View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Aaltodoc Publication Archive

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Jarno Niklas Alanko

Space-efficient clustering of metagenomic read sets

Master's Thesis
Espoo, December 31, 2015

Supervisors:	Professor Jorma Tarhio, Aalto University
Advisor:	Ph.D. Fabio Cunial
	Ph.D. Djamal Belazzougui

Aalto University
School of Science

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author:	Jarno Niklas Alanko		
Title: Space-efficient clustering of metagenomic read sets			
Date:	December 31, 2015	**Pages:**	66
Major:	Information and Computer Science	**Code:**	T-79
Supervisors:	Professor Jorma Tarhio		
Advisor:	Ph.D. Fabio Cunial Ph.D. Djamal Belazzougui		

The collection of all genomes in an environment is called the *metagenome* of the environment. In the past 15 years, high-throughput sequencing has made it feasible to sequence entire environments at once for the first time in history, which has resulted in a variety of interesting new algorithmic problems. This thesis focuses on the basic problem of clustering the reads from an environment according to which species, or more generally, taxonomic unit they originate from.

In this work, we identify and formalize two fundamental string processing tasks useful in clustering metagenomic read sets. We solve the two problems with space efficiency in mind using the recently developed bidirectional Burrows-Wheeler index. The algorithms were implemented in a way which makes parallel processing possible. Our tool is experimentally shown to give good results for simple simulated datasets, and to use less than 10 times less space and time compared to two recently published metagenome clustering tools.

Keywords:	Burrows-Wheeler transform, metagenomics, clustering, space-efficient
Language:	English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan koulutusohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Jarno Niklas Alanko		
Työn nimi: Tilatehokas metagenomisten DNA-fragmenttien ryhmittely			
Päiväys:	31. joulukuuta 2015	**Sivumäärä:**	66
Pääaine:	Tietojenkäsittelytiede	**Koodi:**	T-79
Valvojat:	Professori Jorma Tarhio		
Ohjaaja:	Ph.D. Fabio Cunial Ph.D. Djamal Belazzougui		

Kaikkien ympäristössä esiintyvien genomien joukkoa kutsutaan kyseisen ympäristön *metagenomiksi*. Viimeisen 15 vuoden aikana kehitetyt korkean läpisyötön sekvenssoriteknologiat ovat mahdollistaneet ensimmäistä kertaa historiassa kokonaisen ympäristön metagenomin kartoittamisen. Tämä kehityssuunta on johtanut uusiin mielenkiintoisiin algoritmisiin ongelmiin. Tämä työ käsittelee ympäristöistä näytteistettyjen DNA-fragmenttejen ryhmittelyä lajien, tai yleisemmin taksonomisten yksiköiden mukaan.

Työssä tunnistetaan ja formalisoidaan kaksi merkkijono-ongelmaa, jotka ilmentyvät metagenomisten DNA-fragmentteja ryhmittelyssä. Ongelmiin esitetään tilatehokkaat ratkaisut käyttäen hiljattain kehitettyä kaksisuuntaista Burrows-Wheeler indeksiä. Algoritmit toteutettiin pitäen silmällä rinnakkaista laskentaa. Työssä osoitetaan, että uusi toteutus antaa hyviä tuloksia yksinkertaisille simuloiduille näytteille, ja että työkalu on kymmenen kertaa nopeampi ja tilatehokkaampi, kuin kaksi hiljattain julkaistua metagenomisten näytteiden ryhmittelyyn tarkoitettua työkalua.

Asiasanat:	Burrows-Wheeler muunnos, metagenomiikka, ryhmittely, tilatehokas
Kieli:	Englanti

Acknowledgements

I wish to thank my advisors Fabio Cunial, Djamal Belazzougui for sustained guidance, professor Veli Mäkinen for helpful discussions, professor Jorma Tarhio for acting as the supervisor for the thesis, my friends Mohsin Ali Khan, Antti Laaksonen and Jani Koskinen for helping with proofreading, and the algorithm competition circle of people in Finland for support and encouragement.

Espoo, December 31, 2015

Jarno Niklas Alanko

Notation and acronyms

Throughout the thesis, the letter S is reserved for the string for which an index has been built. The alphabet of S is denoted with Σ, and the size of the alphabet is denoted with σ. Greek letters $\alpha, \beta, \ldots$ are used to denote substrings of S and characters $c, d, \ldots$ are used to denote single characters. All indexing in the thesis starts from 1.

Notation

Σ	The alphabet of a string.
σ	Size of the alphabet of a string.
$T[i]$	The character at index i of string T.
$T[i..j]$	The substring of T between indices i and j, inclusive.
$T <_{lex} T'$	The string T is lexicographically smaller than the string T'.
$T <_{colex} T'$	The string T is colexicographically smaller than the string T'.
TT'	The concatenation of T and T'.
$\|T\|$	The length of the string or array T.
$\bar{T}$	The reverse of T.
$\widetilde{T}$	The reverse complement of T.
$[i_\alpha^\rightarrow, j_\alpha^\rightarrow]$	The lexicographic interval of a substring α.
$[i_\alpha^\leftarrow, j_\alpha^\leftarrow]$	The colexicographic interval of a substring α.

Acronyms

BWT	Burrows-Wheeler transform
SA	Suffix array
ISA	Inverse suffix array

Contents

Notation and acronyms **5**

1 Introduction **8**

2 Biological background **10**
2.1 The structure of the DNA molecule 10
2.2 DNA sequencing technologies 11
2.3 Taxonomy 11
2.4 Metagenomics 12

3 Preliminary data structures **13**
3.1 Model of computation 13
3.2 Suffix trees 14
3.3 Suffix arrays 17
3.4 Bit vectors with rank and select support 18
3.5 Wavelet trees 20
3.6 The Burrows-Wheeler transform 21
3.6.1 Inverting the BWT 23
3.6.2 Searching for patterns using the Burrows-Wheeler transform 24
3.6.3 The relationship with the suffix tree 25
3.7 Union-find data structure 26

4 The bidirectional BWT index **27**
4.1 Left- and right extensions 28
4.2 Left- and right maximality 30
4.3 Iterating all right-maximal nodes of the suffix tree 31
4.4 Iterating all nodes of the suffix tree using only the forward BWT 32
4.5 Applications of the bidirectional index in bioinformatics 32
4.5.1 Iterating all reverse complement right-maximal substrings 33

4.5.2 Finding the intervals of k-mers 34
4.5.3 Marking the intervals of k-submaximal repeats 35
4.5.4 Locating suffixes by sampling the suffix array 36
4.5.5 Generalization to multiple strings 37

5 Clustering metagenomic read sets 38
5.1 Challenges 39
5.2 Existing tools 39
5.3 MetaCluster 42
5.4 Solving the filtering problem with the bidirectional index . . . 47
5.5 Solving the precluster problem with the bidirectional index . . 48

6 Implementation 51
6.1 Correctness 52
6.2 Performance 53

7 Conclusion 60

Bibliography 62

Chapter 1

Introduction

Since the emergence of high-throughput sequencing, the amount of available genomic data has grown exponentially. An active field of research is the study of efficient algorithms and data structures to process the constant influx of new data. Several representations of classical text indices have been developed in the past decade [11, 14, 32] in order to meet the demand of efficient data structures stemming from high-throughput sequencing. Burrows-Wheeler type indices are especially popular in bioinformatics. The Burrows-Wheeler transform, originally designed for data compression [5], has proven to be a versatile and space-efficient data structure for string matching problems. The direction of research using the Burrows-Wheeler as a text index, initiated by P. Ferragina and G. Manzini [10] in the year 2000, has grown considerably in recent years. The aim of this work is to apply the bidirectional BWT index introduced in 2009 [22] to metagenome clustering.

