [4], which is based on the BWT. Colors may

be added to the edges of a de Bruijn graph, enabling them to represent a set of strings from distinct genomes.

Improvements in DNA sequencing technology have increased the throughput and reduced its cost. However, completely assembling a genome is a difficult task and many sequenced genomes are in a draft state, that is, a set of strings (contigs) that may be partially ordered (scaffolds) instead of a single string for each chromosome. Recent approaches have been proposed to compare de Bruijn graphs of draft genomes [25, 31].

The goal of this work was to investigate a space-efficient method for comparing draft genomes. More specifically, we introduce the gcBB, a space-efficient algorithm to compare genomes using the BOSS representation and the BWSD. Given a set of genomes, we represent each pair of genomes as a colored graph using the BOSS representation, where each edge is associated with a color corresponding to the genome it came from. On the colored de Bruijn graph, we use the BWSD to measure the similarity and compare the genomes. We are interested in the similarity based on the BOSS graph representations of the genomes.

Our measures were assessed comparing the genomes of 12 Drosophila species, obtained from FlyBase [40]. We also tested our algorithm with the genomes of Vibrio bacteria [39]. With these set of genomes we compared our method with the recent works presented by Lyman *et al.* [25] and Polevikov and Kolmogorov [31]. These comparisons had the aim of reconstructing a phylogenetic tree using the distance matrices produced by our algorithm and comparing them to referenced trees.

1.1 Organization

This dissertation is organized as follows. In Chapter 2, we define the data structures needed for completely understanding this work. In Chapter 3 we present the genome assembly process and the de Bruijn graph representation of genomes. Also, we introduce the BOSS representation and the related works. Chapter 4 presents the problem formulation and the solution proposed using our Space-Efficient Genome Comparison using the BOSS representation and the BWSD similarity measure algorithm (gcBB). The experiments are presented in Chapter 5. Finally, Chapter 6 concludes the work and provides possible future research problems.

Chapter 2

Background

This chapter presents the terminology and definitions that will be used throughout the text.

2.1 Terminology and Definitions

A *string* is the juxtaposition of symbols from an *alphabet*. We denote an ordered and finite alphabet by Σ, its *size* by σ and the *set of all strings* over Σ by Σ^*. The *length* of a string is the number of symbols in it. Let S be a string of length n in Σ^*. We index its symbols from 1 to n. A *substring* of S is $S[i, j] = S[i] \dots S[j]$, with $1 \leq i \leq j \leq n$. The substring $S[1, i]$ is referred to as a *prefix* of S and $S[i, n]$ is referred to as a *suffix* of S.

The *i-th circular rotation* (or *conjugate* or simply *rotation*) of a string S is the string $S[i + 1] \dots S[n]S[1] \dots S[i]$. Clearly when $i = 0$ we have the original string. A string of length n has n possible rotations. We say that S is *repetitive* if there exists a string w and an integer $k > 1$ such that $S = w^k = \underbrace{ww \dots w}_{k}$, otherwise S is *primitive*. If a string is primitive, all of its rotations are pairwise distinct.

For clarity of definitions and operations on the data structures it is convenient to use a special marker symbol \$ in the end of S. This symbol does not occur elsewhere in S and is the smallest symbol in Σ. Therefore, S is always primitive.

Let c represent a symbol of the alphabet and let S be a string of length n. We define $rank_c(S, i)$ as the number of occurrences of symbol c in $S[1, i]$. We define $select_c(S, i)$ as the position of the i-th symbol c in S, provided that if such symbol does not exist then $select_c(S, i)$ is equal to $n + 1$.

We refer to string of interest to be found in another given string as a *pattern*. A *text index* is a data structure built over a string that enables efficient searches for patterns in the string [12].

2.2 Data Structures

2.2.1 Suffix Array

The *suffix array* [27] of a string S of length n is the integer array SA containing a permutation of values in $[1, n]$ that gives the lexicographic order of all suffixes of S, that is,

$$S[\mathrm{SA}[1], n] < S[\mathrm{SA}[2], n] < \ldots < S[\mathrm{SA}[n], n]$$

For example, the suffix array of string $S = \mathrm{abracadabra\$}$ is shown in Table 2.1.

All the occurrences of a pattern $P[1, m]$ in the string S can be found in $O(m \log n)$ time by performing a binary search on $\mathrm{SA}[1, n]$.

The suffix array can be computed in linear time using $O(\sigma \log n)$ bits of working space [30], which is optimal for alphabets of constant size $\sigma = O(1)$.

2.2.2 LCP Array

By $\mathrm{lcp}(S_1, S_2)$ we denote the length of the *longest common prefix* of two strings S_1 and S_2 in Σ^*. The LCP array of a string S of length n is the array of integers LCP containing the lcp of consecutive suffixes in the suffix array, that is,

$$\mathrm{LCP}[i] = \begin{cases} \mathrm{lcp}(S[\mathrm{SA}[i], n], S[\mathrm{SA}[i-1], n]) & \text{if } 1 < i \leq n \\ 0 & \text{if } i = 1 \end{cases} \tag{2.1}$$

For example, the LCP array of string $S = \mathrm{abracadabra\$}$ is shown in Figure 2.1.

i	SA	LCP	$S[\mathrm{SA}[i], n]$
1	12	0	\$
2	11	0	a\$
3	8	1	abra\$
4	1	4	abracadabra\$
5	4	1	acadabra\$
6	6	1	adabra\$
7	9	0	bra\$
8	2	3	bracadabra\$
9	5	0	cadabra\$
10	7	0	dabra\$
11	10	0	ra\$
12	3	2	racadabra\$

Figure 2.1: Suffix array and LCP array for $S = \mathrm{abracadabra\$}$. Note that the list of sorted suffixes shown in the last column is not part of the suffix array or LCP array.

2.2.3 Burrows-Wheeler Transform

The *Burrows-Wheeler Transform* (BWT) [5] of a string S is a reversible transformation of S that permutes its symbols. The resulting string, denoted by BWT, often allows better

compression because equal symbols tend to be clustered in the BWT.

The BWT is defined as the sequence of symbols in S that precede the rotations of S in lexicographic order. We can obtain BWT by sorting all n rotations of S in a matrix $\mathcal{M}$, and taking the last column, called L, as BWT. We also call the first column of $\mathcal{M}$ by F. This method is illustrated in Figure 2.2 for S = abracadabra$.

rotations		F L
abracadabra$		$abracadabra
bracadabra$a		a$abracadabr
racadabra$ab		abra$abracad
acadabra$abr		abracadabra$
cadabra$abra		acadabra$abr
adabra$abrac		adabra$abrac
dabra$abraca	sorting →	bra$abracada
abra$abracad		bracadabra$a
bra$abracada		cadabra$abra
ra$abracadab		dabra$abraca
a$abracadabr		ra$abracadab
$abracadabra		racadabra$ab
		$\mathcal{M}$

Figure 2.2: The left matrix shows all rotations of S, with rotated prefixes colored in red. The right matrix shows the rotations in lexicographic order. Column F has the symbols of S in lexicographic order, while column L corresponds to BWT = ard$rcaaaabb.

We observe that sorting the rotations is equivalent to sorting the suffixes of S, since $S[n] = \$$ inhibits comparisons past the end of S. Hence, we can define the BWT in terms of the suffix array as

$$\text{BWT}[i] = \begin{cases} S[\text{SA}[i] - 1] & \text{if SA}[i] \neq 1 \\ \$ & \text{otherwise} \end{cases} \tag{2.2}$$

Figure 2.3 shows the suffix array and the BWT for S = abracadabra$. Note that the last column of the table in Figure 2.3 contains the suffixes of S. Taking the first symbol of each one of these suffixes we obtain a column that corresponds to column F of $\mathcal{M}$.

It is not obvious how to reconstruct the original string from the BWT. The key to the reversibility is the following function.

Definition 2.2.1 (LF-mapping) *Let $LF : \{1, \ldots, n\} \rightarrow \{1, \ldots, n\}$ be a function defined as: if $L[i] = c$ is the k-th occurrence of symbol c in column L, then $LF(i) = j$ is the position such that $F[j]$ is the k-th occurrence of c in column F.*

We define the *counter array* C as the integer array of size σ where $C[c]$ stores the number of symbols smaller than c in a string S, where c represents a symbol of the alphabet. We can compute the LF-mapping of symbol c at position i in L to its corresponding position in F as

$$LF(i) = C[c] + rank_c(L, i)$$

i	SA	BWT	LCP	$S[\text{SA}[i], n]$
1	12	a	0	\$
2	11	r	0	a\$
3	8	d	1	abra\$
4	1	\$	4	abracadabra\$
5	4	r	1	acadabra\$
6	6	c	1	adabra\$
7	9	a	0	bra\$
8	2	a	3	bracadabra\$
9	5	a	0	cadabra\$
10	7	a	0	dabra\$
11	10	b	0	ra\$
12	3	b	2	racadabra\$

Figure 2.3: Suffix array, BWT and LCP array for $S =$ abracadabra\$.

Remember that L represents the BWT, therefore this equation can also be written as

$$LF(i) = C[c] + rank_c(\text{BWT}, i)$$

The original string may be reconstructed backwards from the BWT starting with $S[n] = \$$, $j = n - 1$, $i = 1$, and repeatedly setting $S[j] = \text{BWT}[i]$, $i = LF(i)$, and $j = j - 1$. This procedure is executed $n - 1$ times.

2.2.4 FM-index

The FM-index [13, 14] is a text index built on top of the BWT and on the *backward search* algorithm. It can efficiently *count* and *locate* the occurrences of a pattern in the string S. The main components of the FM-index are:

- the BWT and the *backward search* algorithm;
- the *wavelet tree* [15], which solves $rank_c(\text{BWT}, i)$ queries for any $c \in \Sigma$ in $O(\log \sigma)$ time; and
- the *counter array* $C[1, \sigma]$.

The *backward search* algorithm, shown in Algorithm 1, finds a range of positions $[s, e]$ such that the symbols in $\text{BWT}[s, e]$ precede the occurrences of a pattern $P[1, m]$ in $S[1, n]$. In other words, exactly the rows $s, s+1, \ldots, e$ of $\mathcal{M}$ start with the occurrences of P in S.

For example, suppose we are searching for $P =$ bra in $S =$ abracadabra\$. Intuitively, we first search for the range in $\mathcal{M}$ that contains rows (suffixes) starting with $P[m, m] =$ a, see Figure 2.2. With $i = m$ and $c =$ a we evaluate

$$s = C[\text{a}] + rank_{\text{a}}(\text{BWT}, 1) = 1 + 1 = 2$$
$$e = C[\text{a}] + rank_{\text{a}}(\text{BWT}, 12) = 1 + 5 = 6$$

Then, we have that rows in $[2, 6]$ start with $P[m] =$ a.

Algorithm 1: `backward_search`

Input: $P[1, m]$
Output: $[s, e]$

```
1  s = 1;
2  e = n;
3  i = m;
4  while i ≥ 1 and s ≤ e do
5  |  c = P[i];
6  |  s = C[c] + rank_c(BWT, s);
7  |  e = C[c] + rank_c(BWT, e);
8  |  i = i − 1;
9  end
10 return [s, e];
```

Next, we search for the range of rows starting with $P[m-1, m] = \text{ra}$. We look for the positions in $[2, 6]$ that are preceded by $c = \text{r}$ evaluating

$$s = C[\text{r}] + rank_{\text{r}}(\text{BWT}, 2) = 10 + 1 = 11$$
$$e = C[\text{r}] + rank_{\text{r}}(\text{BWT}, 6) = 10 + 2 = 12$$

Finally, we find the range of rows with prefix $P[m-2, m] = \text{bra}$ in $\mathcal{M}$. Given the range $[11, 12]$, we look for the rows preceded by $c = \text{b}$ evaluating

$$s = C[\text{b}] + rank_{\text{b}}(\text{BWT}, 11) = 6 + 1 = 7$$
$$e = C[\text{b}] + rank_{\text{b}}(\text{BWT}, 12) = 6 + 2 = 8$$

At the end, the resulting range $[7, 8]$ indicates that $P = \text{bra}$ occurs twice in S at rows 7 and 8 of matrix $\mathcal{M}$. Note that these rows are equivalent to the suffixes starting in SA[7] and SA[8]. Hence, we can use the suffix array to locate the occurrences of P in S.

Counting queries can be solved in $O(m \log \sigma)$ time, whereas all the *occ* occurrences of P can be located in $O(m \log \sigma + occ)$ time, using extra $O(n \log n)$ bits for the suffix array. The total space required by the FM-index is $O(n \log \sigma)$ bits for the wavelet tree (which encodes BWT), plus $O(\sigma \log n)$ bits for the counter array.

There exist alternatives to reduce the working space of the FM-index, for example with the sampled suffix array (SSA) [14] at the cost of additional running time.

2.2.5 String collections

Let $\mathcal{S} = \{S_1, S_2, \ldots, S_d\}$ be a collection of d strings of lengths $n_1, n_2, \ldots, n_d$. We define S^{cat} as the concatenation of all strings in $\mathcal{S}$ as

$$S^{cat} = S_1[1, n_1 - 1]\$_1 S_2[1, n_2 - 1]\$_2 \cdots S_d[1, n_d - 1]\$_d$$

that is, each terminal symbol \$ is replaced by a (separator) symbol $\$_i$, with $\$_i < \$_j$ if and only if $i < j$. The length of S^{cat} is $N = \sum_{i=1}^{d} n_i$.

We define the *context* of a suffix $S^{cat}[i, N]$ as the substring $S^{cat}[i, j]$ such that $S^{cat}[j]$ is the first occurrence of some symbol $\$_k$ in $S^{cat}[i, N]$.

The suffix array for a collection $\mathcal{S}$ is the suffix array SA$[1, N]$ computed for S^{cat}.

The *document array* is commonly used when indexing string collections. The document array is an array of integers DA that stores which document each suffix in SA "belongs" to. More formally, DA$[i] = j$ if S^{cat}[SA$[i], N]$ has the context that ends with $\$_j$.

The BWT can also be constructed for string collections. The BWT for $\mathcal{S}$ can be obtained in linear time from the SA of the concatenated string using the following generalization of Equation 2.2 for S^{cat}

$$\text{BWT}[i] = \begin{cases} S^{cat}[\text{SA}[i] - 1] & \text{if SA}[i] \neq 1 \\ \$_d & \text{otherwise} \end{cases} \tag{2.3}$$

For example, given the strings $S_1 =$ abra and $S_2 =$ cadabra, Figure 2.4 shows the arrays SA and DA, the BWT and the contexts of S^{cat}.

i	SA	DA	BWT	context
1	5	1	a	$\$_1$
2	13	2	a	$\$_2$
3	4	1	r	a$\$_1$
4	12	2	r	a$\$_2$
5	1	1	$\$_2$	abra$\$_1$
6	9	2	d	abra$\$_2$
7	7	2	c	adabra$\$_2$
8	2	1	a	bra$\$_1$
9	10	2	a	bra$\$_2$
10	6	2	$\$_1$	cadabra$\$_2$
11	8	2	a	dabra$\$_2$
12	3	1	b	ra$\$_1$
13	11	2	b	ra$\$_2$

Figure 2.4: Suffix array, the document array and the BWT of $\mathcal{S} = \{\text{abra}, \text{cadabra}\}$ which is aarr$\$_2$dcaa$\$_1$abb.

In practice, when we have DA, the indexes $k \in [1, d]$ of separator symbols BWT$[i] = \$_k$ can be recovered (circularly) from DA$[i]$ and we can replace each $\$_k$ by \$ in BWT, which fits in $N \log \sigma$ bits.

2.3 Burrows-Wheeler Similarity Distribution

The BWT of two strings $\{S_1, S_2\}$ can be used to compute similarity measures between S_1 and S_2 based on the observation that the more the symbols of S_1 and S_2 are intermixed in the BWT, the greater the number of substrings shared by them and the more similar they are.

The Burrows-Wheeler similarity distribution (BWSD) between S_1 and S_2, denoted by BWSD(S_1, S_2), is a probability mass function defined as follows [41].

Given BWT, we define a bitvector α of size $n_1 + n_2$ such that

$$\alpha[p] = \begin{cases} 0 & \text{if BWT}[p] = \$_2 \text{ or BWT}[p] \in S_1 \\ 1 & \text{if BWT}[p] = \$_1 \text{ or BWT}[p] \in S_2 \end{cases} \tag{2.4}$$

The array α can be represented as a sequence of runs

$$r = 0^{k_1}1^{k_2}0^{k_3}1^{k_4}\ldots 0^{k_m}1^{k_{m+1}}$$

where i^{k_j} means that i repeats k_j times, and only k_1 and k_{m+1} may be zero. The largest possible value for k_j is $k_{\max} = \max(n_1, n_2)$.

Let t_n be the number of occurrences of value $k_j = n$. Let $s = t_1+t_2+\ldots+t_{k_j}+\ldots+t_{k_{\max}}$. The BWSD$(S_1, S_2)$ is the probability mass function

$$P\{k_j = k\} = t_k/s \text{ for } k = 1, 2, \ldots, k_{\max}.$$

Given BWSD(S_1, S_2), the following similarity measures were defined between S_1 and S_2.

Definition 2.3.1 $D_M(S_1, S_2) = E(k_j) - 1$, *where* $E(k_j)$ *is the expectation of* $BWSD(S_1, S_2)$.

Definition 2.3.2 $D_E(S_1, S_2) = -\sum_{k\geq 1, t_k \neq 0}(t_k/s)\log_2(t_k/s)$ *is the Shannon entropy of* $BWSD(S_1, S_2)$.

Note that if S_1 and S_2 are equal, we have the α bitvector as

$$\alpha = \{0, 1, 0, 1, \ldots, 0, 1\}$$

and $P\{k_j = 1\} = \frac{n_1+n_2}{n_1+n_2} = 1$, $D_M(S_1, S_2) = 0$ and $D_E(S_1, S_2) = 0$.

Also, if the α obtained from BWT(S_1, S_2) is equal to the α obtained from BWT(S_2, S_1), then both have the same BWSD and $D_E(S_1, S_2) = D_E(S_2, S_1)$ and $D_M(S_1, S_2) = D_M(S_2, S_1)$.

For example, given the strings $S_1 =$ abra and $S_2 =$ cadabra, we have

$$\begin{aligned} \text{BWT} &= \text{aarr\$}_2\text{dcaa\$}_1\text{abb} \\ \alpha &= \{0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1\} \\ r &= 0^1 1^1 0^1 1^1 0^1 1^2 0^1 1^3 0^1 1^1 \end{aligned}$$

It follows that $t_1 = 8$, $t_2 = 1$, $t_3 = 1$, $t_4, \ldots, t_{k_{\max}} = 0$ and $s = 10$. The BWSD(S_1, S_2) is

$$\begin{aligned} P\{k_j = 1\} &= 8/10 \\ P\{k_j = 2\} &= 1/10 \\ P\{k_j = 3\} &= 1/10 \end{aligned}$$

Calculating the distances $D_M(S_1, S_2)$ and $D_E(S_1, S_2)$ we obtain

$$\begin{aligned} D_M(S_1, S_2) &= (1(8/10) + 2(1/10) + 3(1/10)) - 1 \\ &= 1.3 - 1 \\ &= 0.3 \end{aligned}$$

and

$$\begin{aligned} D_E(S_1, S_2) &= -(0.8 \log_2(0.8) + 0.1 \log_2(0.1) + 0.1 \log_2(0.1)) \\ &= -(-0.92193) \\ &= 0.92183 \end{aligned}$$

The BWSD between all pairs of strings S_i and S_j in $\mathcal{S} = \{S_1, S_2, \ldots, S_d\}$ can be computed in $O(dN)$ time [24] given the BWT of $S^{cat}[1, N]$.

Chapter 3

Genome Comparison

3.1 Genome assembly

Genome assembly is the task of reconstructing the sequence of nucleotides that compose molecules of DNA in the cell of an organism, using as input a set of short DNA sequences, called *reads*, obtained from the sequencing process [18]. Most sequencing processes produce reads that cover each DNA molecule multiple times.

The *coverage* is the number of reads that include a given nucleotide in the reconstructed sequence. The average coverage is typically between 30× and 100× in whole genome sequencing projects. The coverage is used by genome assemblers to solve ambiguities during the reconstruction. Moreover, coverage is directly related to the existence of repeated regions in the genome and to sequencing errors.

The *de novo* genome assembly process reconstructs a genome based exclusively on the information in the set of reads. This process is required when there is no reference genome to which reads may be mapped.

Given a set of reads, a genome assembly software joins the reads through the identification of overlapping regions among them producing continuous sequences, called *contigs*. These contigs, together with known gaps among them, may be linked into *scaffolds*, as illustrated on Figure 3.1. Information regarding the number of reads that overlap may also be used in the assembly process. Such reconstruction is challenging due to repetitive sequences, polymorphisms in DNA sequences, lack of coverage and sequencing errors that introduce ambiguities and limit the length of the contigs that assemblers can build [2].

Several approaches have been proposed to assembly genomes based on different assembly graphs, such as overlap graphs [33], de Bruijn graphs [8], string graphs [38] and repeat graphs [20]. These assembly graphs can be useful for gene discovery, structural variation analysis, hybrid assembly and other applications [36].

3.2 de Bruijn graphs

A *de Bruijn graph*, is a directed graph that represents all possible strings of length k (also known as k-mers) [8]. We remark that, given an alphabet with σ symbols, there are σ^k distinct k-mers. The complete de Bruijn graph of order k of σ symbols is a directed

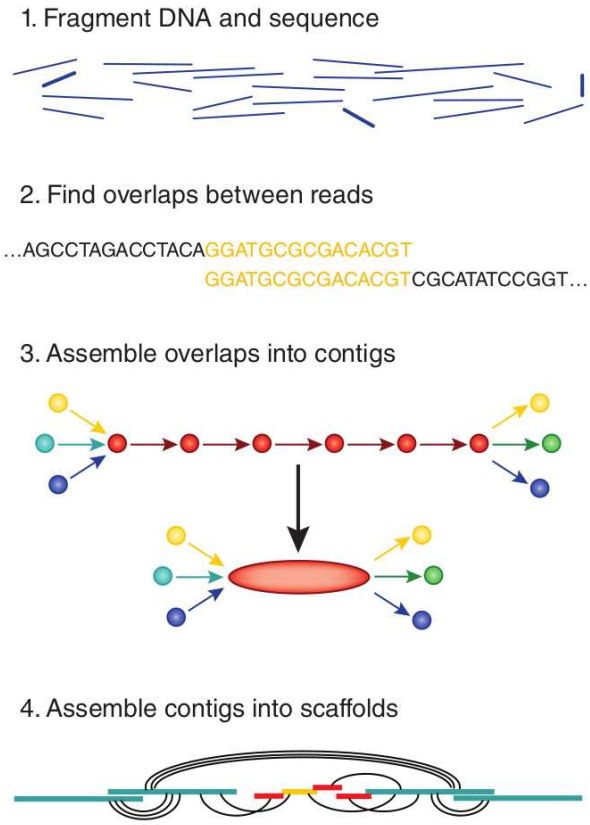


Figure 3.1: The *de novo* genome assembly pipelines. Figure from [2].

graph having a vertex for each k-mer. We denote the label associated to a vertex u by $\overrightarrow{u}$. There is an edge from u to v labelled $\overrightarrow{v}[k]$ if $\overrightarrow{u}[2][k] = \overrightarrow{v}[1][k-1]$. The de Bruijn graphs considered in this work are subgraphs of complete de Bruijn graphs.

Let $\mathcal{S} = \{S_1, S_2, \ldots, S_d\}$ be a string collection. We concatenate k symbols \$ at the beginning of each string in $\mathcal{S}$. We define a de Bruijn graph of order k of $\mathcal{S}$ as the de Bruijn graph with one vertex for each k-mer of a string in $\mathcal{S}$ and an edge for each pair of consecutive k-mers with an overlap of size $k-1$ in the strings in $\mathcal{S}$ [4]. Since there are k symbols \$ at the start of each string in $\mathcal{S}$, there will always be at least one directed path of length k to each *original* vertex (vertices representing k-mers without the symbols \$), where the concatenation of edges in this path corresponds to the k-mer associated to the final vertex of the path.

For example, given $\mathcal{S} = \{\$\$\$\texttt{TACACT}, \$\$\$\texttt{TACTCA}, \$\$\$\texttt{GACTGC}\}$ and $k = 3$, Figure 3.2 illustrates a de Bruijn graph of order k for $\mathcal{S}$. Starting at vertex \$\$\$ it is possible to spell all strings of $\mathcal{S}$ by traversing the graph concatenating the edge labels. Also, we remark that it is not necessary to store the vertex labels. Since every vertex represents a unique k-mer it is possible to recover the label of an original vertex u by walking backwards k edges starting from u (in any direction).

3.2.1 Colored de Bruijn graphs

The *colored de Bruijn graph (CdBG)* [17] generalizes the original formulation of the de Bruijn graph for a set of d string collections such that there exists a set $\mathcal{C} = \{c_1, c_2, \ldots, c_d\}$

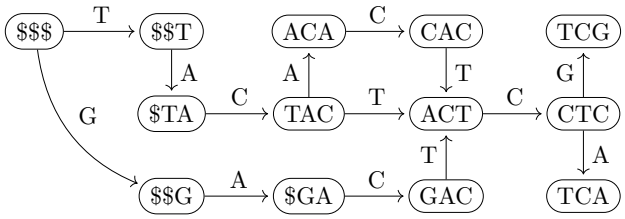


Figure 3.2: A de Bruijn graph for $\mathcal{S} = \{\$\$\$\texttt{TACACT}, \$\$\$\texttt{TACTCA}, \$\$\$\texttt{GACTGC}\}$ and $k = 3$. Starting at node `$$$` and walking through the edges labelled with `T,A,C,A,C,T` successively, we obtain the first string of the collection.

of d colors and all edges that were added by strings in collection i are colored with c_i.

For example, given collections $\mathcal{S}_1 = \{\$\$\$\texttt{TACACT}, \$\$\$\texttt{TACTCA}\}$ and $\mathcal{S}_2 = \{\$\$\$\texttt{GACTCG}\}$ and $k = 3$, Figure 3.3 shows the de Bruijn graphs for these two collections and the colored de Bruijn graph for $\{\mathcal{S}_1, \mathcal{S}_2\}$.

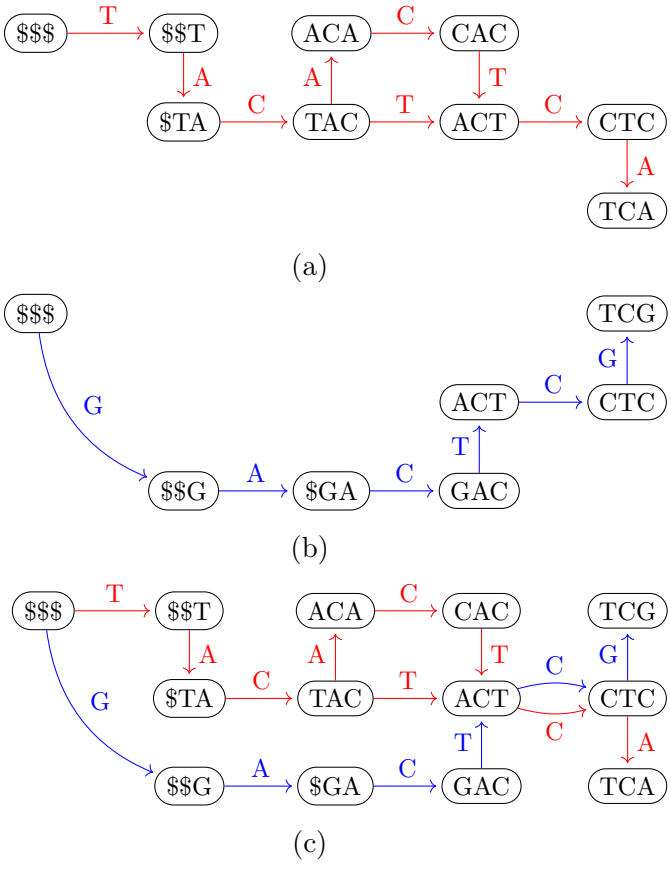


Figure 3.3: (a) de Bruijn graph for $\mathcal{S}_1 = \{\$\$\$\texttt{TACACT}, \$\$\$\texttt{TACTCA}\}$. (b) de Bruijn graph for $\mathcal{S}_2 = \{\$\$\$\texttt{GACTCG}\}$. (c) Colored de Bruijn graph for $\{\mathcal{S}_1, \mathcal{S}_2\}$, where red edges are from $\mathcal{S}_1$ and blue edges are from $\mathcal{S}_2$. We remark that only $\{\mathcal{S}_1, \mathcal{S}_2\}$ has colored edges, whereas $\mathcal{S}_1$ and $\mathcal{S}_2$ edges are colored for example purposes.

3.2.2 BOSS representation

BOSS [4] is a succinct representation of the de Bruijn graph. Let n and m be respectively the number of vertices and edges of a de Bruijn graph G. Consider that its set of vertices $v_1, v_2, \ldots, v_n$ is sorted according to the co-lexicographic order of their label, that is, the vertices are sorted in the lexicographic order of the reverse of their labels

$$\overleftarrow{v_i} = \overrightarrow{v_i}[k] \ldots \overrightarrow{v_i}[1] \,.$$

For example, given the string collection $\mathcal{S} = \{\$\$\$\texttt{TACACT}, \$\$\$\texttt{TACTCA}, \$\$\$\texttt{GACTCG}\}$ and $k = 3$, Figure 3.4 shows all possible k-mers obtainable from distinct $(k+1)$-mers of $\mathcal{S}$. Figure 3.5 shows the co-lexicographic sorting applied to those k-mers.

$$\left\{ \begin{array}{l} \$\$\$, \$\$\$, \$\$\texttt{T}, \$\texttt{TA}, \texttt{TAC}, \texttt{TAC}, \texttt{ACA} \\ \texttt{CAC}, \texttt{ACT}, \texttt{ACA}, \texttt{CTC}, \texttt{CTC}, \texttt{TCA}, \$\$\texttt{G} \\ \texttt{GAC}, \texttt{TCG} \end{array} \right\}$$

Figure 3.4: All possible k-mers obtained from distinct $(k+1)$-mers of $\mathcal{S} = \{\$\$\$\texttt{TACACT}, \$\$\$\texttt{TACTCA}, \$\$\$\texttt{GACTCG}\}$.

$\$\$\$$		$\$\$\$$
$\$\$\$$		$\$\$\$$
$\$\texttt{T}$		ACA
$\$\texttt{TA}$		TCA
TAC		$\$\texttt{GA}$
TAC		$\$\texttt{TA}$
ACA		CAC
CAC	co-lexicographic	GAC
ACT	sorting	TAC
CTC	→	TAC
CTC		CTC
TCA		CTC
$\$\$\texttt{G}$		$\$\$\texttt{G}$
$\$\texttt{GA}$		TCG
GAC		$\$\texttt{T}$
TCG		ACT

Figure 3.5: The left matrix shows all possible k-mers of $\mathcal{S}$. The right matrix shows the same k-mers sorted in co-lexicographic order.

We define *Node* as a conceptual matrix containing the co-lexicographically sorted k-mers. For each vertex v_i, we define W_i as the lexicographically sorted sequence of symbols of the edges outgoing v_i. If v_i has no outgoing edges then $W_i = \$$.

The BOSS representation is composed by the following components:

- The string $W[1, m] = W_1 W_2 \ldots W_n$. Observe that $|W| = |Node|$ and $Node[i]$ is the vertex from which the edge labeled $W[i]$ leaves.

- The bitvector $W^-[1, m]$ such that $W^-[i] = 0$ if there exists $j < i$ such that $W[j] = W[i]$ and the suffixes of length $k - 1$ of $Node[j]$ and of $Node[i]$ are identical, or $W^-[i] = 1$ otherwise.
- The bitvector $last[1, m]$ such that $last[i] = 1$ if $i = n$ or $Node[i]$ is different from $Node[i + 1]$, otherwise $last[i] = 0$.
- The counter array $C[1, \sigma]$ such that $C[c]$ stores the number of symbols smaller than the symbol c in the last column of the conceptual matrix *Node*.

The matrix *Node* and the string W from the previous example can be observed in Figure 3.6. Note that the rows in *Node* that correspond to any W_i have the same label, for $1 \leq i \leq n$. For example, for the vertex $v_i = \texttt{TAC}$, we have $W_i = \texttt{AT}$, and for $v_j = \texttt{TCG}$, $W_j = \$$.

i	*Node*	W
1	$$$	G
2	$$$	T
3	ACA	C
4	TCA	$
5	$GA	C
6	$TA	C
7	CAC	T
8	GAC	T
9	TAC	A
10	TAC	T
11	CTC	A
12	CTC	G
13	$$G	A
14	TCG	$
15	$$T	A
16	ACT	C

Figure 3.6: The *Node* matrix and the string W for collection $\mathcal{S} = \{\$\$\$\texttt{TACACT}, \$\$\$\texttt{TACTCA}, \$\$\$\texttt{GACTCG}\}$. Note that repeated k-mers correspond to a k-mer that has more than one outgoing edge in the graph. For example, vertex TAC is a k-mer obtained from the following k-mers: TACA and TACT.

Since we are working with DNA sequences, we have the alphabet $\Sigma = \{\texttt{A}, \texttt{T}, \texttt{C}, \texttt{G}, \texttt{N}, \$\}$ and $\sigma = 6$. Storing the string W takes $m\lceil \log_2 \sigma \rceil = 3m$ bits. The bitvectors W^- and *last* take $2m$ bits and the counter array C takes $\sigma \log m = 6 \log m$ bits. Therefore, the overall space to store the BOSS structure (for DNA sequences) is $5m + 6 \log m$ bits.

For example, the succinct representation of the de Bruijn graph for $\mathcal{S} = \{\$\$\$\texttt{TACACT}, \$\$\$\texttt{TACTCA}, \$\$\$\texttt{GACTCG}\}$ is illustrated in Figure 3.7 augmented with the *Node* matrix and with edges of the de Bruijn graph to ease the understanding.

There exists an LF-mapping between $Node[i]$ and $W[j]$ such that $LF(j) = i$ when $W^-[i] = 1$ and $last[j] = 1$, otherwise $LF(j)$ is undefined. This property enables the

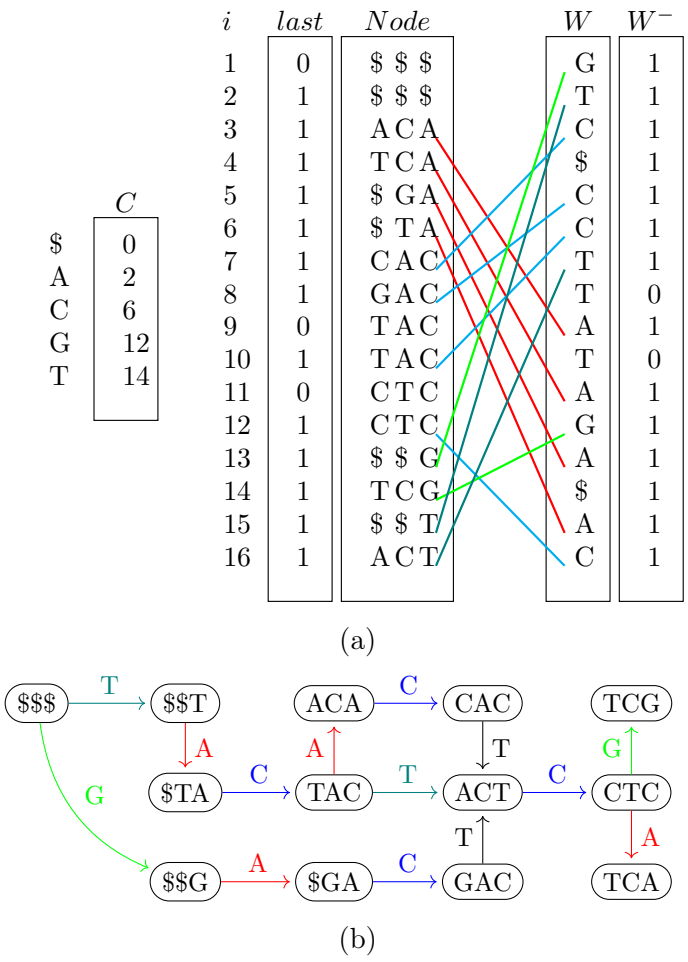


Figure 3.7: (a) BOSS representation for $\mathcal{S} = \{\$\$\$\texttt{TACACT}, \$\$\$\texttt{TACTCA}, \$\$\$\texttt{GACTCG}\}$ augmented with the *Node* matrix and with edges of the de Bruijn graph; (b) de Bruijn graph representing $\mathcal{S}$. Edges with the same color have the same symbol and edges colored black represent edges where $W^-[i] = 0$, that is, there is already one or more edge $W[j]$, such that $j < i$, mapped to the same vertex to which $W[i]$ is mapped.

navigation operations that will be defined in the next section. We can compute $LF(j)$ through

$$LF(j) = select_1(last, rank_1(last, C[c]) + r)$$

where $r = rank_c(W, j)$. For example, let $j = 16$. We have $W[j] = \texttt{C}$ and we compute

$$r = rank_{\texttt{C}}(W, 16) = 4$$

then we have

$$\begin{aligned} LF(j) &= select_1(last, rank_1(last, C[\texttt{C}]) + r) \\ &= select_1(last, rank_1(last, 6) + 4) \\ &= select_1(last, 5 + 4) \\ &= select_1(last, 9) \\ &= 12 \end{aligned}$$

Therefore we have that $LF(j) = 12$ and that v_{16} is a terminal edge of v_{12}, where $\overleftarrow{v_{12}} = \texttt{CTC}$.

3.2.3 BOSS operations

In this section we are going to present navigation operations in the BOSS representation. These operations will not be directly used in our algorithm and were included to illustrate the usage of the representation. The BOSS supports the following operations:

- $outdegree(\overleftarrow{v})$ returns the number of edges leaving vertex $\overleftarrow{v}$;
- $outgoing(\overleftarrow{v}, c)$ returns the vertex $\overleftarrow{u}$ if there is an edge $(\overleftarrow{v}, \overleftarrow{u})$ labeled with c in the graph, if there is any, otherwise $outgoing(\overleftarrow{v}, c)$ is undefined;
- $indegree(\overleftarrow{v})$ returns the number of edges arriving at vertex $\overleftarrow{v}$;
- $incoming(\overleftarrow{v}, c)$ returns the vertices $\overleftarrow{u_i}$ for which there is an edge $(\overleftarrow{u_i}, \overleftarrow{v})$ labeled with c in the graph, if there is any, otherwise $incoming(\overleftarrow{v}, c)$ is undefined.

From now on, we will use the position i of a vertex v in *Node* as the position of the last edge leaving v, that is, $last[i] = 1$. To facilitate the understanding of BOSS operations, we will use a new string W' instead of W and W^-. We define Σ^- as the set of size $|\Sigma|$ such that $\Sigma \cap \Sigma^- = \emptyset$ and for every $c \in \Sigma$ there is an element $c^- \in \Sigma^-$. We define $W'[i]$, for $i = 1, \ldots, m$, as $W'[i] = W[i]$ if $W^-[i] = 1$, or $W'[i] = W[i]^-$ if $W^-[i] = 0$.

outdegree($\overleftarrow{v}$): The $outdegree(\overleftarrow{v})$ operation can be computed as follows. By definition, the bitvector *last* has value 0 for every edge leaving vertex v, except for the last edge. Let i be referred to as the position of v, that is $Node[i] = \overleftarrow{v}$ and $last[i] = 1$. First we compute the position of the vertex preceding v, which is the first value 1 in *last* prior to position i, using a *select* query. A preceding vertex always exists due to the k-mer containing only \$ symbols. Then we compute $outdegree(\overleftarrow{v})$ by subtracting the preceding vertex position from position i as follows

$$outdegree(\overleftarrow{v}) = i - select_1(last, rank_1(last, i - 1))$$

For example, to compute $outdegree(\overleftarrow{v})$ where $\overleftarrow{v} = \texttt{TAC}$, the position i of $\overleftarrow{v}$ with

$Node[i] = \texttt{TAC}$ is $i = 10$ (see Figure 3.8). Therefore,

$$\begin{aligned} outdegree(\overleftarrow{v}) &= 10 - select_1(last, rank_1(last, 9)) \\ &= 10 - select_1(last, 7) \\ &= 10 - 8 = 2 \end{aligned}$$

i	$last$	$Node$	W'
1	0	\$ \$ \$	G
2	1	\$ \$ \$	T
3	1	A C A	C
4	1	T C A	\$
5	1	\$ G A	C
6	1	\$ T A	C
7	1	C A C	T
8	1	G A C	T-
9	0	T A C	A
10	1	T A C	T-
11	0	C T C	A
12	1	C T C	G
13	1	\$ \$ G	A
14	1	T C G	\$
15	1	\$ \$ T	A
16	1	A C T	C

Figure 3.8: *Outdegree* operation for $\overleftarrow{v} = \texttt{TAC}$. The red arrow indicates the *select* query.

outgoing$(\overleftarrow{v}, c)$**:** The $outgoing(\overleftarrow{v}, c)$ operation can be computed as follows. First we calculate the range of vertices labeled with $\overleftarrow{v}$, $R(\overleftarrow{v})$. Let i be the position of $\overleftarrow{v}$, that is $Node[i] = \overleftarrow{v}$ and $last[i] = 1$, we have that i is the last position of $R(\overleftarrow{v})$. To find the starting position of $R(\overleftarrow{v})$ we have to compute the position of the vertex $\overleftarrow{u}$ preceding $\overleftarrow{v}$ in $Node$ and add 1 to the result. The range of $\overleftarrow{v}$ is

$$R(\overleftarrow{v}) = [i - outdegree(\overleftarrow{v}) + 1, i]$$

Then, we check if there is an edge c leaving $\overleftarrow{v}$. To do that, we compute

$$j = select_c(W', rank_c(W', i))$$

If $j \in R(\overleftarrow{v})$, an edge labeled with c leaving $\overleftarrow{v}$ exists and $LF(j)$ returns the position in $Node$ of the vertex $\overleftarrow{u}$ in which this edge arrives. Otherwise, we check if there is an edge c^- leaving $\overleftarrow{v}$. We compute

$$j = select_{c^-}(W', rank_{c^-}(W', i))$$

In the case that $j \in R(\overleftarrow{v})$, there is an edge with c^- leaving $\overleftarrow{v}$, and for computing the position of vertex $\overleftarrow{u}$ in which this edge enters we have to find the largest position $j' < j$

where $W'[j'] = c$. To do that, we compute

$$j' = select_c(W', rank_c(W', j))$$

At the end, $LF(j')$ returns the position in $Node$ of the vertex $\overleftarrow{u}$.

For example, to compute $outgoing(\overleftarrow{v}, c)$ where $\overleftarrow{v} = \texttt{TAC}$ and $c = \texttt{A}$, the position i of $\overleftarrow{v}$ with $Node[i] = \texttt{TAC}$ is $i = 10$ (see Figure 3.7). The range is

$$\begin{aligned} R(\overleftarrow{v}) &= [10 - outdegree(\overleftarrow{v}) + 1, 10] \\ &= [10 - 2 + 1, 10] \\ &= [9, 10] \end{aligned}$$

Now, we compute j as

$$\begin{aligned} j &= select_{\texttt{A}}(W', rank_{\texttt{A}}(W', 10)) \\ &= select_{\texttt{A}}(W', 1) \\ &= 9 \end{aligned}$$

Since $j \in R(\overleftarrow{v})$ we can compute $LF(9) = 3$. Therefore, the destination vertex when leaving vertex labeled with TAC following edge A is $Node[3] = \texttt{ACA}$ (see Figure 3.9).

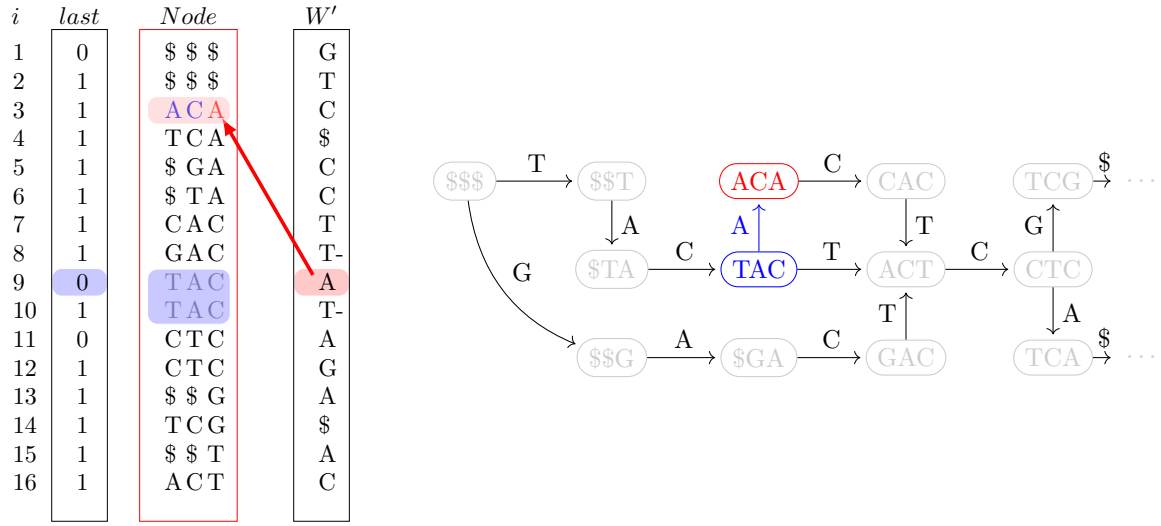


Figure 3.9: *Outgoing* operation for $\overleftarrow{v} = \texttt{TAC}$ and $c = \texttt{A}$. The red arrow indicates the mapping between W' and the $Node$.

Another example, now considering the case where $j \notin R(\overleftarrow{v})$ but there is a $c^- \in \Sigma^-$ in $R(\overleftarrow{v})$. Let us compute $outgoing(\overleftarrow{v}, c)$ for $\overleftarrow{v} = \texttt{TAC}$ and $c = \texttt{T}$. Since we know that $R(\overleftarrow{v}) = [9, 10]$ from the previous example, we compute j as follows

$$\begin{aligned} j &= select_{\texttt{T}}(W', rank_{\texttt{T}}(W', 10)) \\ &= select_{\texttt{T}}(W', 2) \\ &= 7 \end{aligned}$$

Note that $j \notin R(\overleftarrow{v})$, then we search for $c^- = \mathtt{T}^-$ in $R(\overleftarrow{v})$. We compute j as follows

$$\begin{aligned} j &= select_{\mathtt{T}^-}(W', rank_{\mathtt{T}^-}(W', 10)) \\ &= select_{\mathtt{T}^-}(W', 2) \\ &= 10 \end{aligned}$$

Now $j \in R(\overleftarrow{v})$ and the first $c \in \Sigma$ before $R(\overleftarrow{v})$ that corresponds to c^- appears in $j' = 7$, then we compute $LF(7) = 16$. Therefore, the destination vertex when leaving the vertex labeled with `TAC` following edge T is $Node[16] = \mathtt{ACT}$ (see Figure 3.10).