Many algorithmic problems in bioinformatics stem from the limitations of the present sequencing technology. With the current technology, the DNA sequence of an organism can not be read in one piece, but instead has to be sequenced as a number of short segments sampled from random locations in the genome. The length of these *reads* typically range from tens of base pairs to a thousand base pairs [1]. Another limitation is that none of the next generation sequencing technologies can differentiate which of the complementary strands of the DNA helix a read was sampled from. Due to constraints imposed by molecular chemistry, the string of bases on one strand completely determines the string of bases on the other strand, and the two strands are transcribed in opposite directions. The sequences transcribed from opposite strands in the same location of the genome are called *reverse complements* of each other. Algorithms that work on datasets generated by high-throughput

[1] The exception being Pacific Bioscience's RS II sequencer, which can read over ten thousand base pairs at a time, but with a very low accuracy, see Table 2.1.

sequencing technologies should be able to exploit this structure of the DNA. It is shown in this thesis that the bidirectional BWT index is well suited to deal with this structure.

High-throughput sequencing has made it feasible to sequence entire environments at once. The collection of all genomes in an environment is called the *metagenome* of the environment. To this date, most metagenomics projects are run with short read sequencing technologies. A basic problem in metagenomics is to group the reads according to which species, or more generally, taxonomic unit they originate from. The problem has been studied in the literature for around 10 years (a survey of the literature is presented in Section 5.2). There exists a variety of tools to solve the clustering problem, but none of these tools address the issue of space efficiency, which is becoming a problem, since the sizes of the datasets are ever increasing.

In this work, we identify and formalize two fundamental string processing tasks useful in clustering metagenomic read sets. The first is related to separating high-abundance species from low-abundance species, and the second is related to finding and grouping overlapping reads together. We present a novel, space-efficient solution to these problems using the bidirectional Burrows-Wheeler index. Chapter 2 introduces the biological concepts necessary for the thesis and Chapter 3 fills in the necessary algorithmic preliminaries. Chapter 4 builds on Chapter 3 by presenting the bidirectional BWT index and our novel concept of *reverse complement right-maximal* substrings. Chapter 5 applies the data structures and concepts in Chapter 4 to implement a metagenomic read clustering pipeline, and the performance is briefly evaluated against two state-of-the art clustering tools in Chapter 6. Chapter 7 discusses the results in more detail and sketches a path forward.

Chapter 2

Biological background

DNA is the common information coding material in all living organisms on Earth. The sequence information coded in the DNA determines which proteins a cell produces, and at which rates. This in turn determines the biological function of the cell. In this chapter we describe the essential biology needed to understand the clustering problem.

2.1 The structure of the DNA molecule

DNA is a long molecule with two complementary strands running parallel in a helical structure. The strands are bound together by hydrogen bonds between the basic molecules. There are four different basic molecules between the strands: adenine, cytosine, guanine and thymine. These are commonly represented with the letters A,C,G and T, respectively. This allows us to represent a DNA strand as a string with alphabet $\{A, C, G, T\}$. A substring of k consecutive bases is called a k-mer. The molecular chemistry between the bases dictates that A and T can only bind with each other, and C and G can only bind with each other. Therefore, the sequence of one strand completely determines the sequence in the complementary strand. We denote the complement mapping with the function $f_c : \{A, C, G, T\} \to \{A, C, G, T\}$ defined by $f_c(A) = T, f_c(C) = G, f_c(G) = C$ and $f_c(T) = A$.

Both strands of the DNA can encode genes. The sequence on one strand is transcribed in a different direction from that of the other strand. For example, if the sequence ACGTACTAC is read from one strand, then the same segment of the genome will be read as GTAGTACGT from the other strand, i.e. the string is reversed and every character is complemented, or more formally:

Definition 2.1.1. *Reverse complement.* A substring α is the reverse complement of a substring β if and only if $|\alpha| = |\beta|$ and $f_c(\alpha[i]) = \bar{\beta}[i]$ for all indices i, where $\bar{\beta}$ is the reverse string of β.

2.2 DNA sequencing technologies

The DNA sequence of an organism can range from a few thousand base pairs of a viral genome to billions of base-pairs in the human genome. The sequence can not be read as a whole with currently available technology, but a large amount of small snippets of the code can be read, which can then be assembled to form the complete genome. These snippets are commonly called *reads* in the field of bioinformatics. The length of a read can vary from 50 base pairs to tens of thousands of base pairs, depending on the sequencing technology used (see Table 2.1).

DNA sequencing technology is prone to errors. The errors come in three types: insertions, deletions and substitutions. Some sequencing technologies are more prone to some types of errors than others. Algorithms that analyse read sets should somehow take into account these errors. A challenge is distinguishing natural single base mutations (called single nucleotide polymorphisms, or SNPs for short) from these sequencing errors.

Technology	**454 GS FLX**	**HiSeq 2000 (Illumina)**	**SOLIDv4**	**Sanger 3730xl**	**PacBio RS II**
Read length	700	100	50+35 or 50+50	400-900	10000-15000
Reads per run	10^6	$3 \cdot 10^9$	$1.3 \cdot 10^9$	$5 \cdot 10^4$	$5 \cdot 10^5$
Run time	24 hours	3-10 days	7-14 days	20mins - 3 hours	up to 4 hours
Accuracy	99.9%	98%	99.94%	99.999%	77.14%

Table 2.1: Comparison of sequencing technologies [1, 25]

2.3 Taxonomy

The classical definition of what a species is, given by Ernst Mayr in 1942, is "groups of actually or potentially interbreeding natural populations, which are reproductively isolated from other such groups" [29]. This definition is now known as the biological species concept. However, it does not work for bacteria, as bacteria reproduce asexually. For lack of a better alternative, bacterial taxonomists agreed in 1988 to define species on the basis of a DNA-DNA similarity of more than 70% [41]. This highlights the considerable DNA diversity within a single bacterial species, since if the threshold of 70% was

applied to animal classification, the whole order of primates would be a single species [34]. More sophisticated definitions, which integrate phenotypic, genotypic and phylogenetic information have also been proposed [37]. To this date, bacterial taxonomy is ultimately determined case by case by taxonomists without a rigorous method. For example, the species Escherichia coli encompasses a wide variety of strains with only 20% shared DNA [26], but it is still classified as a species for historical reasons. There exists authoritative taxonomy databases, such as the NCBI taxonomy database [9], which try to represent the latest consensus of the biological community.

2.4 Metagenomics

Metagenomics is the study of the genomes present in an environment as a single entity. The environment can be for example the human gut, or a freshwater lake. The set of the genomes of all species present in an environment is called the *metagenome* of the environment. The metagenome can be approximated by indiscriminately sampling reads from all DNA that is present in the environment. Such a sample is called a *metagenomic shotgun dataset*.

Determining the species composition of a metagenome gives insight to the inner workings of the environment. For example, it has been shown that a certain type of gut metagenome is linked to susceptibility to Chron's disease [28]. Some interesting questions are: which species live in the environment, how abundant is each species, which biochemical processes are active, and which sets of species are responsible for which biochemical process. There exists online servers to do this kind of analysis, such as the MG-Rast [30] and WebMGA [42] servers.

To get an adequate coverage of a metagenome, an enormous amout of data must be collected compared to traditional single genome sequencing. Therefore high-throughput sequencing is required, which with the present technology produces only short reads with relatively high error rates compared to traditional Sanger sequencing. (see Table 2.1).

Chapter 3

Preliminary data structures

This chapter describes the data structures and concepts needed in the subsequent chapters. First, we describe suffix trees and suffix arrays. These two structures are not used directly in the main work of the thesis, but the conceptual framework surrounding the structures is required. Next we describe how to index bit vectors and strings for rank queries, which are used together with the Burrows-Wheeler transform to build a succinct text index. Last, we briefly describe the Union-Find structure, which is needed in the metagenomic clustering application.

3.1 Model of computation

Our model computer is an abstract machine with U memory cells. Each memory cell contains $\lceil \log_2 U \rceil$ bits and can therefore store the address of any memory cell in binary. The number $\lceil \log_2 U \rceil$ is called the *memory word size* of the machine. We assume that the machine can perform basic arithmetic operations like addition, subtraction, multiplication, division and taking remainders, as well as bitwise boolean operations, in constant time for two memory words. The machine supports random access, i.e. the contents of a memory cell can be fetched in constant time given the address.