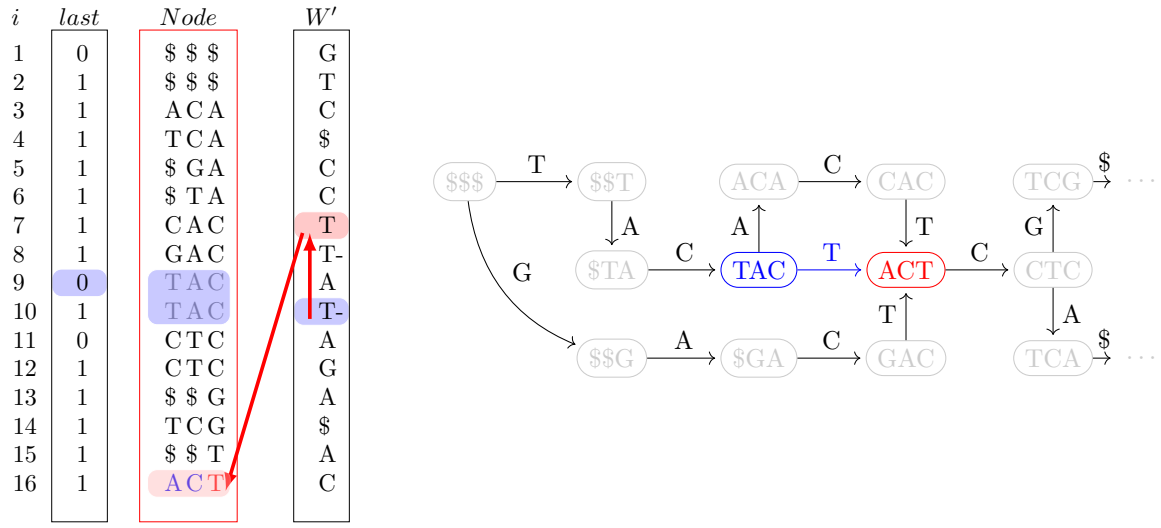


Figure 3.10: *Outgoing* operation for $\overleftarrow{v} = \mathtt{TAC}$ and $c = \mathtt{T}$. The red arrows indicate the search for c from c' and its mapping between W' and the *Node*.

indegree($\overleftarrow{v}$): Assume that i is the position of vertex $\overleftarrow{v}$ in *Node*, that is, $last[i] = 1$ and $Node[i] = \overleftarrow{v}$. The $indegree(\overleftarrow{v})$ operation can be computed as follows. By definition, if there is at least one incoming edge in $\overleftarrow{v}$, then there is an edge $W'[j]$ such that $LF(j) = i$. First, we have to find the value of j. Given the last symbol c of $Node[i]$, we count the number of symbols c up to position i in the last column of *Node*. This value can be obtained by subtracting i from the position of the first *Node* with last symbol equal to c, which is stored in $C[c]$. Recall that the vertices are sorted co-lexicographically. Then j can be obtained using the following *select* query

$$j = select_c(W', i - C[c])$$

If there are more than one incoming edge in $\overleftarrow{v}$, their symbols in W' should belong to Σ^- and their positions are larger than j in W'. Let c represent the edge at $W'[j]$. Clearly, we just have to count the number of occurrences of c^- between position j and the first c in W' after position j, if there is any.

Then we can compute $indegree(\overleftarrow{v})$ as follow

$$indegree(\overleftarrow{v}) = 1 + rank_{c^-}(W', select_c(W', j+1)) - rank_c(W', j)$$

For example, to compute $indegree(\overleftarrow{v})$ where $\overleftarrow{v} = \texttt{ACT}$, the position i of $\overleftarrow{v}$ where $last[i] = 1$ and $Node[i] = \overleftarrow{v}$ is $i = 16$ (see Figure 3.11). We have

$$\begin{aligned} j &= select_{\texttt{T}}(W', i - C[\texttt{T}]) \\ &= select_{\texttt{T}}(W', 16 - 14) \\ &= select_{\texttt{T}}(W', 2) = 7 \end{aligned}$$

Since $W'[7] = \texttt{T}$ we just have to count the number of occurrences of $\texttt{T}^-$ in W' between position 7 and $W'[7]$ next T, that is

$$\begin{aligned} indegree(\overleftarrow{v}) &= 1 + rank_{\texttt{T}^-}(W', select_T(W', i - C[\texttt{T}] + 1)) - rank_{\texttt{T}^-}(W', 7) \\ &= 1 + rank_{\texttt{T}^-}(W', select_{\texttt{T}}(W', 3)) - 0 \\ &= 1 + rank_{\texttt{T}^-}(W', 17) \\ &= 1 + 2 = 3 \end{aligned}$$

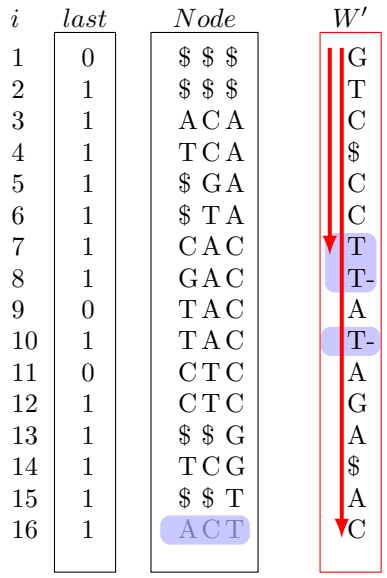


Figure 3.11: *Indegree* operation for $\overleftarrow{v} = \texttt{ACT}$. The red arrows indicate the *rank* queries over W'.

incoming($\overleftarrow{v}$): The $incoming(\overleftarrow{v})$ operation can be computed as follow. Using the same idea from $indegree(\overleftarrow{v})$, we have the first incoming edge of $\overleftarrow{v}$ in position j, clearly, $Node[j]$ is one incoming edge of $\overleftarrow{v}$. Next, we can iterate through the next incoming edges that belong to Σ^-. This will return all incoming edges of $\overleftarrow{v}$.

For example, to compute $incoming(\overleftarrow{v})$ where $\overleftarrow{v} = \texttt{ACT}$, we already know that the first incoming edge is at position $j = 7$, therefore $Node[7] = \texttt{CAC}$ has an incoming edge of $\overleftarrow{v} = \texttt{ACT}$. Now we have to iterate through the the next symbols T^-, which are in the range between $j + 1$ and the next T after $W'[7]$. Again, like in indegree we can compute

this range using

$$rank_{\mathtt{T}^-}(W', select_{\mathtt{T}}(W', select_{\mathtt{T}}(W', 16-14)+1))$$

Therefore, we have that the next $\mathtt{T}^-$ symbols are in the range [8,17]. Iterating on this range using select queries we have

$$select_{\mathtt{T}^-}(W', 1) = 8$$
$$select_{\mathtt{T}^-}(W', 2) = 10$$
$$select_{\mathtt{T}^-}(W', 3) = 17$$

Since 17 exceeds $|W'|$, we have that $Node[8] = \mathtt{GAC}$ and $Node[10] = \mathtt{TAC}$ also are edges incoming in $\overleftarrow{v} = \mathtt{ACT}$. Then, $incoming(v) = \{\mathtt{CAC}, \mathtt{GAC}, \mathtt{TAC}\}$ (see Figure 3.12).

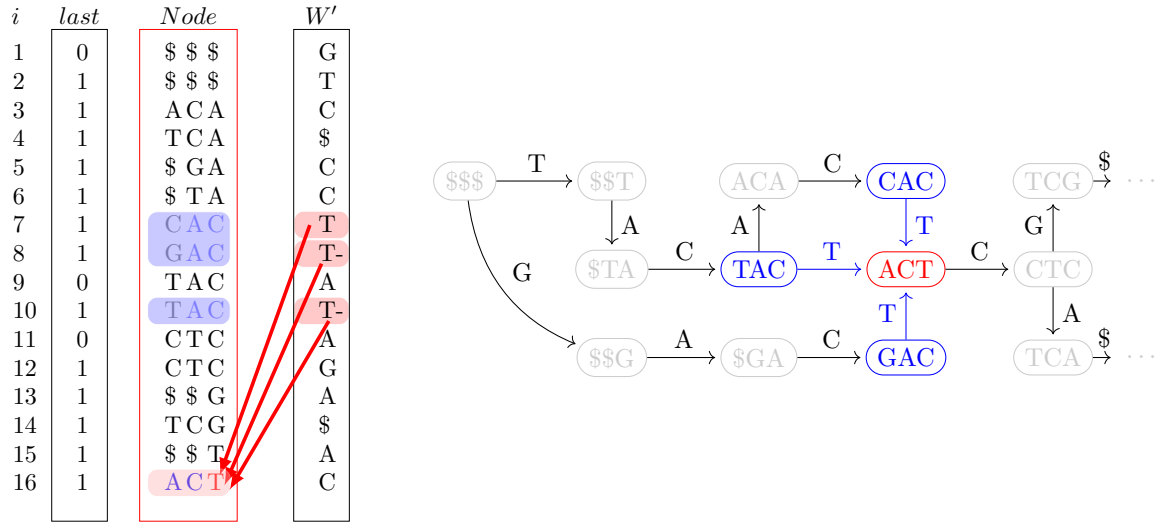


Figure 3.12: *Incoming* operation for every edge arriving at the *Node* with $\overleftarrow{v} = \mathtt{ACT}$. The red arrows indicate the mappings between W' and *Node*.

3.2.4 BOSS construction

Construction with radix sort. For a string collection $\mathcal{S}$, we can co-lexicographically sort its k-mers obtained from the distinct $(k+1)$-mers of $\mathcal{S}$ by applying the radix sort algorithm from right to left to obtain the BOSS representation. For example, Figure 3.13 shows this process for the collection $\mathcal{S} = \{\$\$\$\mathtt{TACACT}, \$\$\$\mathtt{TACTCA}, \$\$\$\mathtt{GACTCG}\}$.

The resulting co-lexicographic order of the k-mers represents the *Node* matrix. To obtain the string W we need one more iteration of radix sort in the last symbol of the corresponding $(k+1)$-mers. By extracting the last symbol of each $(k+1)$-mer we have the *Node* matrix and the string W. This process is shown in Figure 3.14.

We denote by $\mathrm{lcs}(S_1, S_2)$ the length of the longest common suffix of two strings S_1 and S_2 in Σ^*. For example $\mathrm{lcs}(\mathtt{GAC}, \mathtt{TAC}) = 2$ and $\mathrm{lcs}(\mathtt{TAC}, \mathtt{CTC}) = 1$. Let $[i, j]$ be an interval in *Node* where all k-mers share an lcs $\geq k$. We will call this interval a k-lcs-interval.

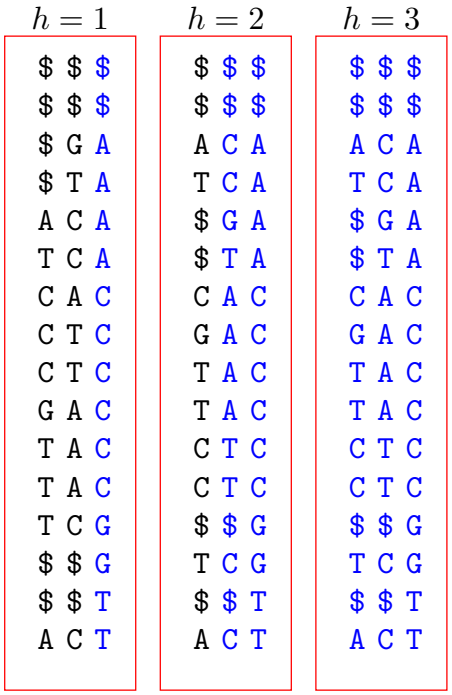


Figure 3.13: Radix co-lexographic sorting of the k-mers of $\mathcal{S}$.

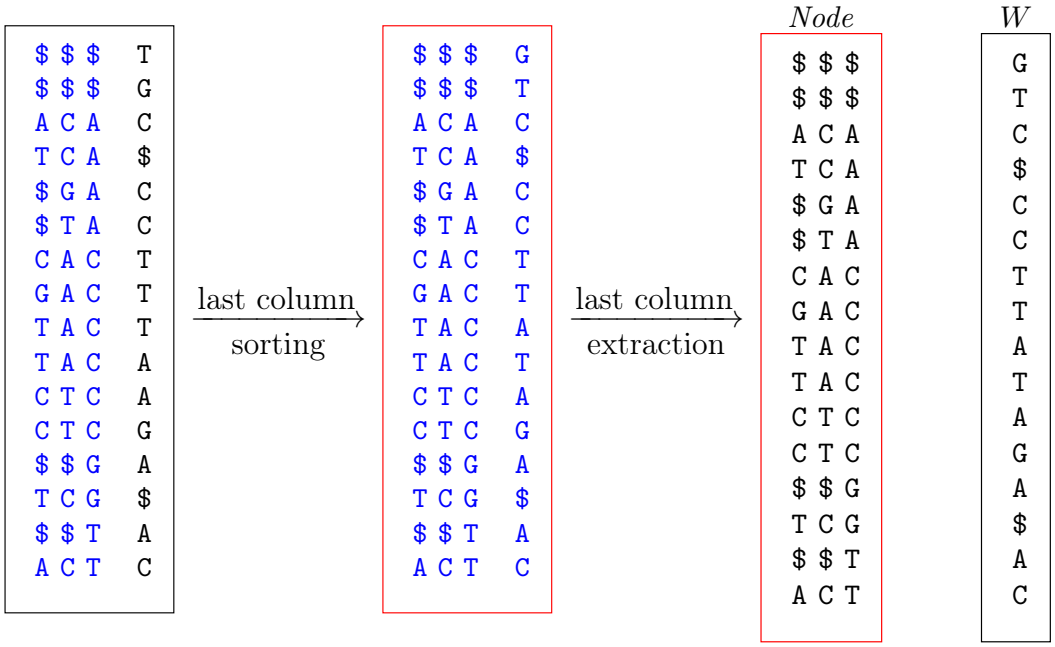


Figure 3.14: Radix sort on the last symbol of the $(k+1)$-mers followed by the extraction of the $(k+1)$-mer last column to obtain *Node* matrix and W string.

The BOSS construction using radix sort can be observed in Algorithm 2. The bitvectors *last* and W^- can be obtained by comparing the lcs between consecutive k-mers in *Node* as follows. For $i = 1, 2, \ldots, m-1$, whenever the value of $\text{lcs}(Node[i], Node[i+1])$ is equal to k, set $last[i]$ to 0 otherwise set $last[i]$ to 1 (Lines 10-14). When $i = m$, set $last[i]$ to 1. For computing $W^-[i]$, we have to check if $Node[i]$ shares $k-1$ symbols (in the reverse order) with a previous k-mer, say $Node[j]$ with $W[j] = W[i]$ $(j < i)$, that is,

if they are in the same $(k-1)$-lcs-interval. If they are, set $W^-[i]$ to 0 otherwise set $W^-[i]$ to 1. To compute $W^-[i]$ we can keep an auxiliary bitvector f of size σ where $f[c] = 1$ if the symbol c already occurred in the current $(k-1)$-mer interval, and $f[c] = 0$ otherwise (Lines 15-20). Whenever $\text{lcs}(Node[j], Node[i]) < k-1$ we will find a new lcs-interval in the next iteration. Therefore we can clean this bitvector, that is, set $f[c] = 0$ for all $c \in \sigma$.

To compute the C array, we count the frequency of each character c in W using the F array (Line 9). Then, we update the values in C (Lines 25-28), such that $C[c]$ stores the number of symbols smaller than c. An example of a complete BOSS construction for a collection $\mathcal{S}$ can be seen in Figure 3.15.

Algorithm 2: `boss_radix`

Input: A matrix of $kmers$ and its length m
Output: BOSS components $C, last, W, W^-$

```
1  int F[1..σ] = 0;
2  char W[1..m] = "\0";
3  bitvector last[1..m] = 1;
4  bitvector W⁻[1..m] = 0;
5  bitvector f[1..σ] = 0;
6  radix_sort(kmers);
7  for i = 1 to m do
8  |  W[i] = kmers[i][k];
9  |  F[W[i]] = F[W[i]] + 1;
10 |  if lcs(kmers[i], kmers[i+1]) == k then
11 |  |  last[i] = 0;
12 |  else
13 |  |  last[i] = 1;
14 |  end
15 |  if f[W[i]] == 0 then
16 |  |  W⁻[i] = 1;
17 |  |  f[W[i]] = 1;
18 |  else
19 |  |  W⁻[i] = 0;
20 |  end
21 |  if lcs(kmers[i], kmers[i+1]) < k-1 then
22 |  |  f.reset();
23 |  end
24 end
25 int C[1..σ] = 0;
26 for c = 2 to σ do
27 |  C[c] = C[c-1] + F[c-1];
28 end
29 return C, last, W, W⁻;
```

Construction with BWT and LCP array. Egidi *et al.* [10] proposed the eGap algorithm for computing the multi-string BWT and LCP array in external memory. As an application, they showed how to compute the BOSS representation by a sequential

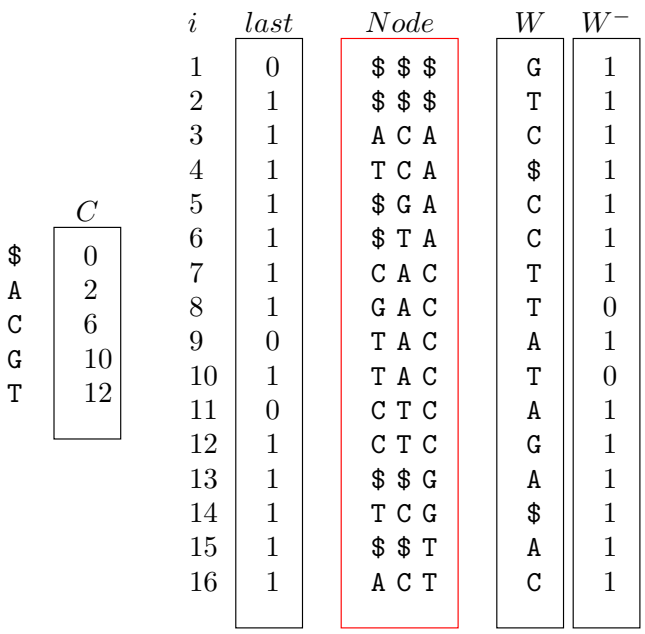


Figure 3.15: BOSS construction using radix for $\mathcal{S} = \{\$\$\$\texttt{TACACT}, \$\$\$\texttt{TACTCA}, \$\$\$\texttt{GACTCG}\}$ and $k = 3$.

scan over the BWT and the LCP array built for the collection $\mathcal{S}$ with all strings reversed, denoted by $\mathcal{S}^R$. For example, if $\mathcal{S} = \{\$\$\$\texttt{TACACT}, \$\$\$\texttt{TACTCA}, \$\$\$\texttt{GACTCG}\}$, then $\mathcal{S}^R = \{\texttt{TCACAT}\$\$\$, \texttt{ACTCAT}\$\$\$, \texttt{GCTCAG}\$\$\$\}$. We can obtain the order of the $(k+1)$-mers of $\mathcal{S}^R$ from the sorted suffixes. Note that the contexts truncated to size k will correspond to the k-mers represented in the *Node* matrix, as shown in Figure 3.16. The LCP values of the truncated contexts will correspond to the LCS values, since the strings are in the reversed form. Also, the lcp-intervals will correspond to the lcs-intervals. We define the truncated LCP array as the LCP array for the truncated contexts.

Let N be the length of the BWT. We define the auxiliary bitvector f of size σ where $f[c] = 1$ if the symbol c already occurred in the current $(k-1)$-lcp-interval, and $f[c] = 0$ otherwise. We also define an auxiliary bitvector b of size σ where $b[c] = 1$ if the symbol c already occurred in the current k-lcp-interval, and $b[c] = 0$ otherwise.

To obtain the string W and the bitvectors W^- and *last* we can scan the BWT and the LCP as follows.

For $i = 1, 2, \ldots, N$ and j starting with 1, if $b[\text{BWT}[i]] = 0$, then $W[j]$ is set to BWT$[i]$, $b[\text{BWT}[i]]$ is set to 1 and j is incremented. Assuming that all values from *last* were initialized with 1, note that while $\text{LCP}[i+1] \geq k$ we have the same k-mer in *Node*, therefore $last[j-1] = 0$. To compute W^- we can use the bitvector f. We check if $\text{LCP}[i+1] < k-1$, in this case we will have a new $(k-1)$-lcp-interval and we reset f to start keeping track of the next $(k-1)$-lcp-interval. To compute the array C, we count the occurrences of $c = W[j]$ in $F[c]$ for every $c \in \sigma$. At the end, we update the C values, such that $C[c]$ stores the number of symbols smaller than c.

This procedure is shown in Algorithm 3. First we add the edge obtained from BWT[1] to the BOSS representation (Line 10) and we increment j and i. Then we iterate through the first k-lcp-interval while $\text{LCP}[i+1] \geq k$ adding the edges to the representation if

i	BWT	LCP	context	truncated context
1	G	0	$\$_1$	$\$_1$
2	T	0	$\$_1$	$\$_1$
3	T	0	$\$_2$	$\$_2$
4	C	0	ACAT$\$_2$	ACA
5	$\$_2$	2	ACTCAT$\$_1$	ACT
6	C	0	AG$\$_1$	AG$\$_1$
7	C	1	AT$\$_1$	AT$\$_1$
8	C	2	AT$\$_2$	AT$\$_2$
9	T	0	CACAT$\$_2$	CAC
10	T	2	CAG$\$_1$	CAG
11	A	2	CAT$\$_2$	CAT
12	T	3	CAT$\$_1$	CAT
13	G	1	CTCAG$\$_1$	CTC
14	A	4	CTCAT$\$_1$	CTC
15	A	0	G$\$_1$	G$\$_1$
16	$\$_1$	1	GCTCAG$\$_1$	GCT
17	A	0	T$\$_1$	T$\$_1$
18	A	1	T$\$_2$	T$\$_2$
19	$\$_1$	1	TCACAT$\$_2$	TCA
20	C	3	TCAG$\$_1$	TCA
21	C	3	TCAT$\$_1$	TCA

Figure 3.16: eGap output for collection $\mathcal{S} = \{\$\$\$\texttt{TACACT}, \$\$\$\texttt{TACTCA}, \$\$\$\texttt{GACTCG}\}$, having BWT, LCP, contexts and truncated contexts respectively.

$b[\text{BWT}[i]] = 0$ (Lines 12-20). We can visualize the `add_edge` function in Algorithm 4. When this loop ends, we reset b because we will have a new k-lcp-interval in the next iteration (Line 19). We increment i and check if $\text{LCP}[i+1] < k$, that is, if we are entering a new $(k-1)$-lcp-interval. If this is true, we reset the f array (Lines 21-23).

For example, consider the $(k-1)$-lcp-interval that corresponds to `AC` in Figure 3.16 (considering the reverse order of the contexts), that is, $9 \leq i \leq 12$. Starting from $i = 9$, we add this edge to the BOSS construction, $f[\texttt{T}] = 1$ and $b[\texttt{T}] = 1$ (Figure 3.17a).

Since $\text{LCP}[i+1] = \text{LCP}[10] = 2 < k$ we reset b and exit the loop (Line 19-20) because the next k-lcp-interval is different. Now we check the condition in Line 21, since $\text{LCP}[i+1] = \text{LCP}[10] = 2 = k-1$, then we do not need to reset the f array. In the next iteration we add the edge $\text{BWT}[10] = \texttt{T}$ to the BOSS representation. We have $\text{BWT}[i] = \texttt{T}$ and $f[\texttt{T}] = 1$ and in this case we have $W^-[j] = 0$ (Figure 3.17b). Also, $b[\texttt{T}] = 1$.

For $i = 10$ we have $\text{LCP}[i+1] = \text{LCP}[11] = 2 < k$, again we reset b and, since we are in the same $(k-1)$-lcp-interval we just continue to the next iteration where we add the edge $\text{BWT}[11] = \texttt{A}$ to the BOSS representation and update $f[\texttt{A}] = 1$ and $b[\texttt{A}] = 1$ (Figure 3.17c).

For $i = 11$ we have $\text{LCP}[i+1] = \text{LCP}[12] = 3 = k$ and there are more edges from this k-mer, so we enter the loop of lines 12-20. First we add the edge $\text{BWT}[12] = \texttt{T}$, $f[\texttt{T}] = 1$ and $b[\texttt{T}] = 1$. Also we are in the same k-lcp-interval and we update the previous *last* value (Figure 3.17d).

Finally, for $i = 12$ we have $\text{LCP}[i+1] = \text{LCP}[13] = 1 < k$ and we leave the loop

Algorithm 3: `boss_construction`

Input: LCP and BWT arrays and selected k
Output: BOSS components $C, last, W, W^-$

```
1  int F[1..σ] = 0;
2  char W[1..m] = "\0";
3  bitvector last[1..m] = 0;
4  bitvector W⁻[1..m] = 0;
5  bitvector f[1..σ] = 0;
6  bitvector b[1..σ] = 0;
7  int j = 1;                                          // BOSS size counter
8  int i = 1;
9  while i ≤ N do
10 |  add_edge(F, last, W, W⁻, j, f, b, i);
11 |  j = j + 1;
12 |  while LCP[i + 1] ≥ k do
13 |  |  if b[BWT[i + 1]] == 0 then
14 |  |  |  add_edge(F, last, W, W⁻, j, f, b, i + 1);
15 |  |  |  last[j − 1] = 0;
16 |  |  |  j = j + 1;
17 |  |  end
18 |  |  i = i + 1;
19 |  |  b.reset();
20 |  end
21 |  if LCP[i + 1] < k − 1 then
22 |  |  f.reset();
23 |  end
24 |  i = i + 1;
25 end
26 int C[1..σ] = 0;
27 for c = 2 to σ do
28 |  C[c] = C[c − 1] + F[c − 1];
29 end
30 return C, last, W, W⁻;
```

(Line 20). Since LCP[13] = 1 < $k - 1$ we reset f. In the next iteration we will have a new $(k-1)$-lcp-interval. Iterating until $i = N$ the algorithm will produce the BOSS representation shown in Figure 3.18.

Recall that m is the total number of $(k+1)$-mers in a collection $\mathcal{S}$ and N is the sum of all string lengths. The BOSS construction is computed in $O(m \cdot (k+1))$ time using radix sort and $O(N)$ time using the BWT and LCP array. The working space required for radix sort is $m \cdot \log \sigma$ bits, whereas with the BWT and the LCP array we have $N \log \sigma$ bits for the BWT and $N \cdot \lceil \log(k+1) \rceil$ bits for the truncated LCP array. Therefore, the overall space required for the construction using the BWT and the LCP array is $N \log \sigma + N \cdot \lceil \log(k+1) \rceil$ bits.

Algorithm 4: `add_edge`

Input: $F, last, W, W^-, j, f, b$ and pos, in the context of Algorithm 3

```
1 W[j] = BWT[pos];
2 last[j] = 1;
3 if f[BWT[pos]] == 1 then
4 |   W^-[j] = 0;
5 else
6 |   f[BWT[pos]] = 1;
7 end
8 F[BWT[pos]] = F[BWT[pos]] + 1;
9 b[BWT[pos]] = 1;
```

3.3 Genome comparison with de Bruijn graphs

3.3.1 Lyman et al.

Lyman *et al.* [25] proposed an assembly-free phylogenetic tree reconstruction method using Colored de Bruijn Graphs (CdBG) [17]. Their algorithm identifies paths of vertices having at least c colors and then constructs a multiple sequence alignment of paths' sequences.

The authors define a *Colored de Bruijn Graph* as a de Bruijn graph in that each vertex is associated to a color (or a set of colors). These colors are used to represent distinct taxon (or species or collection of reads of a genome). Let a CdBG be defined as $\mathcal{G} = \{G_1, G_2, \ldots, G_i, \ldots, G_d\}$ for d collections where $G_i = (V_i, E_i)$ is a de Bruijn Graph for the $i-th$ taxon. They refer to each $G \in \mathcal{G}$ as a distinct color or taxon.

Let a path $P = (v_1, \ldots, v_w)$ in G_i be defined as a sequence of vertices in V_i such that all edges $(v_j, v_{j+1}) \in E_i$. They define a *bubble* as a set of paths $B = \{P_1, \ldots, P_z\}$ such that each $P_i \in B$ has the same initial and final vertices. For example, in Figure 3.19 we have a bubble $B = \{$ACTGTG, ACTAGGTG, ACTAGTG$\}$ in a CdBG with 3 colors and $k = 3$. The colors of the vertices represent the following sets:

- yellow: vertices exclusively in P_1;
- red: vertices exclusively in P_2;
- blue: vertices exclusively in P_3;
- purple: vertices in $P_2 \cap P_3$;
- grey: vertices in $P_1 \cap P_2 \cap P_3$.

The authors proposed the `kleuren` algorithm, which works by iterating over a superset of vertices K, defined as $K = \{V_1 \cup V_2 \cup \ldots \cup V_i \cup V_d\}$, discovering vertices that could form a bubble in the CdBG. A vertex v could form a bubble if v is present in c or more colors of $\mathcal{G}$, where c is given by the user. More bubbles may be found using lower values of c, however more vertices will be considered as a starting vertex of a bubble, which will make `kleuren` take a longer time to run.

i	$last$	*Node*	W	W^-
1	1	$\$_1$	T	1
2	1	$\$_1$	G	1
3	1	ACA	C	1
4	1	TCA	$\$_2$	1
5	1	$\$_1$GA	C	1
6	1	$\$_1$TA	C	1
7	1	CAC	T	1
⋮	⋮	⋮	⋮	⋮

(a) $i = 9$

i	$last$	*Node*	W	W^-
1	1	$\$_1$	T	1
2	1	$\$_1$	G	1
3	1	ACA	C	1
4	1	TCA	$\$_2$	1
5	1	$\$_1$GA	C	1
6	1	$\$_1$TA	C	1
7	1	CAC	T	1
8	1	GAC	T	0
⋮	⋮	⋮	⋮	⋮

(b) $i = 10$

i	$last$	*Node*	W	W^-
1	1	$\$_1$	T	1
2	1	$\$_1$	G	1
3	1	ACA	C	1
4	1	TCA	$\$_2$	1
5	1	$\$_1$GA	C	1
6	1	$\$_1$TA	C	1
7	1	CAC	T	1
8	1	GAC	T	0
9	1	TAC	A	1
⋮	⋮	⋮	⋮	⋮

(c) $i = 11$

i	$last$	*Node*	W	W^-
1	1	$\$_1$	T	1
2	1	$\$_1$	G	1
3	1	ACA	C	1
4	1	TCA	$\$_2$	1
5	1	$\$_1$GA	C	1
6	1	$\$_1$TA	C	1
7	1	CAC	T	1
8	1	GAC	T	0
9	0	TAC	A	1
10	1	TAC	T	0
⋮	⋮	⋮	⋮	⋮

(d) $i = 12$

Figure 3.17: BOSS construction using the BWT and LCP in $(k-1)$-lcp-interval equal to AC.

Within the bubbles in the CdBG identified by kleuren, the authors used MAFFT [19] to perform a multiple sequence alignment for each sequence in every bubble. Finally, each multiple sequence alignment was concatenated to form a supermatrix, which was used to infer the phylogenetic tree using the Maximum Likelihood method. As illustrated in Figure 3.20. The authors argue that kleuren is computationally cheaper than the traditional alignment-based phylogenetic methods and that many biological assumptions made in alignments may be avoided.

The authors used the kleuren algorithm to reconstruct the phylogeny of the 12 Drosophila species and compared it with the reference phylogeny, reporting consistent results.

3.3.2 Polevikov and Kolmogorov

Polevikov and Kolmogorov [31] assume that each genome has a single circular chromosome, implying that each node in a de Bruijn graph has at least one incoming and one outgoing edge. If a node has exactly one incoming and one outgoing edge it is called non-branching, otherwise it is called branching.

The authors define an *annotated de Bruijn graph* (ADBG) as a de Bruijn graph with

i	*last*	*Node*	W	W^-
1	1	$\$_1$	T	1
2	1	$\$_1$	G	1
3	1	ACA	C	1
4	1	TCA	$\$_2$	1
5	1	$\$_1$GA	C	1
6	1	$\$_1$TA	C	1
7	1	CAC	T	1
8	1	GAC	T	0
9	0	TAC	A	1
10	1	TAC	T	0
11	0	CTC	A	1
12	1	CTC	G	1
13	1	$\$_1$T	A	1
14	1	$\$_1$G	A	1
15	1	TCG	$\$_1$	1
16	0	ACT	$\$_1$	1
17	1	ACT	C	1

Figure 3.18: BOSS representations for collection $\mathcal{S} = \{\$\$\$\texttt{TACACT}, \$\$\$\texttt{TACTCA}, \$\$\$\texttt{GACTCG}\}$.

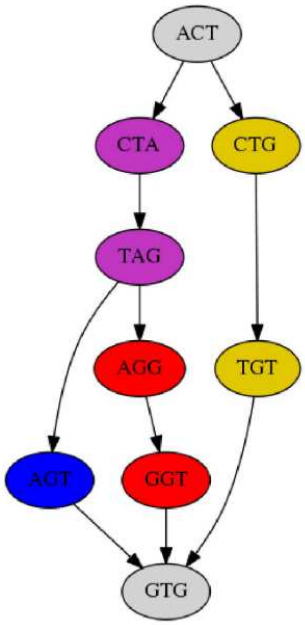


Figure 3.19: An example of a bubble in a Colored de Bruijn Graph, from [25].

each edge labeled as either unique or repetitive, based on whether its $(k-1)$-mer occurs once or more times in the genome.

Given two ADBGs AG_1 and AG_2 for genomes G_1 and G_2, the authors say that a pair of edges e_1 unique in AG_1 and e_2 unique in AG_2 are syntenic if they spell the same subsequence. Two unique edges u and v are compatible in an ADBG AG if either u and v are adjacent or AG contains a path between u and v where all intermediate edges are repetitive.

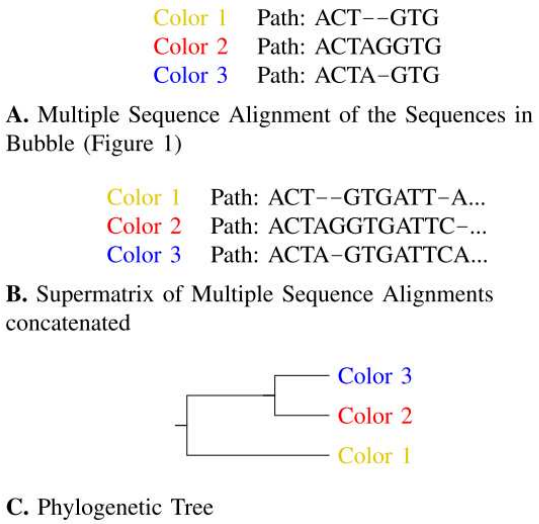


Figure 3.20: Multiple sequence alignment of bubbles and phylogenetic tree reconstruction example, from [25].

Given two ADBGs AG_1 and AG_2 with two pairs of syntenic edges $U = \{u_1, u_2\}$ and $V = \{v_1, v_2\}$, the authors say that U and V are colinear if u_1 and v_1 are compatible in AG_1, and u_2 and v_2 are compatible in AG_2. A synteny path is then defined as a sequence of pairs of syntenic edges $P = (E_1, E_2, \ldots, E_k)$, in which every two consecutive pairs (E_{i-1}, E_i) are colinear.

In order to compare two ADBGs AG_1 and AG_2 with unique edges decomposed into the set of syntenic edge pairs $E = \{(u_1, u_2), (v_1, v_2), (w_1, w_2), \ldots\}$, the authors propose the MSP decomposition problem, that is the problem of finding the minimum number of synteny paths that cover all edge pairs in E. Finding such synteny paths corresponds to finding a minimal set of sequences that include unique edges in AG_1 and AG_2 in the same order, with repeats occurring arbitrarily inbetween.

The authors show that the MSP decomposition is NP-hard by a reduction from the Hamiltonian path problem and present a heuristics for the problem. For the heuristics the authors define a double-stranded de Bruijn graph where for any edge $+e$ that spells a sequence S there exists a unique edge $-e$ that spells the reverse-complement of S.

A breakpoint graph is then constructed for two double-stranded ADBGs AG_1 and AG_2 with unique edges decomposed into $2n$ syntenic pairs $\{\pm 1, \pm 2, ..., \pm n\}$ having two vertices i^t and i^h for each syntenic pair $\pm i$. There is an undirected edge in the breakpoint graph between u^h and v^t for each pair of colinear syntenic edges (u, v) in AG_1 and AG_2.

The authors find a maximum matching in the breakpoint graph, add edges that connect nodes $\{i^h, i^t\}$, $i = 1, \ldots, n$, and finally reconstruct synteny paths by taking the resulting set of paths in the breakpoint graph as a set of synteny paths between AG_1 and AG_2.

The authors used the genome assembler Flye to reconstruct 15 different Drosophila species. They applied synteny paths to compare each assembly against a high-quality *Drosophila melanogaster* reference genome and computed synteny paths between all pairs of assemblies, reporting consistent results.

Chapter 4

gcBB Algorithm

4.1 Genome comparison via BWSD

In this work we propose the gcBB algorithm (Genome Comparison using BOSS and BWSD). The input of gcBB is a set of genomes and the value of k, the output is a pair of distance matrices, with the expectation and entropy distances among all pairs of genomes in the set. Given such distance matrices, one can reconstruct a phylogenetic tree for the set using the Neighbor-Joining [35] or any other algorithm [11].

The idea of gcBB is to construct a colored de Bruijn graph for a pair of genomes using the BOSS representation and, given the colored BOSS, compute the BWSD to measure the similarity between the genomes. The intuition is that the more the edges are intermixed in the colored BOSS, the greater the number of shared nodes and the more similar the genomes are.

4.2 Algorithm

Let $\mathcal{S}$ be a set of genomes and k be the selected size to construct the BOSS representation of the de Bruijn graphs. Our algorithm is divided into three phases summarized in Algorithm 5.

In the first phase we construct the BWT and the LCP array for each genome (Lines 1-3). We also compute an auxiliary array CL, which gives the context lengths. Then we merge these structures for each pair of genomes, obtaining the merged files and the document array DA (Lines 4-6). Both steps can be done using eGap [9].

In the second phase we construct the BOSS representation of the colored de Bruijn graph for each pair of genomes in $\mathcal{S}$. We obtain a bitvector called `colors`, of length m, which indicates from which genome each edge came (Line 11). The construction of the colored BOSS is based on the construction using the BWT and the LCP array, described in Section 3.2.4.

In the third phase we use the `colors` bitvector together with the CL array to compute the BWSD for this pair of genomes, populating the entropy and expectation matrix (Lines 12-16).

Algorithm 5: gcBB

Input: Set $\mathcal{S}$ and the selected k
Output: Matrices D_M and D_E of double precision numbers

```
   // Phase 1
1  for each genome S_i in S do
2  |  eGap(S_i);                                  // compute LCP, BWT and CL
3  end
4  for each pair of genomes {S_i, S_j} in S do
5  |  eGap(S_i, S_j);              // merge LCP, BWT and CL and generates DA
6  end
7  double D_M[1..d][1..d] = 0.0;
8  double D_E[1..d][1..d] = 0.0;
9  for each pair of genomes {S_i, S_j} in S do
   |  // Phase 2
10 |  bitvector colors initialized with 0 for each pair {S_i, S_j};
11 |  colors = colored_boss_construction(LCP_{i,j}, BWT_{i,j}, CL_{i,j}, DA_{i,j}, k);
   |  // Phase 3
12 |  double expectation = 0.0;
13 |  double entropy = 0.0;
14 |  {expectation, entropy} = bwsd_computation(colors, LCP_{i,j}, CL_{i,j}, k);
15 |  D_M[i][j] = expectation;
16 |  D_E[i][j] = entropy;
17 end
18 return D_M, D_E;
```

4.2.1 Phase 1

In the rest of the chapter, we will refer to the BWT as an array of characters. Initially, we construct the BWT, the LCP and the CL arrays for each genome in external memory using the eGap algorithm. We remark that one could use any other tool to construct these data structures, for example [3, 23, 32].

Consider the following set of genomes $\mathcal{S}_1 = \{\texttt{TACTCA}, \texttt{TACACT}\}$ and $\mathcal{S}_2 = \{\texttt{GACTCG}\}$. The eGap output for each one of these genomes is shown in Figures 4.1a and 4.1b, respectively. We remark that eGap does not produce the contexts, it is only shown for a better understanding of the examples.

Given the BWT, the LCP and the CL arrays for a pair of genomes, we merge these arrays while generating the document array DA, containing the genome of each position of the arrays. Note that DA can be stored in a bitvector, since we merge only pairs of genomes. The resulting arrays are stored in external memory. We also use eGap for the merging, but we could use any other available tool.

In Figure 4.2 we have the output of the merge of genomes $\mathcal{S}_1$ and $\mathcal{S}_2$ by eGap.

4.2.2 Phase 2

Given the merged BWT, LCP and CL arrays, together with the document array DA, we construct the BOSS representation of the colored de Bruijn graph with the algorithm

i	BWT	LCP	CL	context
1	T	0	1	$\$_1$
2	T	0	1	$\$_2$
3	C	0	5	ACAT$\$_2$
4	$\$_2$	2	7	ACTCAT$\$_1$
5	C	1	3	AT$\$_1$
6	C	2	3	AT$\$_2$
7	T	0	6	CACAT$\$_2$
8	A	2	4	CAT$\$_2$
9	T	3	4	CAT$\$_1$
10	A	1	6	CTCAT$\$_1$
11	A	0	2	T$\$_1$
12	A	1	2	T$\$_2$
13	$\$_1$	1	7	TCACAT$\$_2$
14	C	3	5	TCAT$\$_1$

(a)

i	BWT	LCP	CL	context
1	G	0	1	$\$_1$
2	C	0	3	AG$\$_1$
3	T	0	4	CAG$\$_1$
4	G	1	6	CTCAG$\$_1$
5	A	0	2	G$\$_1$
6	$\$_1$	1	7	GCTCAG$\$_1$
7	C	0	5	TCAG$\$_1$

(a)

Figure 4.1: The BWT, LCP and CL arrays output by eGap for genomes (a) $\mathcal{S}_1$, (b) $\mathcal{S}_2$.

i	BWT	LCP	CL	DA	context
1	T	0	1	0	$\$_1$
2	T	0	1	0	$\$_2$
3	G	0	1	1	$\$_3$
4	C	0	5	0	ACAT$\$_2$
5	$\$_3$	2	7	0	ACTCAT$\$_1$
6	C	1	3	1	AG$\$_3$
7	C	1	3	0	AT$\$_1$
8	C	2	3	0	AT$\$_2$
9	T	0	6	0	CACAT$\$_2$
10	T	2	4	1	CAG$\$_3$
11	A	2	4	0	CAT$\$_1$
12	T	3	4	0	CAT$\$_2$
13	G	1	6	1	CTCAG$\$_3$
14	A	4	6	0	CTCAT$\$_1$
15	A	0	2	1	G$\$_3$
16	$\$_2$	1	7	1	GCTCAG$\$_3$
17	A	0	2	0	T$\$_1$
18	A	1	2	0	T$\$_2$
19	$\$_1$	1	7	0	TCACAT$\$_2$
20	C	3	5	1	TCAG$\$_3$
21	C	3	5	0	TCAT$\$_1$

Figure 4.2: Merged BWT, LCP, CL arrays, the DA, and contexts for $\mathcal{S}_1, \mathcal{S}_2$. We remark that the context column is not produced by eGap.

presented in Section 3.2.4.

We compute two additional arrays, the `colors` bitvector and the `coverage` array. We also store a summarized LCP and CL arrays, that is, the LCP and CL arrays based on

the edges that entered our colored BOSS representation. The `colors` bitvector gives from which genome each edge came and can be easily obtained from DA. The `coverage` array gives the number of times a $(k+1)$-mer represented by an edge occurred in its genome. Since a de Bruijn graph has one vertex for each k-mer, the first time we have an edge of value c leaving a k-mer it will be added to our BOSS representation and its coverage will be set to 1. Every time we have another occurrence of an edge c leaving that same k-mer, it will not be added to the representation but we increment the coverage of that edge in our graph representation. The `coverage` array can be obtained with two auxiliary arrays d_i and d_j. In the first occurrence at a symbol c from genome i in the k-lcp-interval, we set its position p in $d_i[c]$ and set `coverage`$[p]$ to 1. If there is another occurrence of symbol c in genome i we simply increment `coverage`$[p]$. This idea is analogous for symbols from genome j. When we leave the k-lcp-interval we reset these arrays setting all values to -1. These arrays will be used in Phase 3.

The modifications to obtain the `colors` and `coverage` arrays can be visualized in Algorithms 6 and 7. For a clearer presentation of the algorithms, consider we have the structure `boss` containing $C, last, W$ and W^-, and the structure `c_boss` containing `colors`, `coverage`, and the summarized LCP and CL.

The main difference between Algorithm 6 and the BOSS construction with BWT and LCP in Algorithm 3 can be visualized in Lines 10-22, where we have to check the d arrays to decide whether to include the edge or increase the coverage. In Algorithm 7 we just have new assignments for the new arrays, compared to Algorithm 4.

Consider the merged arrays of genomes $\mathcal{S}_1$ and $\mathcal{S}_2$ obtained in Phase 1 and consider $k = 3$. The colored BOSS representation obtained for the merge of $\mathcal{S}_1$ and $\mathcal{S}_2$ merge can be observed in Figure 4.3.

last	*Node*	W	W^-	`color`	`coverage`	LCP	CL	context
1	$\$_1$	T	1	0	2	0	1	$\$_1$
1	$\$_3$	G	1	1	1	0	1	$\$_3$
1	ACA	C	1	0	1	0	5	ACAT$\$_2$
1	TCA	$\$_3$	1	0	1	2	7	ACTCAT$\$_1$
1	$\$_3$GA	C	1	1	1	1	3	AG$\$_3$
1	$\$_1$TA	C	1	0	2	1	3	AT$\$_1$
1	CAC	T	1	0	1	0	6	CACAT$\$_2$
1	GAC	T	0	1	1	2	4	CAG$\$_3$
0	TAC	A	1	0	1	2	4	CAT$\$_1$
1	TAC	T	0	0	1	3	4	CAT$\$_2$
0	CTC	A	1	0	1	1	6	CTCAG$\$_3$
1	CTC	G	1	1	1	4	6	CTCAT$\$_1$
1	$\$_3$G	A	1	1	1	0	2	G$\$_3$
1	TCG	$\$_2$	1	1	1	1	7	GCTCAG$\$_3$
1	$\$_1$T	A	1	0	1	0	2	T$\$_1$
0	ACT	$\$_1$	1	0	1	1	7	TCACAT$\$_2$
0	ACT	C	1	0	1	3	5	TCAG$\$_3$
1	ACT	C	0	1	1	3	5	TCAT$\$_1$

Figure 4.3: $\mathcal{S}_1\mathcal{S}_2$ merge colored BOSS representation with $k = 3$.