The time complexity of an algorithm is measured in terms of the number of memory accesses and the basic operations mentioned. Space complexity is measured not in terms of the number of memory cells used, but in terms of the number of *bits* used. For instance, storing a permutation of the numbers from 1 to n in our model takes $O(n \log n)$ bits of space, since representing each integer takes $O(\log n)$ bits. The numbers might not be aligned with memory words, but they can be easily retrieved by bit shifting and taking remainders.

This model approximates a real modern computer quite well, but it has its limitations. For example, our model does not have a memory cache. With a memory cache, sequential access is typically considerably faster than random access. However, this does not change the asymptotic time complexity of algorithms, because the cache only gives a constant time speedup to memory access.

3.2 Suffix trees

A suffix tree is one of the most fundamental data structures in string processing. It is often the case that even though an algorithm does not construct a suffix tree, it can be interpreted as traversing through an implicit suffix tree. As this is particularly the case with algorithms on the bidirectional Burrows-Wheeler index, which is the main tool in this thesis, we introduce suffix trees in a reasonable level of detail here. From here on we assume that the input string S is terminated with a unique character \$ that is lexicographically smaller than all other characters of S.

Definition 3.2.1. *Suffix tree.* A suffix tree for the string S is a rooted tree with edges labelled with characters such that for each substring α of S there is a unique path starting from the root such that the concatenation of the labels on the path is α.

For example, Figure 3.1 shows a suffix tree of the string "banana\$". The children of a node are ordered lexicographically from left to right. The terminator symbol \$ at the end of S ensures that there is a one-to-one correspondence from suffixes of S to the leaves of the tree. There exists a number of algorithms to build the suffix tree in $O(|S|)$ time [15]. The construction algorithms are complicated and irrelevant to the thesis, so we will not discuss them.

The number of nodes in a suffix tree can be quadratic in the length of S. In the worst case every character of S is distinct, and the suffix tree has $|S|(|S|+1)/2+1$ nodes, the sum of the lengths of all suffixes, plus one for the root. However there is a way to represent the tree in space linear in $|S|$ with no loss of information. The trick is to compress chains of nodes of degree 1 by replacing the whole path with just one edge, and labelling the edge with the concatenation of the original edge labels. Figure 3.2 shows the compressed version of the tree in Figure 3.1. The number of nodes in the tree is now linear in S. The reasoning is as follows. Let n be the number of leaves and m the number of internal nodes in the tree. The number of edges is at least $2m$, since every internal node has at least two children. On the other hand

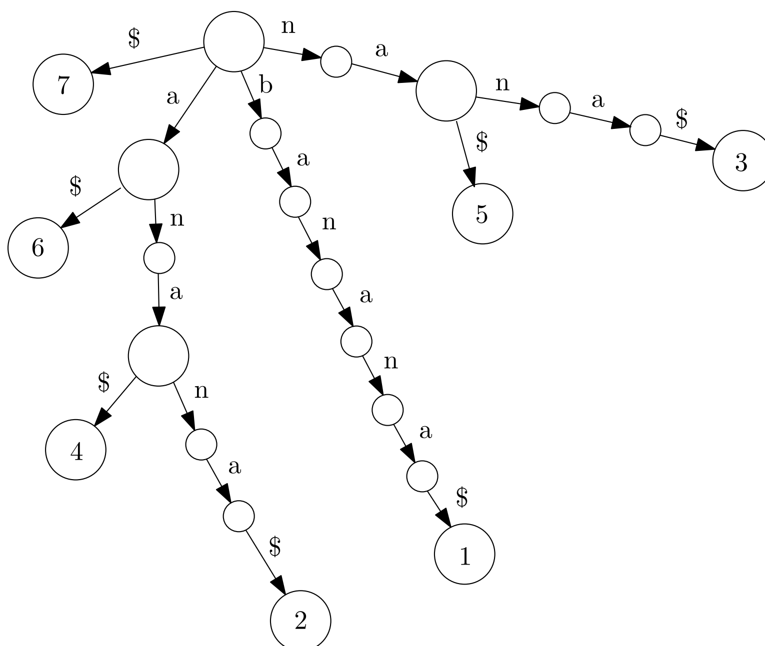

Figure 3.1: Suffix tree of "banana$"

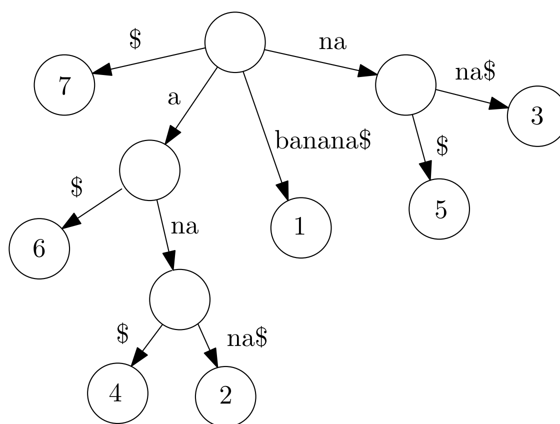

Figure 3.2: Compressed representation of the suffix tree of tree of "banana$"

for any tree it holds that the number of edges is $n+m-1$. Combining these two facts gives $n+m-1 \geq 2m$, from which we can solve $m \leq n-1$, which means $m \leq |S|-1$, since the leaves are in a one-to-one correspondence with the suffixes of S. The bound is tight, because $m = |S|-1$ when the alphabet of S has only a single character in addition to the terminating $.

Even though the number of nodes is now linear in S, the sum of the lengths of the path labels is still quadratic. However the labels can be represented with only two integers by storing the starting position of the label in S along with the length of the label. In summary, the suffix tree can be represented by keeping S and representing a linear number of nodes and edges, with the labels of each edge represented with two integers with values up to $|S|$. Under our model of computation, each such integer takes $O(\log n)$ bits of space, so the total space complexity is $O(n \log n)$.

Still, this representation supports only a very basic traversal of the tree. Depending on the application, a suffix tree may need to support more complicated operations. Sadakane et al. [32] define that *a full functionality suffix tree* supports the following operations.

- root(): returns the root node.
- isleaf(v): returns `yes` if v is a leaf, and `no` otherwise.
- child(v, c): returns the node w that is a child of v and label of the edge (v, w) begins with character c, or returns `null` if no such child exists.
- sibling(v): returns the next sibling of node v.
- parent(v): returns the parent node of node v.
- edge(v, d): returns the d-th character of the edge-label of an edge pointing to v.
- depth(v): returns the length of the concatenation of labels from the root to v.
- lca(v, w): returns the lowest common ancestor between nodes v and w.
- suffixlink(v): If the concatenation of the labels from the root to a node v is $x\alpha$, where x is a character and α is a (possibly empty) substring, returns a node u such that the concatenation of labels from the root to u is α. If v is the root, returns the root.

All operations can be implemented in constant time while retaining the space complexity of $O(n \log n)$, but the constant coefficients in the space complexity tend to be rather large. However, there are ways to reduce the space complexity from $O(n \log n)$ to $O(n \log \sigma)$, where σ is the size of the alphabet, using sophisticated data structures at the cost of a polylogarithmic factor in $|S|$ in the time complexities of the operations [32].

3.3 Suffix arrays

The *suffix array* of a string S is a data structure closely related to the suffix tree of S. It is defined as follows:

Definition 3.3.1. *Suffix array.* A suffix array, denoted SA, of a string S is an array of length $|S|$ such that $SA[i]$ is the starting position of the i-th suffix in lexicographic order among all suffixes of S.

For example, take the string "banana$". Table 3.1 shows the lexicographically sorted list of all suffixes of the string. The suffix array $(7, 6, 4, 2, 1, 5, 3)$ can be read from the starting positions of the suffixes in the second column.

Rank	Suffix	Position
1	$	7
2	a$	6
3	ana$	4
4	anana$	2
5	banana$	1
6	na$	5
7	nana$	3

Table 3.1: Suffixes of banana$

The suffix array can be built in linear time by building the suffix tree of S, and traversing the suffix tree depth first by visiting the children of a node in lexicographic order. This will visit the leaves in lexicographic order of the path labels i.e. the suffixes they represent. The starting position of the suffix of the i-th visited leaf gives the value of $SA[i]$. There also exist linear time algorithms for building the suffix array directly without building the suffix tree first [18, 21].

The suffix array is deeply related to the suffix tree data structure. For every node v in the suffix tree, there exists an interval in the suffix array such that the suffixes in the interval are the suffixes at the leaves of the subtree of v. For example, in Figure 3.1, the node that is connected to the root with the character `a` corresponds to the interval (1,3) in the suffix array in Table 3.1. In fact, if the suffix array is coupled with an array that stores the length of the longest common prefix of all lexicographically adjacent suffixes, then one can already simulate bottom-up traversal of the suffix tree.

3.4 Bit vectors with rank and select support

Let B be a bit vector of length n. Many string algorithms use *rank queries* on bit vectors as a basic building block for more complicated procedures. A rank query on a bit vector is defined as follows:

Definition 3.4.1. *Rank query on a bit vector.* The rank of an index i in a bit vector is the number of ones in the range of positions from 1 to i.

The inverse operation of rank is called *select*.

Definition 3.4.2. *Select query on a bit vector.* A select query takes an integer j, and returns the position of the j-th one in the vector from the beginning.

If space consumption is not an issue, the answer to all rank and select queries can be precomputed, as there are only n different meaningful queries for both query types. This easily gives a support for constant time rank queries with $O(n \log n)$ bits of space. However, it turns out that even sublinear space if sufficient to support constant time rank queries [17].

The idea is to reduce the problem into a smaller subproblem, until the subproblem can be solved in constant time. Remarkably, a constant number of reductions is sufficient. The target is to reduce the problem to computing a rank query for an array of length $\log(n)/2$. There are only $2^{\log(n)/2} = \sqrt{n}$ possible binary arrays of length $\log(n)/2$, and only $\log(n)/2$ different places to query inside such an array, so we can afford to store the answers to all possible rank queries for each array. The answers can be stored as $\log(n)$ bit binary numbers. In total, this lookup table takes $\sqrt{n}\frac{\log^2 n}{2}$ bits, which grows slower than linearly in n. Table 3.2 shows an example of this table for $\log(n)/2 = 3$.

Then, we will precompute some values to be able to reduce any rank query to exactly three table lookups. First, we divide the array into blocks of size $\log^2 n$. For each such block, we store the number of ones up to, but not including the first index of the block as a $\log n$ bit binary integer. There are $n/\log^2 n$ such blocks, so the table takes $n/\log n$ bits of space, which is less than linear in n. Then, we subdivide each block into miniblocks of size $\log(n)/2$. For each miniblock we store the number of ones from the start of the bigger block up to, but not including the first index of the miniblock. As the number of ones inside the bigger block is at most $\log^2 n$, these numbers take only $\log \log^2 n = 1 + \log \log n$ bits each to represent. There are $2n/\log(n)$ miniblocks in total, so storing this information takes $(1 + \log \log n) 2n/\log(n)$

bits, which is again less than linear in n. This term asymptotically dominates the space complexity, so the total space complexity is $O(n \log \log n / \log n)$.

For an example, take the following binary string:

$$1010010101101001110110100111011010010001101001010110100101011001.$$

The length of the string is 64 bits, so the sizes of the big blocks are 36 bits, and the sizes of the miniblocks are 3 bits. For each 3-bit string we compute the rank of all the three positions. Table 3.2 shows the precomputed values for all possible 3-bit strings. Figure 3.3 shows in red the precomputed cumulative ranks of both the large and the small blocks.

	1	2	3
000	0	0	0
001	0	0	1
010	0	1	1
011	0	1	2
100	1	1	1
101	1	1	2
110	1	2	2
111	1	2	3

Table 3.2: Precomputed values

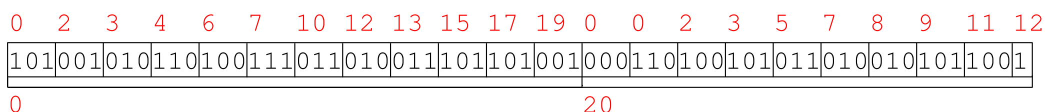

Figure 3.3: Precomputed cumulative ranks

Now we can answer any rank query with exactly three lookups to the precomputed tables – first look up the number of ones up to the superblock containing the query positions, then up to the miniblock containing the query index, and finally the count inside the miniblock up to the query position.

Given an index that supports rank queries on a bit vector, it is easy to count the number of ones in any continuous interval in the bit vector. The number of ones in the range of indices $[i, j]$ in vector B is equal to rank(B, j) $-$ rank$(B, i-1)$. The select query can be implemented in $O(\log n)$ time by binary searching the desired bit using the rank queries. There are also ways to implement the select query in constant time with sublinear space [7], but

they are not needed in the algorithms used in this thesis, and therefore not discussed.

3.5 Wavelet trees

There is a natural generalization of rank and select queries from bit vectors to strings.

Definition 3.5.1. *Rank on a string.* The rank of symbol c at position i in a string S is the number of occurrences of c in the prefix $S[1..i]$.

A wavelet tree is a data structure occupying $O(n \log \sigma)$ bits of memory that can answer rank queries on a string in time $O(\log \sigma)$. It is built on top of an efficient implementation for rank queries on bit vectors.

A wavelet tree is essentially a binary search tree in which every node v is associated with a subset of the alphabet $\mathcal{A}(v) \subseteq \Sigma$ such that if node u is an ancestor of v, then $\mathcal{A}(v) \subseteq \mathcal{A}(u)$. For a simpler explanation assume that σ is a power of two. The tree is then a complete binary tree of height $\log_2(\sigma)$ and the subalphabets associated to the nodes are defined recursively as follows:

- Suppose v is an internal node. Let left(v) and right(v) denote the left and right children of v, respectively. The subalphabet of v is $\mathcal{A}(\text{left}(v)) \cup \mathcal{A}(\text{right}(v))$.

- The subalphabets associated with the leaves are single characters, and the leaves are ordered lexicographically from left to right with respect their characters.

Each node v conceptually represents the subsequence S_v of S consisting of only the characters of S that are in $\mathcal{A}(v)$. For each node v, we build a bit vector B_v of length equal to $|S_v|$, where $B_v[i] = 0$ if $S_v[i]$ is in the first half of $\mathcal{A}(v)$, and 1 otherwise. The sum of the lengths of all the bit vectors is $n \log \sigma$. All bit vectors are indexed for rank queries.

The generalized rank query can be computed by traversing from the root of the wavelet root to the leaf corresponding to the desired character. Suppose we want to compute the rank of character c up to and including position i. Starting from the root, let v be the current node. While descending down the tree, we maintain the number k_v of positions j in the prefix $S[1..i]$ such that $S[j] \in \mathcal{A}(v)$.

Initially, $k_{root} = i$ since $\mathcal{A}(root)$ is the whole alphabet. The values $k_{\text{right}(v)}$ and $k_{\text{left}(v)}$ can be deduced from k_v by $k_{\text{right}(v)} = rank_{B_v}(c, k_v)$ and $k_{\text{left}(v)} = k_v - rank_{B_v}(c, k_v)$. We find the leaf u with $\mathcal{A}(u) = \{c\}$ by applying these

formulas recursively down the tree, and visiting only nodes whose alphabet contains c. At the leaf u we can deduce $rank_S(c, i) = k_u$. The depth of the tree is logarithmic in σ, and rank queries take constant time, so the time complexity is $O(\log \sigma)$.