Algorithm 6: `colored_boss_construction`

Input: LCP, BWT, DA and CL arrays and the selected k
Output: Boss components in `boss` and colored boss in `c_boss`

bitvector $f[1..\sigma] = 0$;
bitvector $b[1..\sigma] = 0$;
integer array, both initialized with 0;
int $g = 0$; `// genome identification`
int $d_0[1..\sigma] = 0$;
int $d_1[1..\sigma] = 0$;
int $d\_pos = -1$;
Let `boss` be an structure containing the BOSS components $(last, W, W^-, C)$;
Let `c_boss` be an structure containing the `colors` bitvector and `coverage`;
int $j = 1$; `// BOSS size counter`
int $i = 1$;
while $i \leq N$ **do**
    $g = \text{DA}[i]$;
    $add\_edge(\text{BWT}, \text{LCP}, \text{CL}, \texttt{boss}, \texttt{c\_boss}, f, b, d, g, j, i)$;
    $j = j + 1$;
    while $LCP[i+1] \geq k$ **do**
        $g = \text{DA}[i+1]$;
        $d\_pos = d_g[\text{BWT}[i+1]]$; `// position of BWT`$[i+1]$ `in` d_g `if exists`
        if $b[BWT[i+1]] == 0$ *and* $d\_pos \neq -1$ **then**
            $add\_edge(\text{BWT}, \text{LCP}, \text{CL}, \texttt{boss}, \texttt{c\_boss}, f, b, d, g, j, i+1)$;
            $last[j-1] = 0$;
            $j = j + 1$;
        else
            `coverage`$[d\_pos] =$ `coverage`$[d\_pos] + 1$;
        end
        $i = i + 1$;
        d.`reset`();
        b.`reset`();
    end
    if $LCP[i+1] < k - 1$ **then**
        f.`reset`();
    end
    $i = i + 1$;
end
`boss`$.C[1] = 0$;
for $c = 2$ **to** σ **do**
    `boss`$.C[c] =$ `boss`$.C[c-1] + F[c-1]$;
end
return `boss, c_boss`;

4.2.3 Phase 3

The distance between each pair of genomes can be computed by applying the BWSD to the `colors` bitvector, obtaining the expectation and entropy distance matrices.

Algorithm 7: `add_colored_edge`

Input: BWT, LCP, CL, `boss`, `c_boss`, f, b, d, g, j and pos, in the context of Algorithm 6

```
1  boss.W[j] = BWT[pos];
2  boss.last[j] = 1;
3  if f[BWT[pos]] == 1 then
4  |  boss.W⁻[j] = 0;
5  else
6  |  f[BWT[pos]] = 1;
7  end
8  boss.F[BWT[pos]] = F[BWT[pos]] + 1;
9  b[BWT[pos] = 1;
10 c_boss.colors[j] = g;
11 d_g[BWT[pos]] = j;
12 c_boss.LCP = LCP[pos];
13 c_boss.CL = CL[pos];
14 c_boss.coverage[pos] = 1;
```

Note that the colored BOSS representation contains the edges of every i-mer from the merged genomes, where $1 \leq i \leq k+1$. The edges representing i-mers where $i < k+1$ are part of the BOSS representation and are needed in BOSS navigation. Nevertheless, we considered that these i-mers where $i < k+1$ can compromise our comparisons, since if we compare a pair of genomes using a value of $(k+1)$ and then compare this same pair using a value $(k'+1) > (k+1)$, all i-mers where $1 \leq i \leq k+1$ will be considered in both representations.

Since we are just interested in the $(k+1)$-mers for the comparisons, we filtered out all the edges of the colored BOSS representation using the CL array during the BWSD computation. That is, for every edge in the representation, we only considered edges where $\text{CL}[i] \geq k+1$ during the BWSD computation. For example, Figure 4.4 shows the colored BOSS representation of $\{\mathcal{S}_1, \mathcal{S}_2\}$ which will be considered in the BWSD computation, that is, excluding the edges where $\text{CL}[i] < k+1$.

With the filtered colored BOSS we can consider the `colors` bitvector as the α bitvector from the BWSD, thus we have

$$\alpha = \{0,0,0,1,0,0,0,1,1,0,0,1\}$$
$$r = 0^3 1^1 0^3 1^2 0^2 1^1$$

and $t_1 = 2$, $t_2 = 2$, $t_3 = 2$ and $s = 6$. The BWSD$(\mathcal{S}_1, \mathcal{S}_2)$ is

$$P\{k_j = 1\} = 2/6$$
$$P\{k_j = 2\} = 2/6$$
$$P\{k_j = 3\} = 2/6$$

last	*Node*	W	W^-	color	coverage	LCP	CL	context
1	$\$_1$	T	1	0	1	0	1	$\$_1$
1	$\$_3$	G	1	1	1	0	1	$\$_3$
1	ACA	C	1	0	1	0	5	ACAT$\$_2$
1	TCA	$\$_3$	1	0	1	2	7	ACTCAT$\$_1$
1	$\$_3$GA	C	1	1	1	1	3	AG$\$_3$
1	$\$_1$TA	C	1	0	1	1	3	AT$\$_1$
1	CAC	T	1	0	1	0	6	CACAT$\$_2$
1	GAC	T	0	1	1	2	4	CAG$\$_3$
0	TAC	A	1	0	1	2	4	CAT$\$_1$
1	TAC	T	0	0	1	3	4	CAT$\$_2$
0	CTC	A	1	0	1	1	6	CTCAG$\$_3$
1	CTC	G	1	1	1	4	6	CTCAT$\$_1$
1	$\$_3$G	A	1	1	1	0	2	G$\$_3$
1	TCG	$\$_2$	1	1	1	1	7	GCTCAG$\$_3$
1	$\$_1$T	A	1	0	1	0	2	T$\$_1$
0	ACT	$\$_1$	1	0	1	1	7	TCACAT$\$_2$
0	ACT	C	1	0	1	3	5	TCAG$\$_3$
1	ACT	C	0	1	1	3	5	TCAT$\$_1$

filtering CL $< k+1$

↓

last	*Node*	W	W^-	colors	coverage	LCP	CL	context
1	ACA	C	1	0	1	0	5	ACAT$\$_2$
1	TCA	$\$_3$	1	0	1	2	7	ACTCAT$\$_1$
1	CAC	T	1	0	1	0	6	CACAT$\$_2$
1	GAC	T	0	1	1	2	4	CAG$\$_3$
0	TAC	A	1	0	1	2	4	CAT$\$_1$
1	TAC	T	0	0	1	3	4	CAT$\$_2$
0	CTC	A	1	0	1	1	6	CTCAG$\$_3$
1	CTC	G	1	1	1	4	6	CTCAT$\$_1$
1	TCG	$\$_2$	1	1	1	0	7	GCTCAG$\$_3$
0	ACT	$\$_1$	1	0	1	0	7	TCACAT$\$_2$
0	ACT	C	1	0	1	3	5	TCAG$\$_3$
1	ACT	C	0	1	1	3	5	TCAT$\$_1$

Figure 4.4: Colored BOSS representation with $k = 3$ having only edges with $\text{CL}[i] \geq k+1$ for $\{\mathcal{S}_1, \mathcal{S}_2\}$.

Computing the distances $D_M(\mathcal{S}_1, \mathcal{S}_2)$ and $D_E(\mathcal{S}_1, \mathcal{S}_2)$ we have

$$\begin{aligned} D_M(\mathcal{S}_1, \mathcal{S}_2) &= (1(2/6) + 2(2/6) + 3(2/6)) - 1 \\ &= 2 - 1 \\ &= 1 \end{aligned}$$

and,

$$\begin{aligned} D_E(\mathcal{S}_1, \mathcal{S}_2) &= -(((2/6)\log_2(2/6)) + ((2/6)\log_2(2/6)) + ((2/6)\log_2(2/6))) \\ &= -(-1.58496) \\ &= 1.58496 \end{aligned}$$

As shown in the example, to compute the expectation and entropy we need r and t, and the value of s. The process we used to obtain these values in gcBB is shown in Algorithm 8. The expectation and entropy computations are shown in Algorithms 9 and 10, respectively.

Algorithm 8: `bwsd_computation`

Input: Bitvector `colors`, LCP and CL arrays and the selected k
Output: Computed `expectation, entropy` values

```
1  int color = 0;                                   // current color in r
2  int max = 0;                                     // max frequency in r
3  int j = 0;                                               // size of r
4  int r[1..j] = 0;                                  // run-length array
5  int i = 0;
6  for i = 1 to |colors| do
7  |  if CL[i] ≥ k + 1 then
8  |  |  if colors[i] == color then
9  |  |  |  r[j] = r[j] + 1;
10 |  |  else
11 |  |  |  if r[j] > max then
12 |  |  |  |  max = r[j];
13 |  |  |  end
14 |  |  |  color = colors[i];
15 |  |  |  j = j + 1;
16 |  |  |  r[j] = 1;
17 |  |  end
18 |  end
19 end
20 int s = 0;
21 int t[1..j] = 0;
22 for i = 1 to j do
23 |  s = s + r[i];
24 |  t[r[i]] = t[r[i]] + 1;
25 end
26 double expectation = bwsd_expectation(t, s, max);
27 double entropy = bwsd_entropy(t, s, max);
28 return entropy, expectation;
```

In our example, every value in the `coverage` array is equal to 1, that is, all $(k+1)$-mers occurred once in each genome. However, in real genomes the $(k+1)$-mers may occur

Algorithm 9: `bwsd_expectation`

Input: t, s, max
Output: `expectation`

```
1 int i = 0;
2 double expectation = 0.0;
3 for i = 1 to max + 1 do
4 |   if t[i] ≠ 0 then
5 |   |   double frac = t[i]/s;
6 |   |   expectation = expectation + (i · frac);
7 |   end
8 end
9 return expectation − 1.0;
```

Algorithm 10: `bwsd_entropy`

Input: t, s, max
Output: `entropy`

```
1 int i;
2 double entropy = 0.0;
3 for i = 1 to max + 1 do
4 |   if t[i] ≠ 0 then
5 |   |   double frac = t[i]/s;
6 |   |   entropy = entropy + (frac ∗ log(frac));
7 |   end
8 end
9 if entropy then
10 |   return entropy · (−1.0);
11 end
12 return 0.0;
```

multiple times and we can use this information in the BWSD to weight the edges of the graph, aiming at improving the accuracy of the results.

In order to illustrate how coverage information can change the measured similarity, we added reads to both genomes representing a $(k+1)$-mer that occurred in the last example. For example, let ${\mathcal{S}_1}^c = \{\texttt{TACTCA}, \texttt{TACACT}, \texttt{ACTC}, \texttt{ACTC}, \texttt{ACTC}\}$ and ${\mathcal{S}_2}^c = \{\texttt{GACTCG}, \texttt{ACTC}, \texttt{ACTC}\}$. In both genomes we have to increment the coverage information of the k-mers `ACT` with the outgoing edge `C`. The filtered edges for $\{\mathcal{S}_1^c, \mathcal{S}_2^c\}$ can be observed in Figure 4.5.

last	*Node*	W	W^-	`colors`	`coverage`	LCP	CL	context
1	`ACA`	`C`	1	0	1	0	5	`ACAT`$\$_2$
1	`TCA`	$\$_3$	1	0	1	2	7	`ACTCAT`$\$_1$
1	`CAC`	`T`	1	0	1	0	6	`CACAT`$\$_2$
1	`GAC`	`T`	0	1	1	2	4	`CAG`$\$_3$
0	`TAC`	`A`	1	0	1	2	4	`CAT`$\$_1$
1	`TAC`	`T`	0	0	1	3	4	`CAT`$\$_2$
0	`CTC`	`A`	1	0	1	1	6	`CTCAG`$\$_3$
1	`CTC`	`G`	1	1	1	4	6	`CTCAT`$\$_1$
1	`TCG`	$\$_2$	1	1	1	0	7	`GCTCAG`$\$_3$
0	`ACT`	$\$_1$	1	0	1	0	7	`TCACAT`$\$_2$
0	`ACT`	`C`	1	0	4	3	5	`TCAG`$\$_3$
1	`ACT`	`C`	0	1	3	3	5	`TCAT`$\$_1$

Figure 4.5: Colored BOSS representation with $k = 3$ having only edges with $\text{CL}[i] \geq k+1$ for $\{\mathcal{S}_1^c, \mathcal{S}_2^c\}$. Note that the $(k+1)$-mer `ACTC` increased in the last two rows because this $(k+1)$-mer was found 3 times in the first genome and 2 times in the second genome.

We remark that the BWSD values increase when we have more intermixed values in the `colors` bitvector, that is, more subsequent zeros and ones. If there are two equal $(k+1)$-mers from distinct genomes in the colored BOSS, they will always appear in the α array as

$$\{\ldots, 0, 1, \ldots\}$$

independently of the number of times these $(k+1)$-mers occurred in both genomes.

However, if these values occurred many times in both genomes, this should increase their similarity. To include this information we can apply the coverage information during the run length computation.

For example, take the α bitvector from the previous example with coverage as

$$\alpha^c = \{0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1\}$$

The last 0 and 1 values from α^c represent the $(k+1)$-mer `ACTC` from both genomes. In the colored BOSS this can be detected using the LCP array and the `colors` bitvector as follows.

Whenever $\text{LCP}[j] \geq k$, $\text{LCP}[j+1] \geq k$ and $\texttt{colors}[j] \neq \texttt{colors}[j+1]$ we have the same $(k+1)$-mer from distinct genomes. When this happens we apply the coverage value to the positions of the r^c where these values occurred. That is

$$\begin{aligned}
\alpha^c &= \{0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1\} \\
r^c &= 0^3 1^1 0^3 1^2 0^1 0^{1+3} 1^{1+2} \\
r^c &= 0^3 1^1 0^3 1^2 0^1 0^4 1^3
\end{aligned}$$

Finally, we expand r^c in the positions of the equal $(k+1)$-mers while merging them, that

is

$$r^c = 0^3 1^1 0^3 1^2 0^1 0^4 1^3$$
$$r^c = 0^3 1^1 0^3 1^2 0^1 0^1 1^1 0^1 1^1 0^1 1^1 0^1 1^0$$

Continuing the BWSD computation we have that $t_1 = 9$, $t_2 = 1$, $t_3 = 2$ and $s = 12$. Thus, the BWSD$(\mathcal{S}_1^c, \mathcal{S}_2^c)$ is

$$P\{k_j = 1\} = 9/12$$
$$P\{k_j = 2\} = 1/12$$
$$P\{k_j = 3\} = 2/12$$

Computing the distances $D_M(\mathcal{S}_1^c, \mathcal{S}_2^c)$ and $D_E(\mathcal{S}_1^c, \mathcal{S}_2^c)$ we have

$$\begin{aligned} D_M(\mathcal{S}_1^c, \mathcal{S}_2^c) &= (1(9/12) + 2(1/12) + 3(2/12)) - 1 \\ &= (17/12) - 1 \\ &= 5/12 \\ &= 0.41666 \end{aligned}$$

and,

$$\begin{aligned} D_E(\mathcal{S}_1^c, \mathcal{S}_2^c) &= -(((9/12)\log_2(9/12)) + ((1/12)\log_2(1/12)) + ((2/12)\log_2(2/12))) \\ &= -(-1.04085) \\ &= 1.04085 \end{aligned}$$

We can see that the use of coverage information can modify the expectation and entropy computed from the BWSD. This difference will be explored with experiments in the next section.

The modifications in Algorithm 8 to apply coverage information can be visualized in Algorithm 11. The function intermix is shown in Algorithm 12. The main difference can be observed in Lines 7-25, where we check if we have the same k-mer from distinct genomes. Entering this condition, we expand r as shown in last example. The computations of t, s, expectation and entropy remain the same.

4.3 Time and space analysis

Let n_1 and n_2 be the sizes of two genomes.

Phase 1: To construct and merge the BWT, LCP and CL we used the eGap algorithm, which runs in $O((n_1 + n_2) \cdot \mathsf{maxlcp})$ time, where maxlcp is the largest LCP value.

Phase 2: In the BOSS construction we have a sequential scan over LCP, BWT and DA, which takes $O(n_1 + n_2)$ time. Let m be the number of edges in the BOSS structure. The space required for the BOSS representation is $5m + 6\log m$ bits, as shown in Section 3.2.2. The `colors` bitvector and the `coverage` array require extra m bits and $4m$

Algorithm 11: `bwsd_coverage_computation`

Input: Bitvector `colors`, LCP and CL arrays and the selected k
Output: Computed `expectation, entropy` values

```
1  int color = 0;                                   // current color in r
2  int max = 0;                                     // max frequency in r
3  int j = 0;                                               // size of r
4  int r[1..j] = 0;                                   // run-length array
5  int i = 0;
6  for i = 1 to |colors| do
7      if CL[i] ≥ k + 1 then
8          if LCP[i] ≥ k and LCP[i + 1] ≥ k and colors[i] != colors[i + 1] then
9              j = intermix_r(coverage[i], coverage[i + 1], r, j);
10             if r[j] > max then
11                 max = r[j];
12             end
13         else
14             if colors[i] == color then
15                 r[j] = r[j] + 1;
16             else
17                 if r[j] > max then
18                     max = r[j];
19                 end
20                 color = colors[i];
21                 j = j + 1;
22                 r[j] = 1;
23             end
24         end
25     end
26 end
27 int s = 0;
28 int t[1..j] = 0;
29 for i = 1 to j do
30     s = s + r[i];
31     t[r[i]] = t[r[i]] + 1;
32 end
33 double expectation = bwsd_expectation(t, s, max);
34 double entropy = bwsd_entropy(t, s, max);
35 return entropy, expectation;
```

bytes respectively. For short reads, both LCP and CL can be stored in arrays of short integers, that is, $2m$ bytes for each one. Therefore, the overall space required is $6m$ bytes plus $6m + 6 \log m$ bits.

Phase 3: In the BWSD computation we have a sequential scan over `colors` bitvector and CL array, this takes $O(m)$ time. The arrays r and t require $O(m)$ bytes.

Conclusion: The running time for a pair of genomes is $O((n_1 + n_2) \cdot \mathsf{maxlcp}) + O(n_1 + n_2) + O(m)$ and the required space is $O(m)$.

Algorithm 12: `intermix_r`

Input: c_1, c_2, r and pos, in the context of Algorithm 11

```
1  int repetitions = 0;
2  while repetitions ≤ c_2 do
3  |   pos = pos + 1;
4  |   r[pos] = 1;
5  |   repetitions = repetitions + 1;
6  end
7  remaining = c_1 − repetitions + 1;
8  pos = pos + 1;
9  r[pos] = remaining;
10 return pos;
```

Chapter 5

Experiments

The gcBB algorithm was evaluated by reconstructing phylogenies of genomes from their sets of reads. Our algorithm was implemented in C and compiled with `gcc` version 4.9.2. The source code can be accessed at `https://github.com/lucaspr98/gcBB`. We used the eGap algorithm [10] to construct the data structures in external memory during Phase 1. From the output of our algorithm, we used the Neighbor-Joining algorithm to reconstruct the phylogenetic trees.

The experiments were conducted on a machine with `Debian GNU/Linux` 4.9.2 64 bits operating system with an `Intel Xeon E5-2630 v3` 20M Cache 2.40 GHz processor, 378 GB of RAM and a 13 TB SATA storage. Our experiments were limited to 48 GB of RAM.

We evaluated the performance of our algorithm on a dataset of Drosophila species and on a dataset of Vibrios. We compared the phylogenies obtained with gcBB with reference phylogenies.

5.1 Drosophilas

The Drosophila genus is a unique group containing a wide range of species that occupy diverse ecosystems. In addition to the most widely studied species, *Drosophila melanogaster*, this genus also has many other species that were fully or partially sequenced [29].

A gene family evolution across 12 fully sequenced Drosophila species was studied in [16]. This study generated the phylogeny that can be observed in Figure 5.1. This phylogeny will be used as a reference phylogeny for comparisons with the results of gcBB.

The work of Lyman *et al.* [25] used the same set of Drosophilas of the reference phylogeny. They generated a phylogeny of the 12 Drosophilas that can be observed in Figure 5.2. They used the Robinson-Foulds distance [34] between their tree and the established tree in [16] and the resulting distance was 0 using $k = 17$, which means that the phylogenies are consistent.

Polevikov and Kolmogorov *et al.* [31] used a slightly distinct set of Drosophilas on their experiments. Miller *et al* [29] used the single-molecule sequencing technology by Oxford Nanopore to reconstruct the phylogeny of 15 Drosophila species shown in Figure 5.3. They used 10 of the 12 originally sequenced Drosophila species (*D. ananassae, D. erecta, D. mojavensis, D. persimilis, D. pseudoobscura, D. sechellia, D. simulans, D. virilis, D.*

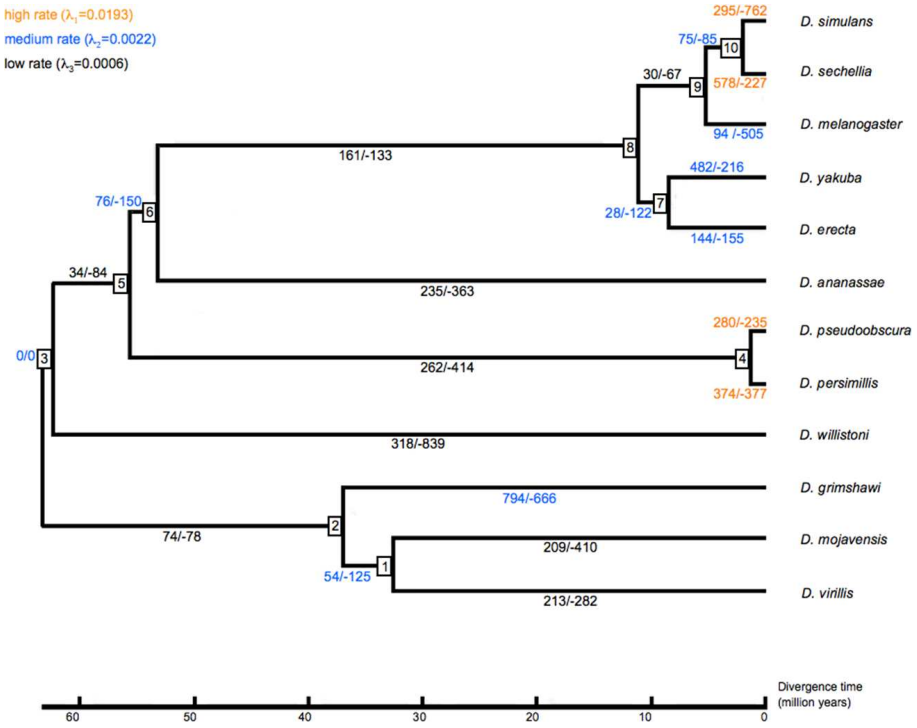


Figure 5.1: The phylogeny of 12 Drosophilas. Figure from [16].

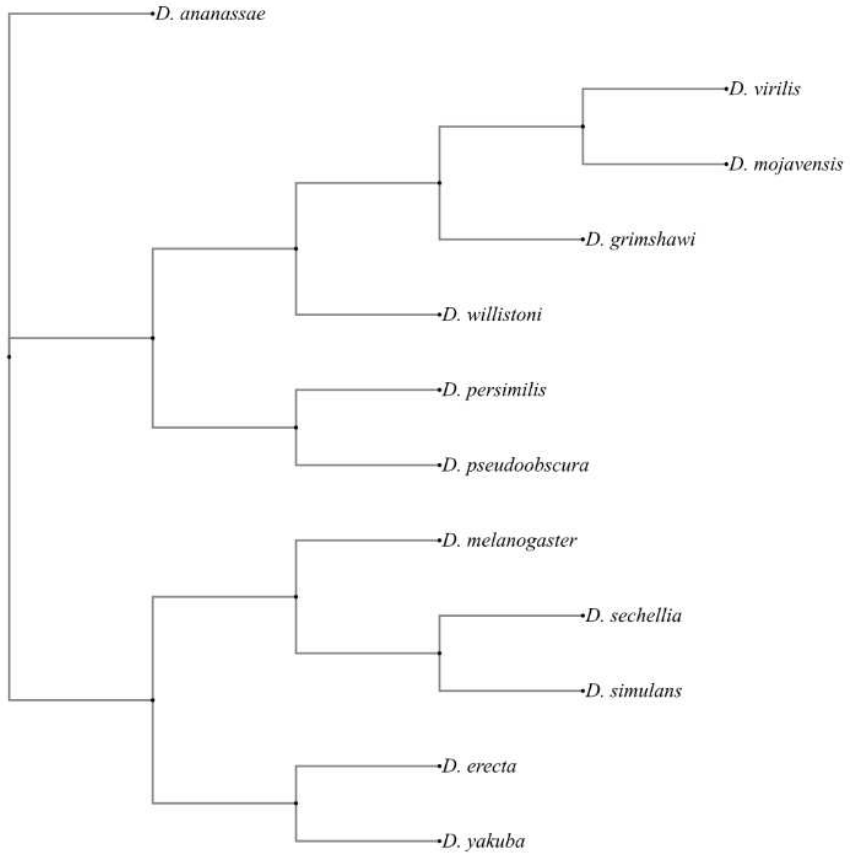


Figure 5.2: The phylogeny of 12 Drosophilas using $k = 17$. Figure from [25].

willistoni, and D. yakuba), four additional species that had previously reported assemblies (*D. biarmipes, D. bipectinata, D. eugracilis, and D. mauritiana*), and one novel assembly (*D. triauraria*). The work of Polevikov and Kolmogorov generated the phylogeny of such 15 Drosophilas that can be observed in Figure 5.4 using $k = 15$.