In this work we use one additional operation on wavelet trees. The operation takes an index i and a character c, and returns the sum of ranks of lexicographically smaller characters in the prefix $S[1..i]$, i.e. the sum

$$\sum_{d \in \Sigma, d \leq c} rank(i, d) \tag{3.1}$$

This can be done by in time $O(\sigma \log \sigma)$ by directly evaluating Formula 3.1 using the wavelet tree. This can be improved to $O(\log \sigma)$ by counting all characters simultaneously with only one traversal. The key is that if the subalphabet of the current node v is completely disjoint or completely contained in the set $\{d \leq c \mid d \in \Sigma\}$, then we need not visit the descendants of the node. Fortunately, we only have to sum up at most one node at each level of the tree. We can thus decompose the sum in Formula 3.1 into $O(\log \sigma)$ parts such that we only have to traverse $O(\log \sigma)$ nodes (see Figure 3.4). For details, see Algorithm 1.

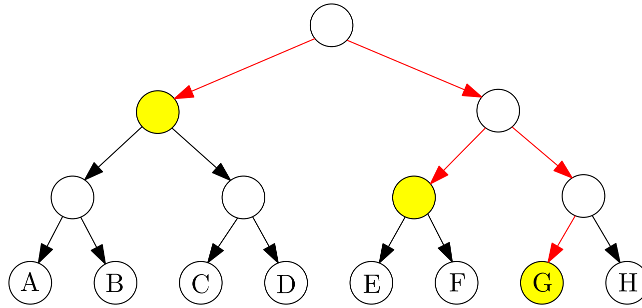

Figure 3.4: The sum of the number of characters lexicographically smaller or equal to G is the sum of the k_v values of the yellow nodes.

3.6 The Burrows-Wheeler transform

The Burrows-Wheeler transform is a permutation on a string invented by Michael Burrows and David Wheeler in 1994 [5]. It was originally used for text compression, because it tends to form long sequences of the same characters if the text is repetitive, which can be exploited e.g. by encoding

Algorithm 1 Evaluating Formula 3.1 for index i and character c.

1: **return** COUNT($root$, i, c, i)
2: **procedure** COUNT(node v, count k_v, character c, index i)
3: **if** $\max(\mathcal{A}(v)) \leq c$ **then return** k_v
4: **if** $\min(\mathcal{A}(v)) > c$ **then return** 0
5: $k_{\text{right}(v)} = rank_{B_v}(c, k_v)$
6: $k_{\text{left}(v)} = k_v - rank_{B_v}(c, k_v)$
7: **return** COUNT(left(v), $k_{\text{left}(v)}$, c, i) + COUNT(right(v), $k_{\text{right}(v)}$, c, i)
8: **end procedure**

long runs of characters by specifying the character and the length of the run. For convenience, we define the transform only for strings terminated with a unique character \$ that is lexicographically smaller than all characters in the input S.

Definition 3.6.1. *Burrows-Wheeler transform.* Let S be a \$-terminated string of length n, and SA be the suffix array of S. The Burrows-Wheeler transform BWT is a string of length n such that $BWT[i] = S[SA[i] - 1]$ if $SA[i] \neq 1$, and $S[n]$ otherwise.

Sorted suffixes	BWT
banana \$	a
banan a\$	n
ban ana\$	n
b anana\$	b
banana\$	\$
bana na\$	a
ba nana\$	a

Table 3.3: Lexicographically sorted suffixes of banana

To make indexing easier, we assume the convention that $S[0]$ means $S[n]$ and the substring $S[0..n]$ means $S[n..n]$. Table 3.3 shows an example of the Burrows-Wheeler transform for the string banana\$. The Burrows-Wheeler transform can be interpreted as the list of left extensions of all suffixes of S in lexicographical order. Lemma 1 describes a property that allows us to infer the lexicographical rank of the left extension of a suffix given the lexicographical rank of the suffix, which is the key property for the inversion of the BWT.

Lemma 1. *The number of suffixes that are lexicographically smaller than the left extension of* $S[i..n]$ *is equal to the number of positions* j *such that either* $BWT[j] <_{lex} BWT[i]$ *or* $(j < i$ *and* $BWT[j] = BWT[i])$*.*

Proof. If $BWT[j] <_{lex} BWT[i]$, then by definition $S[SA[j]-1] <_{lex} S[SA[i]-1]$, and therefore $S[SA[j]-1..n] <_{lex} BWT[SA[i]-1..n]$. On the other hand if $j < i$ and $BWT[j] = BWT[i]$, then the suffixes $S[SA[j]-1..n]$ and $S[SA[i]-1..n]$ start with the same character, but since by the definition of the suffix array it holds that $S[SA[j]..n] <_{lex} S[SA[i]..n]$, we have that $S[SA[j]-1..n] <_{lex} S[SA[i]-1..n]$. Similarly if the two conditions of the Lemma do not hold for a position j, then $S[SA[j]-1..n] >_{lex} BWT[SA[i]-1..n]$ □

3.6.1 Inverting the BWT

Lemma 1 is the foundation for an algorithm to invert the Burrows-Wheeler transform. The inversion is based on *backward steps* on the string.

Definition 3.6.2. *Backward step.* Given the lexicographical rank of a suffix $S[i..n]$, a backward step gives the lexicographical rank of the suffix $S[i-1..n]$.

The backward step can be implemented by counting the number of BWT positions that satisfy either of the conditions in Lemma 1. To do the computation efficiently, some preprocessing has to be done on the BWT. The count of positions that fulfil the first condition only depend of the character at the index i. Given that the alphabet size is typically small compared to the length of the string, the count of positions that fulfil the first condition can be precomputed in a table for all distinct characters. This array is often called the C-array in the literature.

Definition 3.6.3. *C-array.* The C-array is an array of length σ such that $C[i]$ is the number of characters in S that have lexicographical rank strictly less than i.

The count of positions that fulfil the second condition can be solved by a rank query (see Definition 3.5.1) on the BWT. The answers to the rank queries for all positions can be precomputed by one linear pass over the BWT, or by indexing the BWT as a wavelet tree, making the space consumption smaller, but increasing the query complexity from $O(1)$ to $O(\log \sigma)$. The original string can be recovered from the BWT by executing n backward steps starting from the first position of the BWT. The details are shown in Algorithm 2.

Algorithm 2 Inverting the Burrows-Wheeler transform

1: Precompute the C-array.
2: Build an index to answer rank queries on the BWT
3: $S \leftarrow$ empty string
4: $k \leftarrow$ the index of \$ in S.
5: **for** $i = 1 \rightarrow |S|$ **do**
6: $\quad S \leftarrow S \cdot BWT[k]$ // concatenation
7: $\quad k \leftarrow C[BWT[k]] + rank_{BWT}(k, BWT[k])$
8: **end for**
9: **return** $\overline{S}$.

3.6.2 Searching for patterns using the Burrows-Wheeler transform

Observe that the lexicographic ranks of the suffixes that are prefixed by α form a continuous interval. We call this interval the *lexicographic interval* or *lexicographic range* of α. The terms *suffix array interval* and *BWT interval* are also used in the literature. For example, the lexicographic range of the substring `ab` in the string `ababbababab$` is $(8, 11)$ (see Table 3.4).

Rank	Suffix	BWT
1	\$	b
2	ab\$	b
3	abab\$	b
4	ababab\$	b
5	ababbababab\$	\$
6	abbababab\$	b
7	b\$	a
8	bab\$	a
9	babab\$	a
10	bababab\$	b
11	babbababab\$	a
12	bbababab\$	a

Table 3.4: the lexicographic range of the substring `ba` in the string `ababbababab$`

Given an interval corresponding to a substring α, the BWT can be used to find the interval corresponding to the substring $c\alpha$ for any $c \in \Sigma$. This operation is called a left extension.

Definition 3.6.4. *Left extension.* A left extension takes an interval $[i, j]$ representing the substring α and a character c, and returns an interval representing the substring $c\alpha$, or an empty interval if no such substring occurs in S.

The left extension can be implemented using the backward step operation. Let $[i, j]$ be the interval of a substring α, and suppose we want to find the interval of $c\alpha$. The BWT tells us the character that precedes each of the suffixes in the lexicographic range $[i, j]$. The lexicographic ranks of suffixes that are prefixed by α and preceded by c are exactly the indices k in the BWT such that $i \leq k \leq j$ and $BWT[k] = c$. For example let $S =$ `bbbb$baaabaa`, $\alpha =$ `ba` and $c =$ `a`. From Table 3.4 we see that the BWT has the character `a` only at positions 8,9 and 11 in the lexicographic range (8,11) of `ba`. Executing backward steps for these three positions gives the positions 3, 4 and 5, which is the lexicographic range of `aba`.

However, it is not necessary to execute the backward step for all of the positions k which match the criteria $i \leq k \leq j$ and $BWT[k] = c$. As with all substrings, the suffixes that are prefixed by $c\alpha$ form a continuous lexicographic interval. It suffices to find the first and last index of this interval. By Lemma 1, this is the range $[C[c] + rank_{BWT}(i_c, c), C[c] + rank_{BWT}(j_c, c)]$, where i_c and j_c are the indices of the first and the last occurrences of c in the interval $[i, j]$. By definition of i_c and j_c we know that c does not appear in the BWT intervals $[i, i_c - 1]$ and $[j_c + 1, j]$, we can simplify the expression to $[C[c] + rank_{BWT}(i, c), C[c] + rank_{BWT}(j, c)]$.

We can now find the interval of any substring α of S step by step by starting from the empty string and applying the left extension operation for the characters of α from right to left. The length of the interval gives the number of occurrences of α, but does not give the positions of the occurrences yet, only the lexicographical ranks of the suffixes that have α as a prefix. The text positions of a suffix can be looked up in constant time from the suffix array of S. This way, we can find the text positions of all occurrences of α in time $O(|\alpha| \log \sigma + occ)$, where occ is the number of times α appears in S. The suffix array is often stored in some compressed form to save space [3]. The combination of a BWT indexed for rank queries, a C-array and a sampled suffix array is called an FM-index, and there are many variants.

3.6.3 The relationship with the suffix tree

Recall that a suffix link is a directed edge from the node corresponding to the substring $c\alpha$ to the node corresponding to the substring α. The suffix links induce a *suffix link tree*. When talking about the suffix link tree, the

directions of the edges are usually reverse so that the edges point away from the root of the tree. The edge in the inverse direction of a suffix link is called a *Weiner link*.

Definition 3.6.5. *Suffix link tree.* The suffix link tree of a string S is the tree induced by all Weiner links of the suffix tree of S.

The substring search algorithm can be interpreted as traversing the suffix link tree along the path of the query substring.

3.7 Union-find data structure

The union-find data structure, also known as the disjoint set data structure, is a structure that maintains sets of disjoint elements, and supports two operations:

- find(x): returns a handle to the set containing element x.
- union(s_1, s_2): merges the two sets s_1 and s_2, and returns a handle to the newly formed set.

A structure implementing these operations for the set of integers from 1 to n can be implemented in $O(n \log n)$ bits of space, such that the union-operation takes constant time, and the find-operation takes $O(\alpha(n))$ time on average, where α is the inverse Ackerman function. The data structure works by representing each set as a tree, where the root of the tree is the handle of the set. The union-operation is implemented by making the root of the larger tree point to the root of the smaller tree. This guarantees that all trees are of height $O(\log n)$. The find-operation is implemented by traversing the tree from the given element to the root. The elements on the way to the root can be directly attached to the root after this, which gives the $O(\alpha(n))$ average time complexity for the find operation. A more detailed exposition can be found in e.g. [8].

Chapter 4

The bidirectional Burrows-Wheeler index

The Burrows-Wheeler transform is asymmetric in the sense that it allows backward steps and left extensions, but not forward steps and right extensions. The bidirectional Burrows-Wheeler transform is a way of making the index symmetric such that these operations become possible. The idea dates back to 2009 in the literature [22]. The bidirectionality gives the index more expressive power and allows one to solve more complicated problems in succinct space, such as finding all maximal unique matches between two strings [4].

The bidirectional index consists of the regular BWT along with the BWT of the reverse string, denoted $\overline{BWT}$. The regular BWT can be interpreted as the list of all left extensions of lexicographically sorted suffixes, whereas the $\overline{BWT}$ can be interpreted as the list of all right extensions of colexicographically sorted prefixes. A string α is defined to be colexicographically smaller than a string β if and only if the reverse string of α is lexicographically smaller than the reverse string of β.

Algorithms based on the bidirectional Burrows-Wheeler index maintain two intervals instead of one. First, the lexicographic range of the suffixes that are prefixed by the current substring, and second, the colexicographic range of all prefixes that are suffixed by the current substring. We call the first the *lexicographic interval* or *lexicographic range*, and the second the *colexicographic interval* or *colexicographic range* of the substring. For example in the string abbabbaabbababbabaababa, the lexicographic range of the substring aab is (3,4) and the colexicographic range is (14,15). See Table 4.1.

Rank	*BWT*	Suffixes in lexicographic order	Prefixes in colexicographic order	$\overline{BWT}$
1	a	$	$	a
2	b	a$	$a	b
3	b	aababa$	$abbabbaabbababbabaa	b
4	b	aabbababbabaababa$	$abbabbaa	b
5	b	aba$	$abbabbaabbababbabaaba	b
6	b	abaababa$	$abbabbaabbababbabaababa	$
7	a	ababa$	$abbabbaabbaba	b
8	b	ababbabaababa$	$abbabbaabbababbaba	a
9	b	abbaabbababbabaababa$	$abba	b
10	b	abbabaababa$	$abbabbaabba	b
11	a	abbababbabaababa$	$abbabbaabbababba	b
12	$	abbabbaabbababbabaababa$	$abbabba	a
13	a	ba$	$ab	b
14	a	baababa$	$abbabbaabbababbabaab	a
15	b	baabbababbabaababa$	$abbabbaab	b
16	a	baba$	$abbabbaabbababbabaabab	a
17	b	babaababa$	$abbabbaabbabab	b
18	b	bababbabaababa$	$abbab	b
19	b	babbaabbababbabaababa$	$abbabbaabbab	a
20	a	babbabaababa$	$abbabbaabbababbab	a
21	a	bbaabbababbabaababa$	$abb	a
22	a	bbabaababa$	$abbabbaabb	a
23	a	bbababbabaababa$	$abbabbaabbababb	a
24	a	bbabbaabbababbabaababa$	$abbabb	a

Table 4.1: The bidirectional index for the string abbabbaabbababbabaababa.

4.1 Left- and right extensions

In Section 3.6.2, we saw how to compute the lexicographic interval of a substring $c\alpha$ given the lexicographic interval of the substring α. Now we show how to compute both the lexicographic and the colexicographic intervals of $c\alpha$ given the lexicographic and the colexicographic intervals of α.

Some new notation is in order to explain the procedure. We denote the start and end indices of the lexicographic interval of a substring α with $i^{\rightarrow}_{\alpha}$ and $j^{\rightarrow}_{\alpha}$, respectively, and the start and end points of the colexicographic interval of the same substring with $i^{\leftarrow}_{\alpha}$ and $j^{\leftarrow}_{\alpha}$, respectively.

Let the current substring be α, and let us extend to the left with the character c. First we find the updated lexicographic interval $[i^{\rightarrow}_{c\alpha}, j^{\rightarrow}_{c\alpha}]$ using the method described in Chapter 3 section 3.6.2. Next we find the colexicographic interval $[i^{\leftarrow}_{c\alpha}, j^{\leftarrow}_{c\alpha}]$. The start of the interval is computed by counting the number of prefixes of S that are suffixed by a substring β such that $\beta <_{colex} c\alpha$. Every prefix with colexicographc rank strictly less than $i^{\leftarrow}_{\alpha}$ fulfills this criterion, and every prefix with colexicographic rank strictly greater

than $j^{\leftarrow}_{\alpha}$ does not. Only the prefixes with colexicographic rank in the interval $[i^{\leftarrow}_{\alpha}, j^{\leftarrow}_{\alpha}]$ remain to be considered. For these, we want to count the number of prefixes that are suffixed by $d\alpha$ for some $d < c$. We simply count the number of positions in the forward BWT in the interval $[i^{\rightarrow}_{\alpha}, j^{\rightarrow}_{\alpha}]$ that contain a character smaller than c. These are precisely the desired occurrences of α that are preceded by a character smaller than d. This can be done in time $O(\log \sigma)$ [1] using the method described section 3.5. The end point $j^{\leftarrow}_{c\alpha}$ can be easily deduced from the fact the the length of the colexicographic interval must be the same as the length of the lexicographic interval. A summary of the left extension procedure is shown in Algorithm 3.