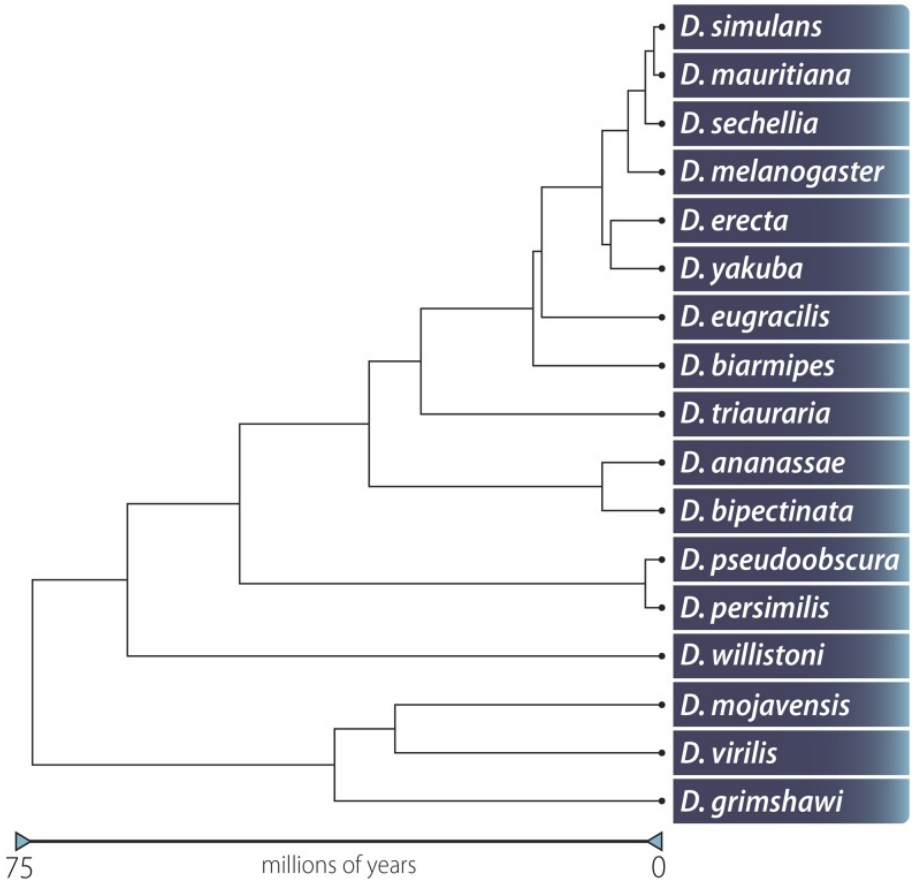


Figure 5.3: The phylogeny of 15 Drosophilas. Figure from [29].

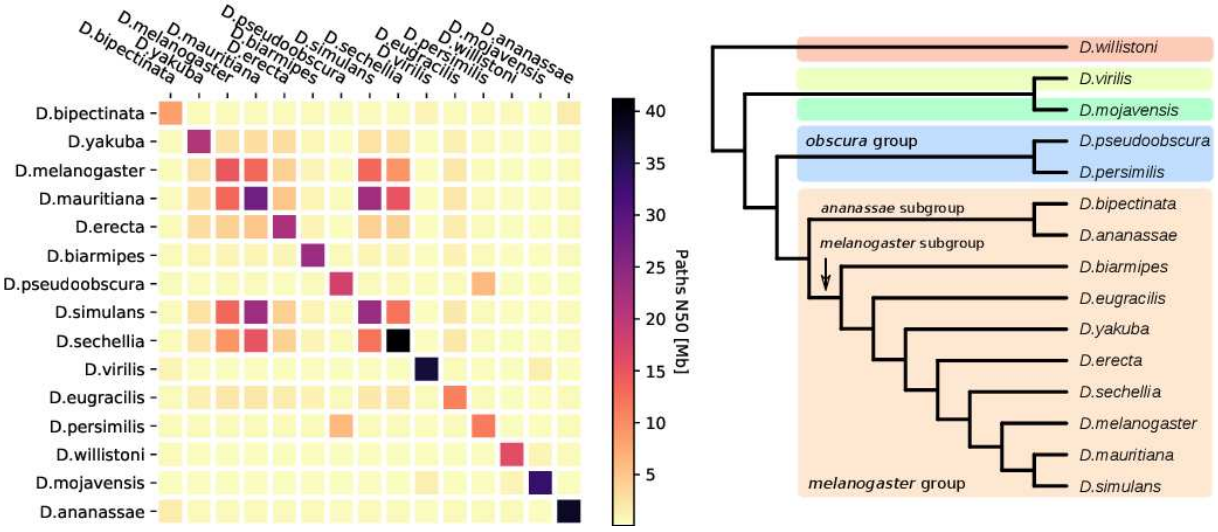


Figure 5.4: The phylogeny of 15 Drosophilas using $k = 15$. Figure [31].

5.1.1 Dataset

In this work we reconstructed the phylogeny of the 12 Drosophila species in Table 5.1 obtained from FlyBase [40], which is the main repository and WEB portal for genetic data related to *Drosophila melanogaster*, the fruit fly. The average length of the reads is 302 characters for these genomes, except for *D. grimshawi*, which has average read length of 6520 characters. The reads of *D. grimshawi* were sequenced with `MinION`[1], while the other genomes were sequenced with `NextSeq 500`[2].

Table 5.1: Information on the genomes of Drosophilas. The bases column specifies the number of sequenced bases of the genome (in Gbp). The reference column specifies the size of the complete referenced genome (in Mb). All genomes can be easily accessed through its Run accession number or BioSample in `https://www.ncbi.nlm.nih.gov/genbank/`.

Organism	Run	BioSample	Bases	Reference
D. melanogaster	SRR6702604	SAMN08511563	6.20	138.93
D. ananassae	SRR6425991	SAMN08272423	7.13	215.47
D. simulans	SRR6425999	SAMN08272428	9.22	131.66
D. virilis	SRR6426000	SAMN08272429	11.16	189.44
D. willistoni	SRR6426003	SAMN08272432	11.66	246.98
D. pseudoobscura	SRR6426001	SAMN08272435	12.28	163.29
D. mojavensis	SRR6425997	SAMN08272426	12.45	163.17
D. yakuba	SRR6426004	SAMN08272438	12.78	147.90
D. persimilis	SRR6425998	SAMN08272433	13.32	195.51
D. erecta	SRR6425990	SAMN08272424	14.01	146.54
D. sechellia	SRR6426002	SAMN08272427	14.44	154.19
D. grimshawi	SRR13070661	SAMN16729613	14.50	191.38

5.1.2 Running time

First, during Phase 1, we constructed the BWT, LCP and CL arrays for each genome in the dataset using eGap algorithm with the RAM usage limited to 32GB. The running time for each genome and the sizes of the arrays are shown in Table 5.2. The longest running time was approximately 57 hours, with the resulting arrays taking about 68GB of memory.

Then, during Phase 2 we merged the constructed data structures in pairs used to construct the colored graphs in Phase 3. The average time to merge each pair depends on their sizes. The fastest merge took approximately 27 hours, between *D. ananassae* and *D. melanogaster*, the slowest merge took approximately 60 hours, between *D. grimshawi* and *D. sechellia*. The size of the merged files was approximately the sum of the sizes of the input files. The document array file has the same size of the BWT merged file, since both store each value using one byte.

[1] `https://nanoporetech.com/products/minion`
[2] `https://www.illumina.com/systems/sequencing-platforms/nextseq.html`

Table 5.2: Construction information on data structures for the Drosophilas genomes.

Organism	BWT	LCP	CL	Time	LCP avg	LCP max
D. melanogaster	5.9GB	12GB	12GB	17.30h	61.54	302
D. ananassae	6.7GB	14GB	14GB	18.37h	55.56	302
D. simulans	8.7GB	18GB	18GB	24.18h	58.96	302
D. virilis	11GB	21GB	21GB	28.45h	55.06	302
D. willistoni	11GB	22GB	22GB	37.31h	57.79	302
D. pseudoobscura	12GB	23GB	23GB	40.07h	58.72	302
D. mojavensis	12GB	24GB	24GB	32.65h	58.09	302
D. yakuba	12GB	24GB	24GB	42.42h	59.98	302
D. persimilis	13GB	25GB	25GB	35.30h	58.76	302
D. erecta	14GB	27GB	27GB	42.92h	60.47	302
D. sechellia	14GB	27GB	27GB	37.88h	61.25	302
D. grimshawi	14GB	27GB	27GB	57.4h	43.07	2648

5.1.3 Phylogenetic trees

We expect to reconstruct a phylogeny that agrees with the phylogeny in Figure 5.1, which we will refer to as *reference phylogeny* during the analysis of the experiments. In Figure 5.5 we can observe Drosophila groups and subgroups, which may help our analysis.

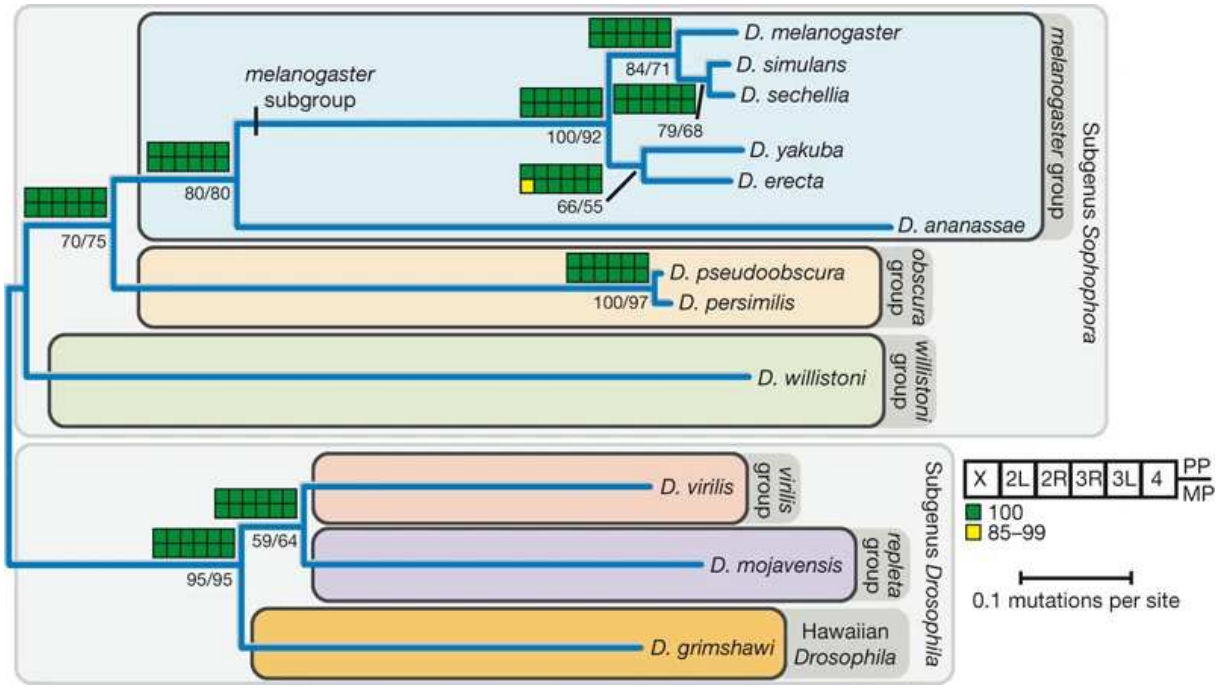
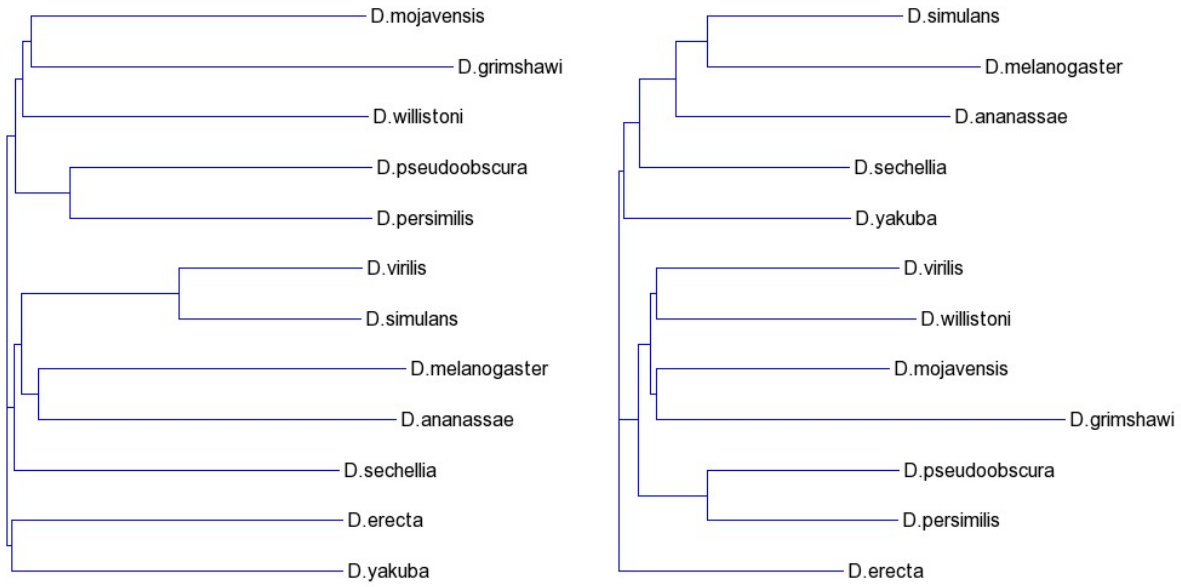


Figure 5.5: Drosophila phylogeny with groups and subgroups divisions, from [7]

We ran gcBB using different values for k, namely $k = 15, 31, 63$, each experiment outputs an entropy and an expectation matrix which were used to produce the phylogenies. Also, we considered the graphs with and without coverage information in the BWSD computation.

The phylogenies using $k = 15$ can be observed in Figure 5.6. In both phylogenies (based entropy and expectation matrices) we can observe that the pair *D. pseudoobscura*

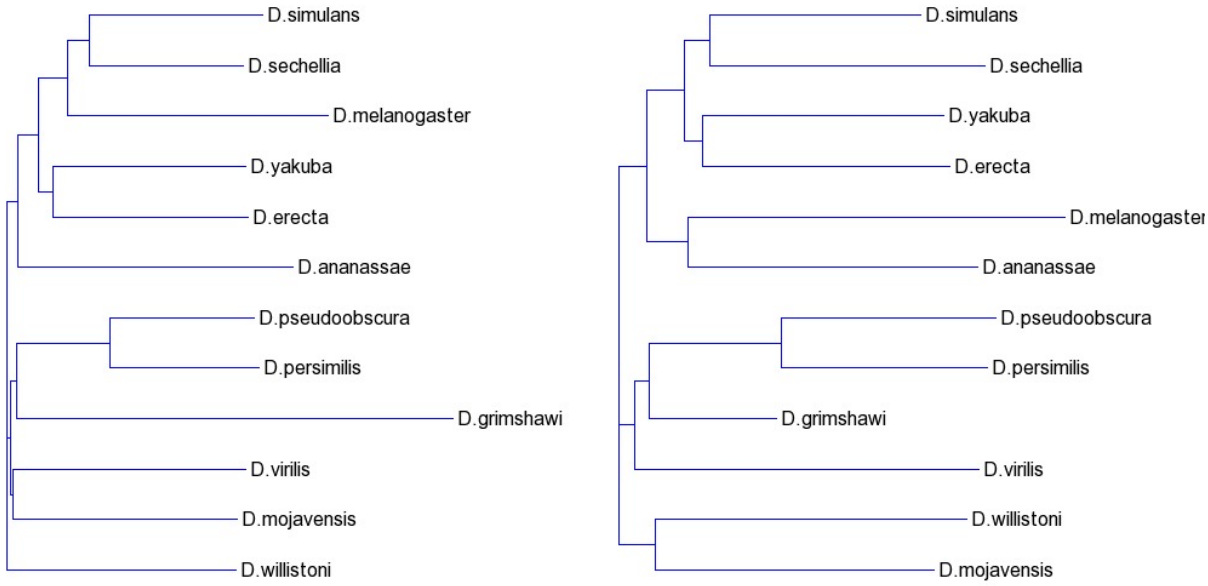
and *D. persimilis* from the *obscura* group agrees with the reference phylogeny. In Figure 5.6a we can also observe that *D. yakuba* and *D. erecta* from *melanogaster* subgroup are segregated together, agreeing with the reference. In Figure 5.6b we can observe a subtree composed by *D. willistoni*, *D. virilis*, *D. mojavensis* and *D. grimshawi*, which are species placed outside of the *melanogaster* and *obscura* groups. In general, both phylogenies disagree with the reference phylogeny.



(a) (b)

Figure 5.6: gcBB phylogenies for Drosophilas with $k = 15$, (a) using entropy, (b) using expectation.

By including coverage information in the BWSD computation, we obtained the phylogenies shown in Figure 5.7. We observe that the genomes that were close and agreed with the reference phylogeny in the previous experiment are still grouped together. In both phylogenies (for entropy and expectation matrices) we have a clear separation between the *melanogaster* group and the outside groups. With entropy we have the exact *D. melanogaster* group, while when using expectation we have some genomes positioned in different branches. From the Drosophila groups and subgroups shown in Figure 5.5 we can observe that genomes from outside the *melanogaster* group (*D. pseudoobscura, D. persimilis, D. grimshawi, D. virilis, D. mojavensis, D. willistoni*) still disagree with the reference phylogeny. In this case, there is an indication that the coverage information helped separating the *melanogaster* group from the other genomes.



(a) (b)

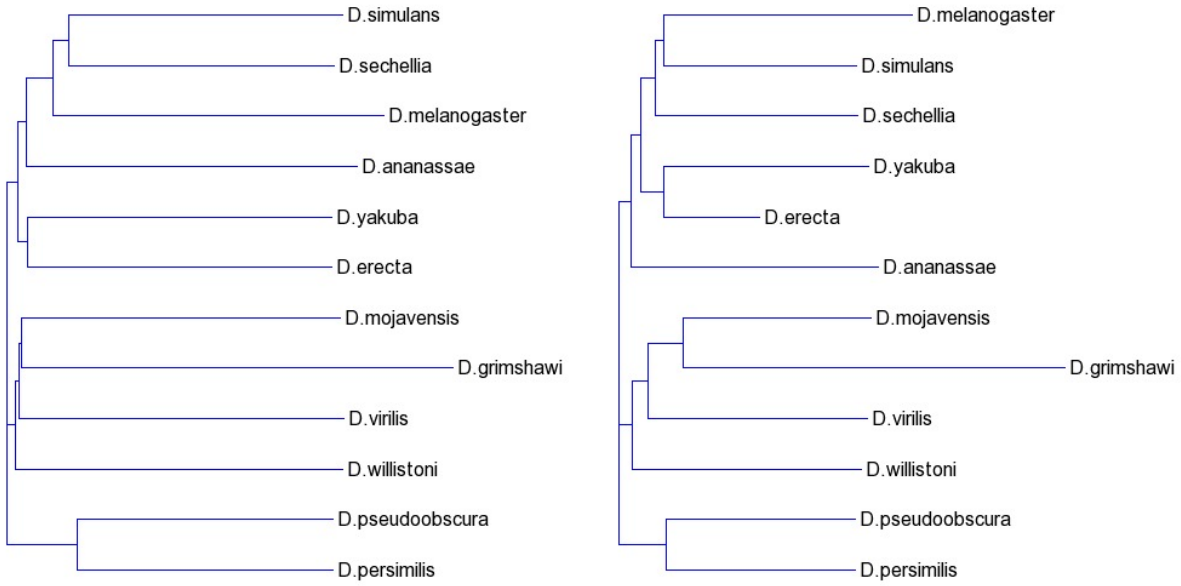
Figure 5.7: gcBB phylogenies for Drosophilas with $k = 15$ and coverage information, (a) using entropy, (b) using expectation.

For $k = 31$ we have the phylogenies shown in Figure 5.8. Again, the Drosophila groups and subgroups shown in Figure 5.5 we can observe that in both phylogenies (for entropy and expectation matrices) there is a division between the *melanogaster* group, the *obscura* group and the remaining groups together in a subtree (*willistoni* group, *virilis* group, *repleta* group, Hawaiian *Drosophila*). We can observe a few inconsistencies inside the subgroups, but the high level groups division agrees with the reference phylogeny. Using coverage with $k = 31$ we obtained basically the same phylogenies, which are shown in Figure A.1 (Page 76).

Finally for $k = 63$ we can observe in Figure 5.9 that the phylogenies are very similar to those resulting with $k = 31$. The same happens with applied coverage, which can be observed in Figure A.2 (Page 77).

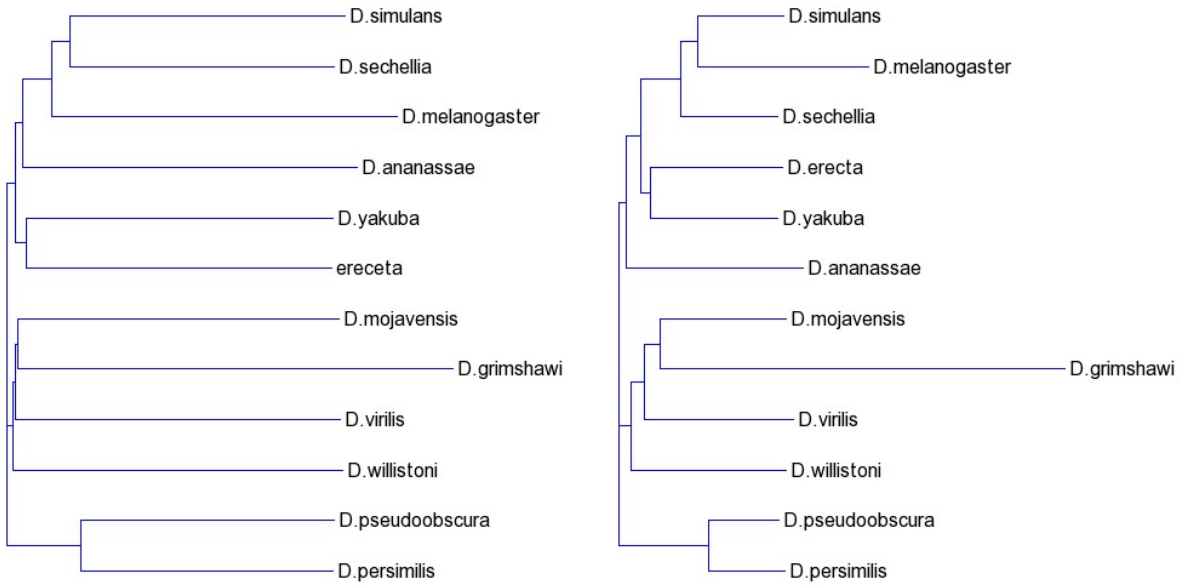
For a pair of phylogenetic trees T_1 and T_2 with n vertices labeled by $\mathcal{U} = \{1, 2, \ldots, n\}$, the distance by Robinson and Foulds [34] is the number of bipartitions of $\mathcal{U}$ exclusive to T_1 or to T_2 that may be obtained by removing a single edge of both T_1 or to T_2. The Robinson-Foulds distance values vary from 0 to $2n - 6$.

Computing the Robinson-Foulds [34] distance between our phylogenies and the reference phylogeny we obtained the values shown in Table 5.3. These results indicate that our method produces reasonable phylogenies when k is closer to the average LCP of the reads. Also, the use of coverage information reduced the Robinson-Foulds distance to the reference in most cases. Finally, the fact that all reads in the dataset were obtained using the same sequencing strategy and, in average, have a similar number of sequenced bases may have helped obtaining favorable results.



(a) (b)

Figure 5.8: gcBB phylogenies for Drosophilas with $k = 31$, (a) using entropy, (b) using expectation.



(a) (b)

Figure 5.9: gcBB phylogenies for Drosophilas with $k = 64$, (a) using entropy, (b) using expectation.

5.1.4 Effect of data size

In order to evaluate the effect of the number of sequenced reads in the resulting phylogeny, we considered a dataset of Illumina HiSeq 2000 reads dataset of the *D. grimshawi*, described in Table 5.4. The running time taken by eGap to construct the data structures and their sizes are shown in Table 5.5.

Table 5.3: Robinson-Foulds distance computed between phylogenies constructed by gcBB and the reference phylogeny. The symbol c represents the phylogenies constructed using coverage information.

	15	15c	31	31c	63	63c
Entropy	7	2	2	1	2	1
Expectation	6	5	2	3	2	2

Table 5.4: Information on the genome of *D. grimshawi*. The bases column specifies the number of sequenced bases of the genome (in Gbp). The reference column specifies the size of the complete referenced genome (in Mb). All genomes can be easily accessed through its Run accession number or BioSample in `https://www.ncbi.nlm.nih.gov/genbank/`.

Organism	Run	BioSample	Bases	Reference
D. grimshawi	SRR7642855	SAMN09764638	1.80	191.38

Table 5.5: Construction information for *D. grimshawi*.

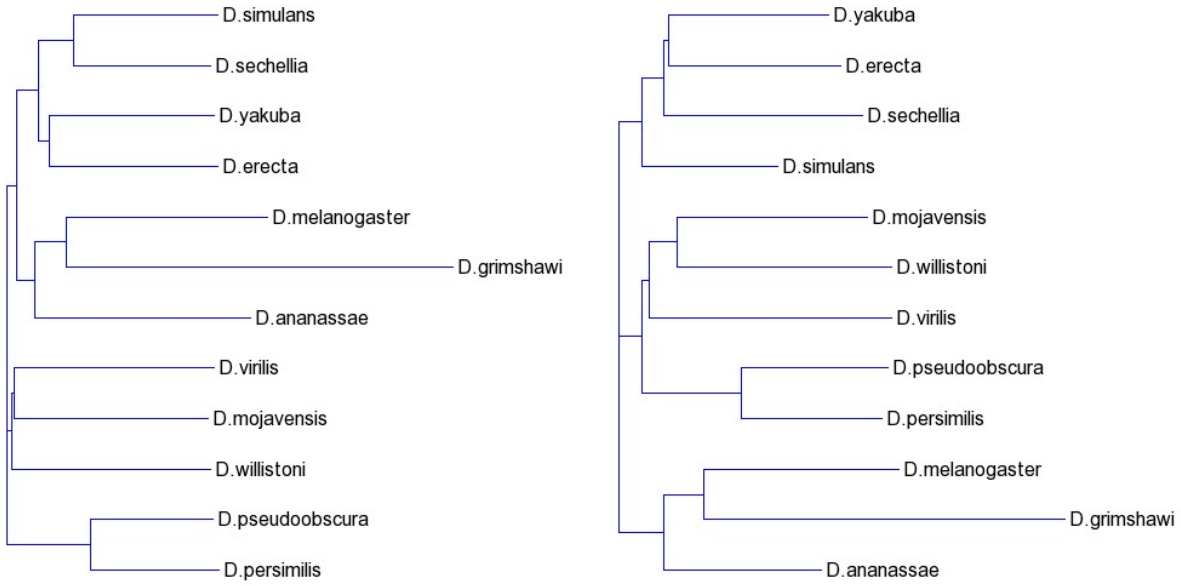
Organism	BWT	LCP	CL	Time	LCP avg	LCP max
D. grimshawi	1.80GB	3.5GB	3.5GB	2.22h	28.74	76

We executed gcBB using the same parameters and values of k. The best phylogeny was obtained with $k = 15$ and using coverage information, and is shown in Figure 5.10.

We can observe that in both phylogenies the genomes outside the *melanogaster* group are correct, but the *melanogaster* group and subgroup disagree with the reference phylogeny. In the phylogenies for $k = 31$ and $k = 64$ (with and without coverage information) we had similar trees with few variations in the *melanogaster* subgroup with *D. simulans*, *D. schelia*, *D. yakuba* and *D. erecta.* Moreover, all phylogenies had a subtree containing *D. ananassae*, *D. melanogaster* and *D. grimshawi*, which disagree with the reference phylogeny.

Our feeling is that we have a limitation when there are significantly distinct amounts of sequenced bases. In this example, we have *D. grimshawi* with 1.80G sequenced bases, while we have genomes as *D. sechellia* and *D. simulans* exceeding 14G of sequenced bases. When constructing the BOSS representation for *D. grimshawi* and *D. sechellia* there will be many more edges from *D. sechellia* than from *D. griwshawi,* and the similarity between these two genomes tends to be small. On the other hand, when constructing the BOSS representation for *D. grimshawi* and *D. melanogaster* there will also be many more edges from *D. melanogaster*. Moreover, the difference between the amount of sequenced bases from *D. melanogaster* to *D. grimshawi* is around 4GB, while from *D. sechellia* to *D. grimshawi* is around 12GB.

Therefore we conclude that among these phylogenies, *D. grimshawi* will always be closer to the smallest genomes, such as *D. melanogaster* and *D. ananassae*, which can be observed in Figure 5.10 and in Figures A.3-A.4 (Pages 77-78).



(a) (b)

Figure 5.10: gcBB phylogenies for Drosophilas with *D. grimshawi* with $k = 15$ and coverage information, (a) using entropy, (b) using expectation.

When we compute the Robinson-Foulds distance between these phylogenies and the reference phylogeny we obtained the values in Table 5.6 notably larger than the previous ones.

Table 5.6: Robinson-Foulds distance computed between phylogenies with *D. grimshawi* from another experiment and the reference phylogeny. The symbol c represents the phylogenies constructed using coverage information.

	15	15c	31	31c	63	63c
Entropy	7	5	5	5	5	5
Expectation	6	7	6	5	6	6

5.2 Vibrios

Thompson *et al.* [39] constructed 6 phylogenies of 16 Vibrio genomes combining 16S rRNA, Multilocus Sequence Analysis (MLSA) and Supertree with Maximum Parsimony and Neighbor-Joining. For the 16S rRNA trees, the sequences of 16S rRNA genes were concatenated. For MLSA and Supertree, different sets of house-keeping genes were concatenated. The sequences were aligned by the software CLUSTALX and trees were generated using the softwares MEGA and PAUP.

We have selected 9 genomes out of those 16 to test gcBB. The selected genomes are described in Table 5.7. The reference phylogeny for our analysis was constructed using Supertree and Neighbor-Joining and is shown in Figure 5.11. We have selected Supertree because it covers a larger portion of the genes in Vibrios, and Neighbor-Joining to be

consistent with our experiment with Drosophilas. Also, we can observe Vibrio groups in Figure 5.11, which may help our analysis.

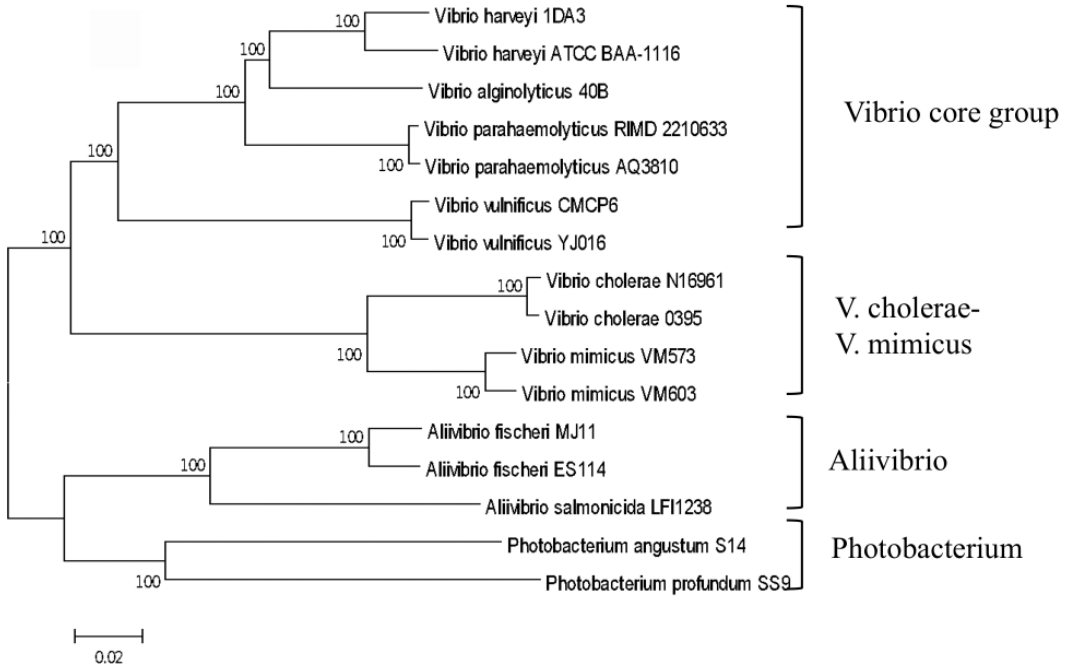


Figure 5.11: Phylogenetic tree for supertree and Neighbour-Joining. Figure from [39].

Table 5.7: Information on the genomes of Vibrios. The bases column specifies the number of sequenced bases of the genome (in Mbp). The reference column specifies the size of the complete referenced genome (in Mb). Reads column specifies the average length of the reads. All genomes can be easily accessed through its Run accession number or BioSample in `https://www.ncbi.nlm.nih.gov/genbank/`.

Organism	Run	BioSample	Bases	Reference	Reads
A. fischeri	SRR13978570	SAMN18318596	503.71	4.11	150
P. angustum	SRR6219161	SAMN07327723	267.39	4.88	226
P. profundum	SRR6219413	SAMN07327759	243.41	6.19	201
V. alginolyticus	SRR9163535	SAMN11896020	306.79	5.18	285
V. cholerae	SRR8364453	SAMN10610880	441.21	4.02	250
V. harveyi	SRR2931635	SAMN04279316	322.22	5.02	251
V. mimicus	SRR8535051	SAMN10711948	130.89	4.31	242
V. parahaemolyticus	SRR5023255	SAMN05271621	218.79	5.11	240
V. vulnificus	SRR7239526	SAMN09270178	994.95	5.00	251

The time taken to construct the BWT, LCP and CL arrays for each genome and the arrays sizes are shown in Table 5.2. In this experiment we limited the memory to 8GB of RAM.

We ran gcBB using different values for k, namely $k = 15, 31, 63$. Computing the Robinson-Foulds distance between the phylogenies and the reference phylogeny we obtained the values that are shown in Table 5.9.

Table 5.8: Construction information on data structures for the Vibrios genomes.

Organism	BWT	LCP	CL	Time	LCP avg	LCP max
A. fischeri	780MB	1.6GB	1.6GB	2.21h	70.62	150
P. angustum	213MB	426MB	426MB	0.96h	92.26	251
P. profundum	181MB	361MB	361MB	0.76h	65.41	251
V. alginolyticus	247MB	493MB	493MB	1.33h	109.89	301
V. cholerae	333MB	665MB	665MB	1.37h	62.00	250
V. harveyi	237MB	474MB	474MB	1.01h	68.35	251
V. mimicus	111MB	222MB	222MB	0.50h	96.05	251
V. parahaemolyticus	176MB	351MB	351MB	0.79h	100.96	251
V. vulnificus	834MB	1.7GB	1.7GB	3.82h	109.63	251

Table 5.9: Robinson-Foulds distance computed between the phylogenies for Vibrios constructed by gcBB and the reference phylogeny. The symbol c represents the phylogenies constructed using coverage information.

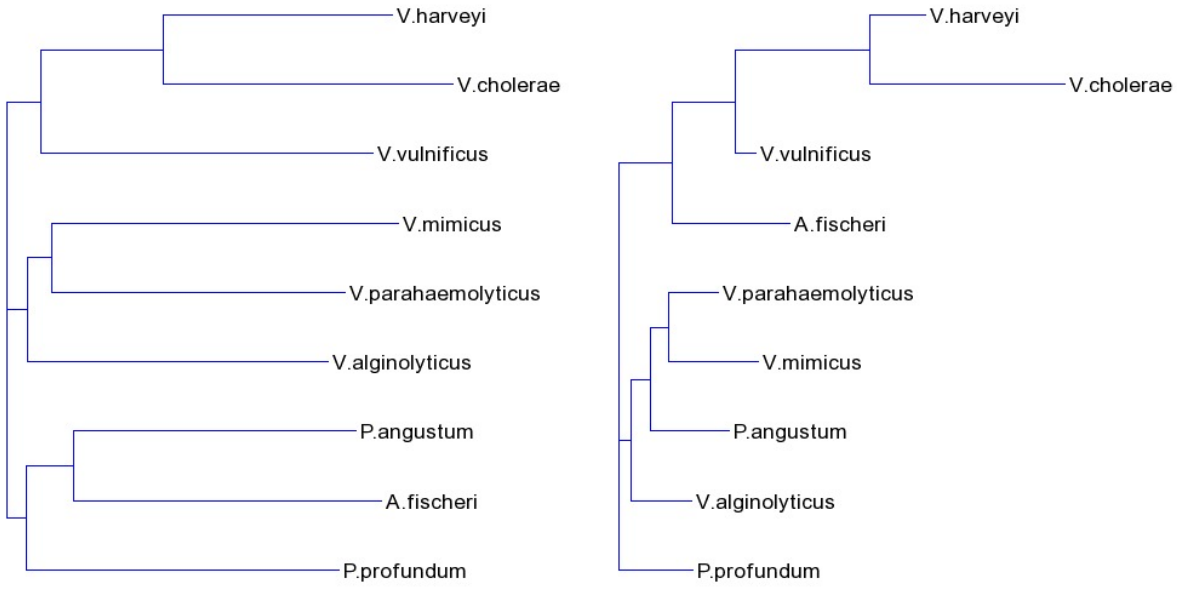
	15	15c	31	31c	63	63c
Entropy	6	5	6	5	6	6
Expectation	6	6	6	6	6	6

In general, we can observe that all phylogenies disagree with the reference. The best gcBB phylogenies were reconstructed from the $k = 15$ and 31 with coverage information entropy matrices.

For $k = 15$ with coverage information we have the phylogenies shown in Figure 5.12 (for entropy and expectation matrices). In Figure 5.12a we can observe that the *Allivibrio* and *Photobacterium* groups are in the correct subtree, but segregated incorrectly, since *P. angustum* and *P. profundum* should be closer. In Figure 5.12b we can observe that none of the groups is correctly segregated. We can also observe a significant difference between the branch lengths, which are considerably longer in the expectation matrix. Moreover, both phylogenies disagree with the reference.

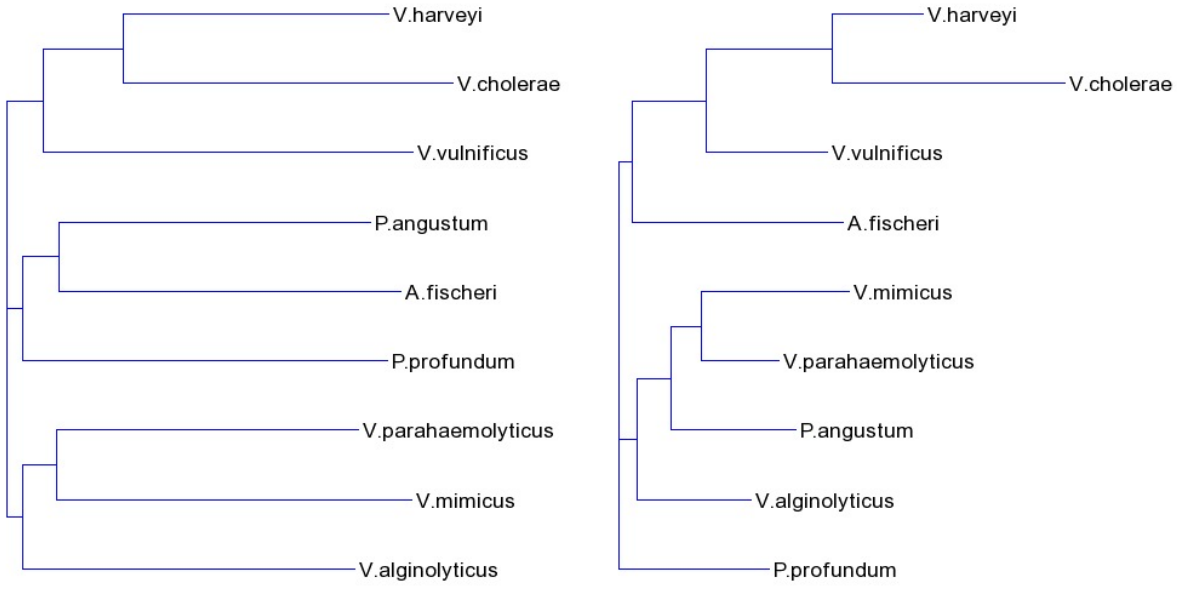
For $k = 31$ with coverage information we have the same phylogeny for $k = 15$ with coverage information, as shown in Figure 5.13. The Robinson-Foulds distance among these phylogenies is 0. The phylogenies generated for the remaining parameters can be observed in Figures B.1-B.4(Pages 79-81).

These results indicate that our method does not produce consistent phylogenies for small sets of reads. Again, the use of coverage information reduced the Robinson-Foulds distance to the reference in a few cases.



(a) (b)

Figure 5.12: gcBB phylogenies for Vibrios with $k = 15$ and coverage information, (a) using entropy, (b) using expectation.



(a) (b)

Figure 5.13: gcBB phylogenies for Vibrios with $k = 31$ and coverage information, (a) using entropy, (b) using expectation.

Chapter 6

Conclusions

In this dissertation we introduced a new method to compare draft genomes using space-efficient data structures implemented as the gcBB algorithm, an algorithm to compare sets of reads of genomes using the BOSS representation and to compute the similarity measures based on the BWSD.

We evaluated our algorithm reconstructing the phylogeny of the 12 *Drosophila* genomes. We used the Neighbor-Joining method over the matrices output by gcBB for the creation of the phylogenetic trees. Then we computed the Robinson-Foulds distance between our phylogenies and the reference. One issue when working with the de Bruijn graph is to set the value of k. We observed that values over the average LCP from the genomes produces more consistent results. We also observed better results using the entropy measure and coverage information.

We observed that gcBB does not output consistent results for genomes that does not have a large set of reads, as the `Vibrio` species in our experiments (less than 1GB of sequenced bases). Also, the phylogeny may become disorganized when there is one genome with considerably less sequenced bases than the other genomes in the set.

Further experiments may help understanding the limits and the advantages of the approach introduced in this work. Future research may also investigate different strategies for dealing with coverage information, as the experiments indicate a positive contribution of coverage to the resulting phylogenies. The quality of sequenced bases may also be investigated in future work, as a means to improve the method.

Bibliography

[1] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern information retrieval*, volume 463. ACM Press New York, 1999.

[2] Monya Baker. De novo genome assembly: what every biologist should know. *Nature methods*, 9(4):333–337, 2012.

[3] Paola Bonizzoni, Gianluca Della Vedova, Yuri Pirola, Marco Previtali, and Raffaella Rizzi. Multithread multistring Burrows-Wheeler transform and longest common prefix array. *J. Comput. Biol.*, 26(9):948–961, 2019.

[4] Alexander Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya. Succinct de Bruijn graphs. In *WABI*, volume 7534 of *Lecture Notes in Computer Science*, pages 225–235. Springer, 2012.

[5] Michael Burrows and David Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital SRC Research Report, 1994.

[6] Rudi Cilibrasi and Paul M. B. Vitányi. Clustering by compression. *IEEE Transactions on Information Theory*, 51(4):1523–1545, 2005.

[7] Andrew G Clark and Lior Pachter. Evolution of genes and genomes on the Drosophila phylogeny. *Nature*, 450(7167):203–218, 2007.

[8] Nicolaas Govert De Bruijn. A combinatorial problem. In *Proc. Koninklijke Nederlandse Academie van Wetenschappen*, volume 49, pages 758–764, 1946.

[9] Lavinia Egidi, Felipe A Louza, and Giovanni Manzini. Space-efficient merging of succinct de Bruijn graphs. In *Proc. SPIRE*, volume 11811 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 2019.

[10] Lavinia Egidi, Felipe A Louza, Giovanni Manzini, and Guilherme P Telles. External memory BWT and LCP computation for sequence collections with applications. *Algorithms for Molecular Biology*, 14(1):1–15, 2019.

[11] Joseph Felsenstein. *Inferring phylogenies*. Sinauer Associates Sunderland, MA, 2004.

[12] Paolo Ferragina, Rodrigo González, Gonzalo Navarro, and Rossano Venturini. Compressed text indexes: From theory to practice. *Journal of Experimental Algorithmics (JEA)*, 13:1–12, 2009.

[13] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 390–398. IEEE, 2000.

[14] Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, July 2005.

[15] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. 2003.

[16] Matthew W Hahn, Mira V Han, and Sang-Gook Han. Gene family evolution across 12 drosophila genomes. *PLoS genetics*, 3(11):e197, 2007.

[17] Zamin Iqbal, Mario Caccamo, Isaac Turner, Paul Flicek, and Gil McVean. De novo assembly and genotyping of variants using colored de Bruijn graphs. *Nature genetics*, 44(2):226–232, 2012.

[18] Ananth Kalyanaraman. *Genome Assembly*. Springer US, Boston, MA, 2011.

[19] Kazutaka Katoh and Daron M. Standley. MAFFT Multiple Sequence Alignment Software Version 7: Improvements in Performance and Usability. *Molecular Biology and Evolution*, 30(4):772–780, 01 2013.

[20] Mikhail Kolmogorov, Derek M Bickhart, Bahar Behsaz, Alexey Gurevich, Mikhail Rayko, Sung Bong Shin, Kristen Kuhn, Jeffrey Yuan, Evgeny Polevikov, Timothy PL Smith, and Pavel A Pevzner. MetaFlye: scalable long-read metagenome assembly using repeat graphs. *Nature Methods*, 17(11):1103–1110, 2020.

[21] Ben Langmead and Steven L Salzberg. Fast gapped-read alignment with bowtie 2. *Nature Methods*, 9(4):357, 2012.

[22] Heng Li and Richard Durbin. Fast and accurate short read alignment with Burrows–Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[23] Felipe A Louza, Guilherme P Telles, Simon Gog, Nicola Prezza, and Giovanna Rosone. gsufsort: constructing suffix arrays, LCP arrays and BWTs for string collections. *Algorithms Mol. Biol.*, 15(1):18, 2020.