For example, let us search for the string aab from the string S = ababbaabbababbabaababa. Table 4.1 shows the BWT, the reverse BWT, the lexicographically sorted list of suffixes, and the colexicographically sorted list of prefixes. Initially we set $i^{\rightarrow} = i^{\leftarrow} = 1$ and $j^{\rightarrow} = j^{\leftarrow} = |S|$. By repeatedly applying Algorithm 4 we obtain the sequence of interval pairs:

$$
\begin{aligned}
(i^{\rightarrow}, j^{\rightarrow}), (i^{\leftarrow}, j^{\leftarrow}) &= (1, 24), (1, 24) \\
(i^{\rightarrow}_{b}, j^{\rightarrow}_{b}), (i^{\leftarrow}_{b}, j^{\leftarrow}_{b}) &= (13, 24), (13, 24) \\
(i^{\rightarrow}_{ab}, j^{\rightarrow}_{ab}), (i^{\leftarrow}_{ab}, j^{\leftarrow}_{ab}) &= (5, 12), (13, 20) \\
(i^{\rightarrow}_{aab}, j^{\rightarrow}_{aab}), (i^{\leftarrow}_{aab}, j^{\leftarrow}_{aab}) &= (3, 4), (14, 15)
\end{aligned}
$$

The algorithm for a right extension is similar, but with the roles of BWT and reverse BWT switched. The procedure is a straightforward modification of Algorithm 3, and it is shown for completeness in Algorithm 4. The C-array that appears in the algorithms is the same as in Definition 3.6.3.

Algorithm 3 Left extension on the bidirectional BWT

1: **procedure** ExtendLeft(intervals $[i^{\rightarrow}_{\alpha}, j^{\rightarrow}_{\alpha}]$ and $[i^{\leftarrow}_{\alpha}, j^{\leftarrow}_{\alpha}]$, character c)
2: $\quad i^{\rightarrow}_{c\alpha} := C[c] + rank_{BWT}(i^{\rightarrow}_{\alpha}, c)$
3: $\quad j^{\rightarrow}_{c\alpha} := C[c] + rank_{BWT}(j^{\rightarrow}_{\alpha}, c)$
4: $\quad i^{\leftarrow}_{c\alpha} := i^{\leftarrow}_{c\alpha} + \sum_{d \in \Sigma, d < c} (rank_{BWT}(j^{\rightarrow}_{\alpha}, d) - rank_{BWT}(i^{\rightarrow}_{\alpha} - 1, d))$
5: $\quad j^{\leftarrow}_{c\alpha} := i^{\leftarrow}_{c\alpha} + (j^{\rightarrow}_{c\alpha} - i^{\rightarrow}_{c\alpha})$
6: $\quad$ **return** $[i^{\rightarrow}_{c\alpha}, j^{\rightarrow}_{c\alpha}]$, $[i^{\leftarrow}_{c\alpha}, j^{\leftarrow}_{c\alpha}]$
7: **end procedure**

[1] The time complexity can be further improved to constant time by using monotone minimal hash functions [4]. The techniques involved are rather complex, and we will not discuss them here.

Algorithm 4 Right extension on the bidirectional BWT

1: **procedure** EXTENDRIGHT(intervals $[i^{\rightarrow}_{\alpha}, j^{\rightarrow}_{\alpha}]$ and $[i^{\leftarrow}_{\alpha}, j^{\leftarrow}_{\alpha}]$, character c)
2: $\quad i^{\leftarrow}_{\alpha c} := C[c] + rank_{\overline{BWT}}(i^{\leftarrow}_{\alpha}, c)$
3: $\quad j^{\leftarrow}_{\alpha c} := C[c] + rank_{\overline{BWT}}(j^{\leftarrow}_{\alpha}, c)$
4: $\quad i^{\rightarrow}_{\alpha c} := i^{\rightarrow}_{\alpha c} + \sum_{d \in \Sigma,\, d<c} (rank_{\overline{BWT}}(j^{\leftarrow}_{\alpha}, d) - rank_{\overline{BWT}}(i^{\leftarrow}_{\alpha} - 1, d))$
5: $\quad j^{\rightarrow}_{\alpha c} := i^{\rightarrow}_{\alpha c} + (j^{\leftarrow}_{\alpha c} - i^{\leftarrow}_{\alpha c})$
6: $\quad$ **return** $[i^{\rightarrow}_{\alpha c}, j^{\rightarrow}_{\alpha c}]$, $[i^{\leftarrow}_{\alpha c}, j^{\leftarrow}_{\alpha c}]$
7: **end procedure**

4.2 Left- and right maximality

We will now introduce the concepts of left- and right maximal substrings, which will be central in the applications of the bidirectional index.

Definition 4.2.1. *Right-maximal substring.* A substring α of a string S is right-maximal if and only if there exist at least two distinct characters c and d such that both αc and αd are substrings of S.

Definition 4.2.2. *Left-maximal substring.* A substring α of a string S is left-maximal if and only if there exist at least two distinct characters c and d such that both $c\alpha$ and $d\alpha$ are substrings of S.

Definition 4.2.3. *Maximal substring.* A substring α is maximal if it is both left- and right maximal.

In other words, a substring α is right-maximal if and only if the suffix tree of S branches at the locus of α, and the substring α is left-maximal if and only if the suffix *link* tree branches at the locus of α.

Checking both maximality conditions is easy given the bidirectional index. The substring α is left-maximal if and only if the lexicographic interval of α in the forward BWT contains two or more distinct characters, and symmetrically α is right-maximal if and only if the colexicographic interval in the reverse BWT contains two or more distinct characters. This can be computed with $O(\sigma)$ rank queries on the interval of the corresponding BWT, giving a total time complexity of $O(\sigma \log \sigma)$.

There is also a simple data structure to improve this to constant time by storing a bit vector B of length equal to the length of the BWT. For example, to implement left maximality checks define $B[i] = 1$ if and only if $i = 1$ or the character at position i in the BWT differs from the character at position $i-1$. The substring α is left-maximal if and only if the interval $B[i^{\rightarrow}_{\alpha}+1..j^{\rightarrow}_{\alpha}]$ contains at least one bit set to 1. This can be checked in constant time by

indexing B for constant time rank queries. Symmetrically, the same trick works in the other direction.

4.3 Iterating all right-maximal nodes of the suffix tree

Algorithms that use the suffix tree often do not use the full functionality of the suffix tree. For example, to find the lexicographic intervals of all repeating substrings of length k, one traverses the suffix tree and reports the current lexicographic interval when string depth k in the tree is reached. The only information needed is the depth and the lexicographic range of the current node. The parent-child relationships, or in other words the *topology* of the suffix tree is irrelevant. In these applications, the bidirectional index can often be used to implement the same algorithm. In this section we show how the bidirectional index can iterate the lexicographic ranges of all right-maximal suffix tree nodes efficiently.

Each substring α of S corresponds to a node in the suffix tree of S and to a node in the suffix link tree of S. The corresponding node in the suffix tree is the node such that the concatenation of the edge labels from the root down to the node is α, and the corresponding node in the suffix link tree is a node such that the concatenation of the path labels from the root is the reverse string of α.