[24] Felipe A Louza, Guilherme P Telles, Simon Gog, and Liang Zhao. Algorithms to compute the Burrows-Wheeler similarity distribution. *Theor. Comput. Sci.*, 782:145–156, 2019.

[25] Cole A Lyman, M Stanley Fujimoto, Anton Suvorov, Paul M Bodily, Quinn Snell, Keith A Crandall, Seth M Bybee, and Mark J Clement. Whole genome phylogenetic tree reconstruction using colored de Bruijn graphs. In *2017 IEEE 17th International Conference on Bioinformatics and Bioengineering (BIBE)*, pages 260–265. IEEE, 2017.

[26] Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I. Tomescu. *Genome-Scale Algorithm Design*. Cambridge University Press, 2015.

[27] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[28] S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino. An extension of the Burrows–Wheeler transform. *Theoretical Computer Science*, 387(3):298 – 312, 2007. The Burrows-Wheeler Transform.

[29] Danny E Miller, Cynthia Staber, Julia Zeitlinger, and R Scott Hawley. Highly contiguous genome assemblies of 15 Drosophila species generated using nanopore sequencing. *G3: Genes, Genomes, Genetics*, 8(10):3131–3141, 2018.

[30] Ge Nong. Practical linear-time $O(1)$-workspace suffix sorting for constant alphabets. *ACM Transactions on Information Systems*, 31(3):1–15, 2013.

[31] Evgeny Polevikov and Mikhail Kolmogorov. Synteny paths for assembly graphs comparison. In *19th International Workshop on Algorithms in Bioinformatics (WABI 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[32] Nicola Prezza and Giovanna Rosone. Space-efficient construction of compressed suffix trees. *Theor. Comput. Sci.*, 852:138–156, 2021.

[33] Raffaella Rizzi, Stefano Beretta, Murray Patterson, Yuri Pirola, Marco Previtali, Gianluca Della Vedova, and Paola Bonizzoni. Overlap graphs and de Bruijn graphs: data structures for de novo genome assembly in the big data era. *Quantitative Biology*, 7(4):278–292, 2019.

[34] David F Robinson and Leslie R Foulds. Comparison of phylogenetic trees. *Mathematical Biosciences*, 53(1-2):131–147, 1981.

[35] Naruya Saitou and Masatoshi Nei. The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Molecular Biology and Evolution*, 4(4):406–425, 07 1987.

[36] Karsten Scheibye-Alsing, Steve Hoffmann, A Frankel, Peter Jensen, Peter F Stadler, Yuan Mang, Niels Tommerup, Mike J Gilchrist, A-B Nygård, Susanna Cirera, Claus B Jörgensen, Merete Fredholm, and Jan Gorodkin. Sequence assembly. *Computational Biology and Chemistry*, 33(2):121–136, 2009.

[37] Joao Carlos Setubal and Joao Meidanis. *Introduction to computational molecular biology*. PWS Pub. Boston, 1997.

[38] Jared T Simpson and Richard Durbin. Efficient de novo assembly of large genomes using compressed data structures. *Genome Research*, 22(3):549–556, 2012.

[39] Cristiane C Thompson, Ana Carolina P Vicente, Rangel C Souza, Ana Tereza R Vasconcelos, Tammi Vesth, Nelson Alves, David W Ussery, Tetsuya Iida, and Fabiano L Thompson. Genomic taxonomy of Vibrios. *BMC Evolutionary Biology*, 9(1):1–16, 2009.

[40] Jim Thurmond, Joshua L Goodman, Victor B Strelets, Helen Attrill, L Sian Gramates, Steven J Marygold, Beverley B Matthews, Gillian Millburn, Giulia Antonazzo, Vitor Trovisco, Thomas C Kaufman, Brian R Calvi, and the FlyBase Consortium. FlyBase 2.0: the next generation. *Nucleic Acids Research*, 47(D1):D759–D765, 10 2018.

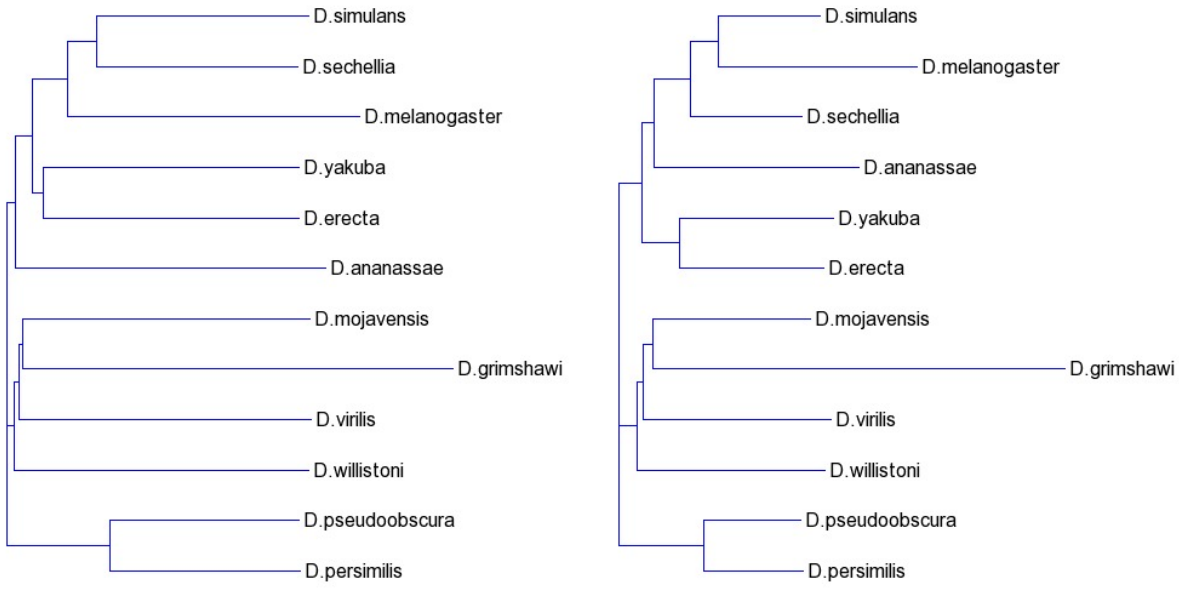
[41] Lianping Yang, Xiangde Zhang, and Tianming Wang. The Burrows–Wheeler similarity distribution between biological sequences based on Burrows–Wheeler transform. *Journal of Theoretical Biology*, 262(4):742 – 749, 2010.

Appendix A

Drosophilas

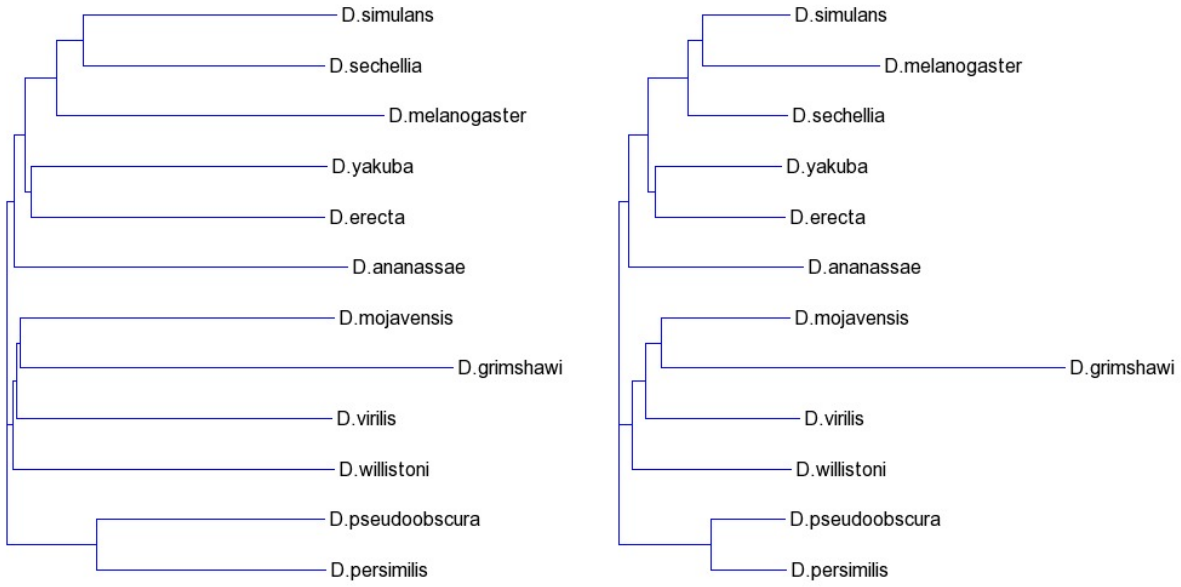
A.1 Additional phylogenies for 12 Drosophilas

In this appendix we present the additional phylogenies produced by gcBB for the 12 Drosophilas.



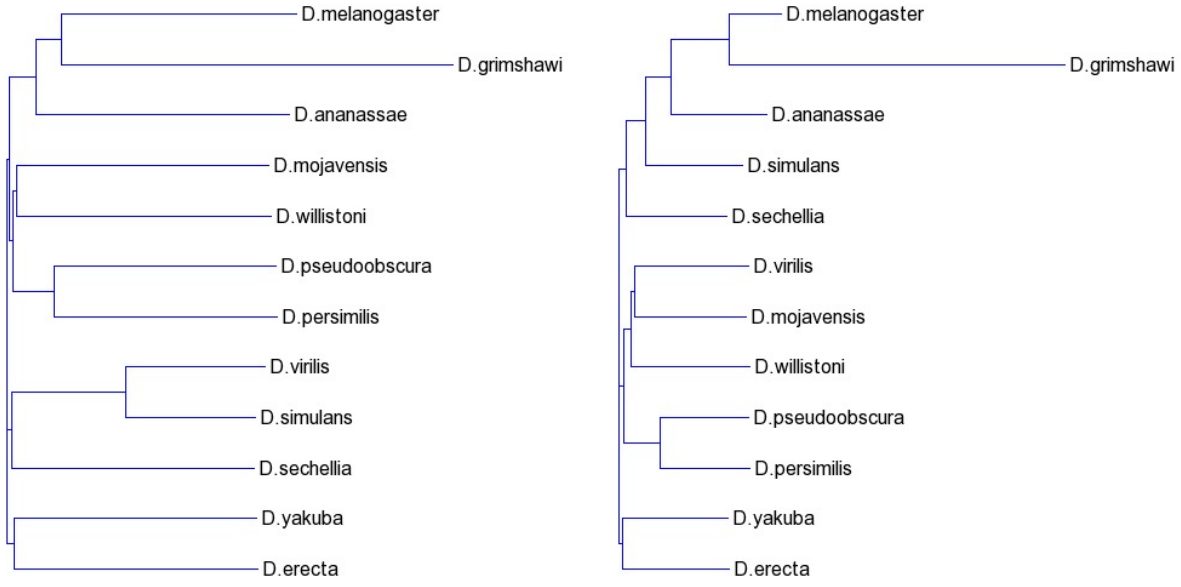
(a) (b)

Figure A.1: gcBB phylogenies for Drosophilas with $k = 31$ and coverage information, (a) using entropy, (b) using expectation.



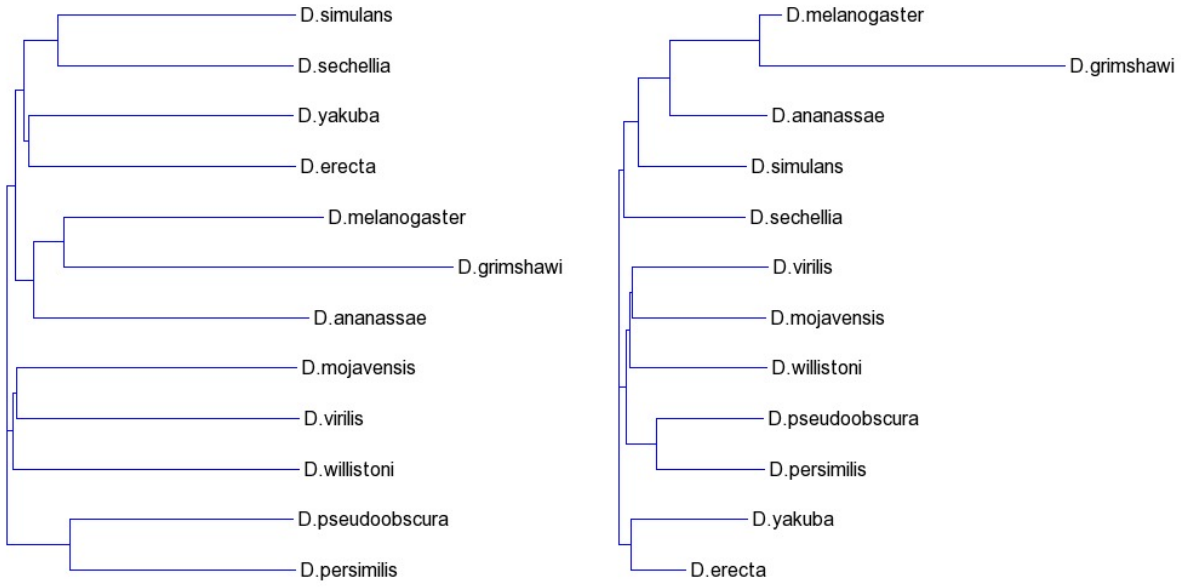
(a) (b)

Figure A.2: gcBB phylogenies for Drosophilas with $k = 63$ and coverage information, (a) using entropy, (b) using expectation.



(a) (b)

Figure A.3: gcBB phylogenies for Drosophilas with distinct *D. grimshawi* with $k = 15$, (a) using entropy, (b) using expectation.



(a) (b)

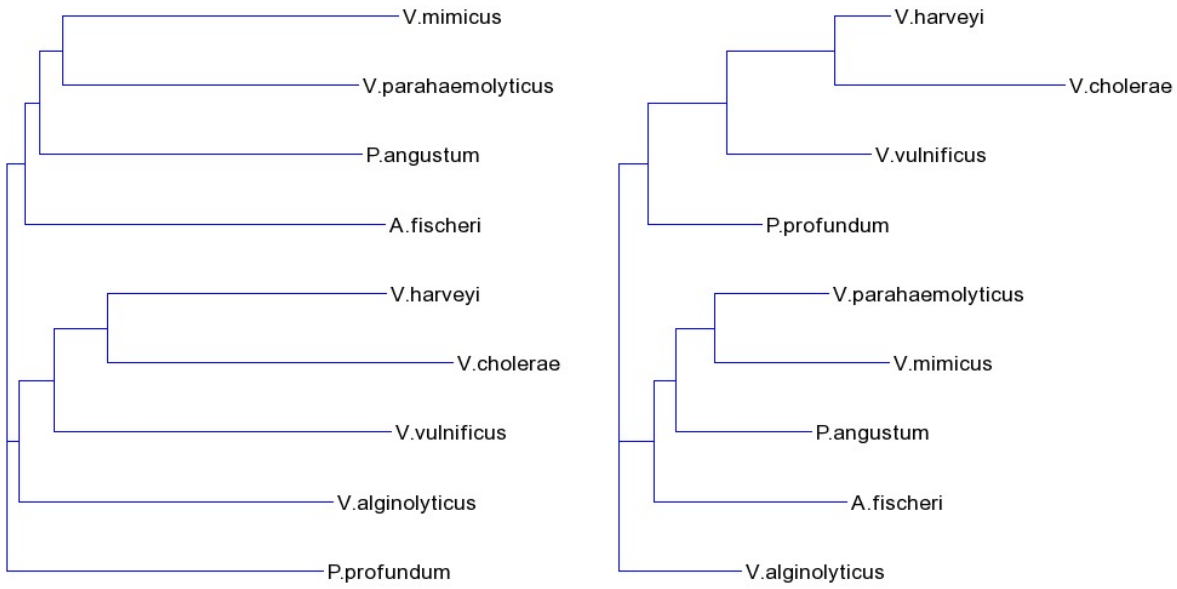
Figure A.4: gcBB phylogenies for Drosophilas with distinct *D. grimshawi* with $k = 31$, (a) using entropy, (b) using expectation.

Appendix B

Vibrios

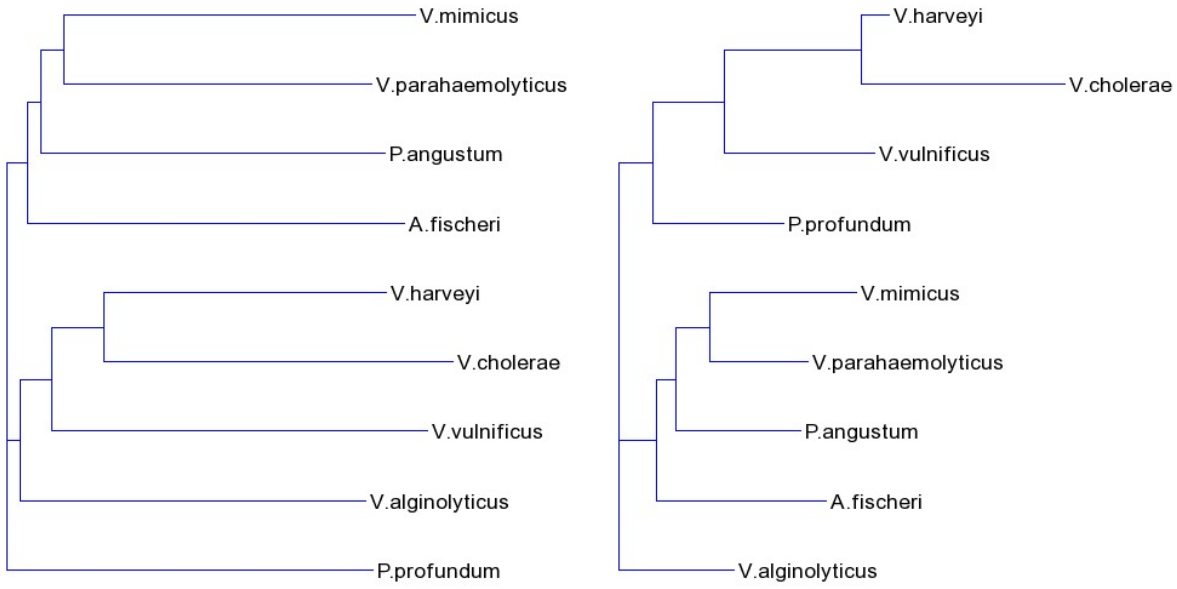
B.1 Additional phylogenies for 6 Vibrios

In this appendix we present the additional phylogenies produced by gcBB for the 6 Vibrios.



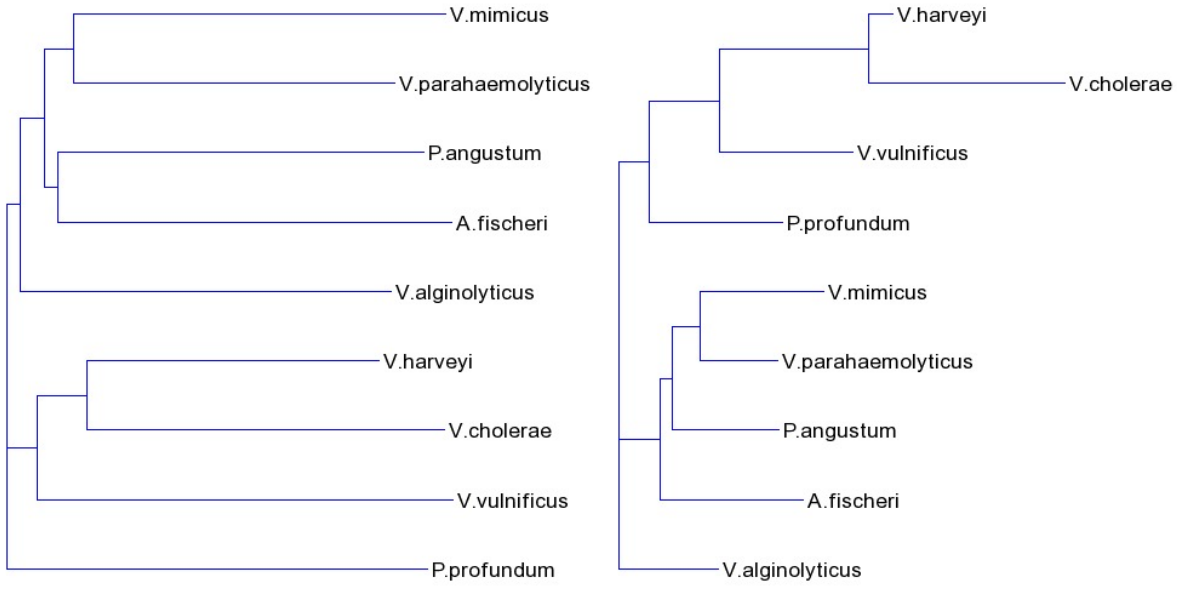
(a) (b)

Figure B.1: gcBB phylogenies for Vibrios with $k = 15$, (a) using entropy, (b) using expectation.



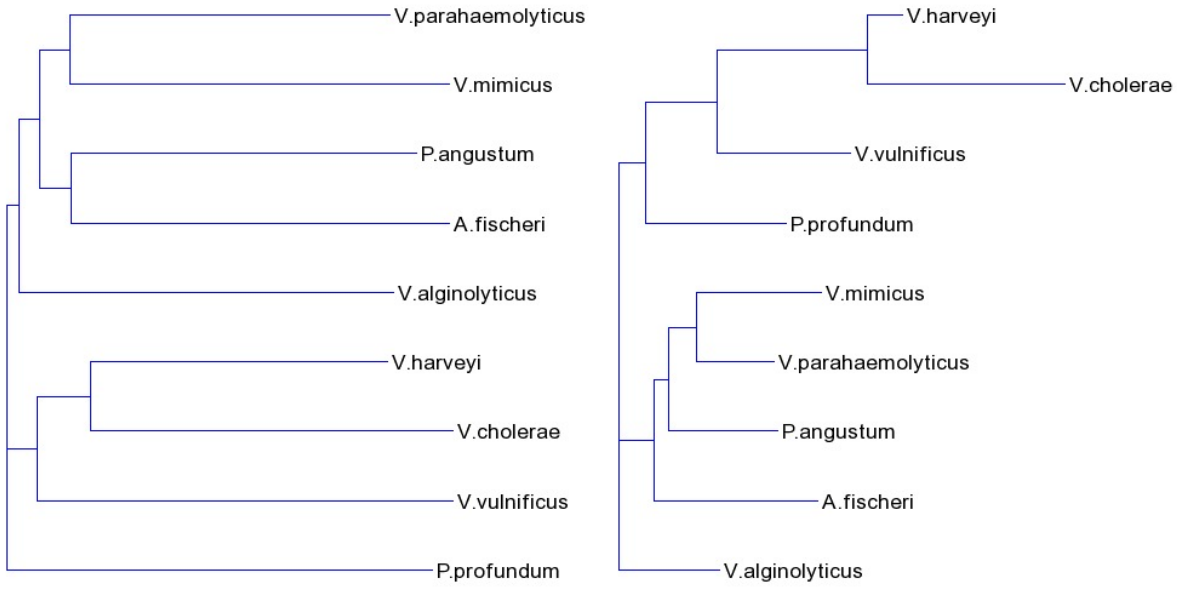
(a) (b)

Figure B.2: gcBB phylogenies for Vibrios with $k = 31$, (a) using entropy, (b) using expectation.



(a) (b)

Figure B.3: gcBB phylogenies for Vibrios with $k = 63$, (a) using entropy, (b) using expectation.



(a) (b)

Figure B.4: gcBB phylogenies for Vibrios with $k = 63$ and coverage information, (a) using entropy, (b) using expectation.