Lemma 2. *If the label of a node in the suffix link tree is not right-maximal, then none of its children are right-maximal either.*

Proof. Suppose the substring α is non-right-maximal and there exists a right maximal child $e\alpha$ for some extension e in the suffix link tree. If this was the case, then there would exist substrings $e\alpha c$ and $e\alpha d$, for distinct $c, d \in \Sigma$ by the right-maximality of $e\alpha$, but this also implies that substrings αc and αd exist as they are part of $e\alpha c$ and $e\alpha d$ respectively, which contradicts the assumption that α was not right-maximal. □

By induction Lemma 2 implies that if a node in the suffix link tree is non right-maximal, then all the nodes in its subtree are non right-maximal. On the other hand, for every right-maximal substring, there is a path in the suffix link tree from the root to a node corresponding to that substring such that every node on the way is right-maximal. This means that to iterate every right-maximal substring, we only need to iterate the right-maximal nodes of the suffix link tree, as the nodes of a suffix tree of S are in one-to-to correspondence with the right-maximal substrings of S. There are at most

$|S|$ right-maximal nodes, so the iteration can be done in linear time in $|S|$. Algorithm 5 shows the iteration loop.

Algorithm 5 Iterating the lexicographic intervals of all right-maximal substrings using the bidirectional index

1: stack $\leftarrow$ Empty iteration stack
2: push $([1,n],[1,n])$ to stack
3: **while** stack is not empty **do**
4: $([i^{\rightarrow}_{\alpha}, j^{\rightarrow}_{\alpha}],[i^{\leftarrow}_{\alpha}, j^{\leftarrow}_{\alpha}])$:= top of the stack
5: **for** all $c \in \Sigma$ **do**
6: $([i^{\rightarrow}_{c\alpha}, j^{\rightarrow}_{c\alpha}],[i^{\leftarrow}_{c\alpha}, j^{\leftarrow}_{c\alpha}])$:= ExtendLeft$(([i^{\rightarrow}_{\alpha}, j^{\rightarrow}_{\alpha}],[i^{\leftarrow}_{\alpha}, j^{\leftarrow}_{\alpha}]), c)$
7: **if** $([i^{\rightarrow}_{c\alpha}, j^{\rightarrow}_{c\alpha}],[i^{\leftarrow}_{c\alpha}, j^{\leftarrow}_{c\alpha}])$ is right-maximal **then**
8: push $([i^{\rightarrow}_{c\alpha}, j^{\rightarrow}_{c\alpha}],[i^{\leftarrow}_{c\alpha}, j^{\leftarrow}_{c\alpha}])$ to stack
9: **end if**
10: **end for**
11: **end while**

4.4 Iterating all nodes of the suffix tree using only the forward BWT

Algorithm 5 does not use the full functionality of the bidirectional index since it never uses the right extension operation. In fact it turns out that it is possible to iterate all suffix link tree nodes with just the forward BWT [3]. If this is all that is needed, then the reverse BWT is unnecessary.

The trick is to store for all substrings α in the iteration stack of Algorithm 5 the intervals of right extensions αc for all $c \in \Sigma$. This can be done without having the reverse BWT because if we know all the intervals αc for a node corresponding to α in the suffix tree, then we can compute for any left extension $d\alpha$ all intervals of right extensions $d\alpha c$ for $c \in \Sigma$ by simply left extending the intervals αc. This information is enough to decide whether a substring α is right-maximal. The substring α is right-maximal if and only if at least two of the intervals αc are non-empty.

4.5 Applications of the bidirectional index in bioinformatics

A prominent feature of most DNA shotgun sequencing technologies is that the reads come from both of the complementary strands, and the sequencer

can not tell from which strand each read was sampled from. Since repeating substrings are rare in bacterial genomes and metagenomes, if one finds the reverse complement (see Definition 2.1.1) of a long susbtring in another read, then most likely both reads have been sampled from the same part of the same genome, just from different strands of the DNA. Identifying pairs of reads where this happens is useful in bioinformatics. The bidirectional index is a clean solution to the problem

4.5.1 Iterating all reverse complement right-maximal substrings

From here on we denote the reverse complement of a string α with $\widetilde{\alpha}$. We call a substring reverse complement right-maximal (RC right-maximal for short) if there are two distinct characters $c, d \in \Sigma$ such that αc or $\widetilde{\alpha c}$ is a substring of S, and αd or $\widetilde{\alpha d}$ is a substring of S. Algorithm 6 describes how to iterate all RC right-maximal substrings of a string in a linear number of steps in $|S|$ and space $O(|S| \log \sigma)$. To our knowledge, no such algorithm has been described in the literature before.

The idea is to maintain two synchronized iterators at the same time. One iterates the suffix link tree of S, and the other mirrors all movements to the suffix link tree of $\widetilde{S}$. Each node in the iteration stack is now represented by a total of four intervals describing the lexicographic and colexicographic ranges of both the current string and the reverse complement. The algorithm emulates traversal of the suffix link tree of the read set combined with the reverse complement of the read set.

Whenever the main iterator extends the current string α to the left with a character c, the mirror iterator extends $\widetilde{\alpha}$ to the right with the complement character of c. If the resulting substring $c\alpha$ is RC right-maximal, a new stack frame consisting of the 4 intervals $[i_{\overrightarrow{c\alpha}}, j_{\overrightarrow{c\alpha}}]$, $[i_{\overleftarrow{c\alpha}}, j_{\overleftarrow{c\alpha}}]$, $[i_{\overrightarrow{\widetilde{c\alpha}}}, j_{\overrightarrow{\widetilde{c\alpha}}}]$, $[i_{\overleftarrow{\widetilde{c\alpha}}}, j_{\overleftarrow{\widetilde{c\alpha}}}]$ is pushed to the iteration stack.

The algorithm decides whether the substring $c\alpha$ is RC right-maximal by using the intervals $[i_{\overleftarrow{c\alpha}}, j_{\overleftarrow{c\alpha}}]$ and $[i_{\overrightarrow{\widetilde{c\alpha}}}, j_{\overrightarrow{\widetilde{c\alpha}}}]$. The interval $[i_{\overleftarrow{c\alpha}}, j_{\overleftarrow{c\alpha}}]$ in the reverse BWT lists all right-extensions of $c\alpha$ and the interval $[i_{\overrightarrow{\widetilde{c\alpha}}}, j_{\overrightarrow{\widetilde{c\alpha}}}]$ in the forward BWT lists all left-extensions of $\widetilde{c\alpha}$. Let $\Sigma_1 = \{c \mid c \in \overline{BWT}[i_{\overleftarrow{c\alpha}}..j_{\overleftarrow{c\alpha}}]\}$ be the set of distinct characters in $\overline{BWT}[i_{\overleftarrow{c\alpha}}..j_{\overleftarrow{c\alpha}}]$ and $\Sigma_2 = \{\widetilde{c} \mid c \in BWT[i_{\overrightarrow{\widetilde{c\alpha}}}..j_{\overrightarrow{\widetilde{c\alpha}}}]\}$ be the set of reverse complements of the distinct characters in $BWT[i_{\overrightarrow{\widetilde{c\alpha}}}, j_{\overrightarrow{\widetilde{c\alpha}}}]$. The string $c\alpha$ is RC right-maximal if and only if $|\Sigma_1 \cup \Sigma_2| \geq 2$. This condition can be checked by checking the existence of each character $c \in \Sigma$ in both intervals $\overline{BWT}[i_{\overleftarrow{c\alpha}}..j_{\overleftarrow{c\alpha}}]$ and $BWT[i_{\overrightarrow{\widetilde{c\alpha}}}..j_{\overrightarrow{\widetilde{c\alpha}}}]$ with 2σ rank queries on the wavelet trees of BWT and $\overline{BWT}$.

Algorithm 6 Iterating all RC right-maximal substrings

1: stack $\leftarrow$ Empty iteration stack
2: push $(([1,n],[1,n]),([1,n],[1,n]))$ to stack
3: **while** stack is not empty **do**
4: $[i_\alpha^{\rightarrow}, j_\alpha^{\rightarrow}], [i_\alpha^{\leftarrow}, j_\alpha^{\leftarrow}], [i_{\widetilde{\alpha}}^{\rightarrow}, j_{\widetilde{\alpha}}^{\rightarrow}], [i_{\widetilde{\alpha}}^{\leftarrow}, j_{\widetilde{\alpha}}^{\leftarrow}]$:= top of the stack
5: **for** $c \in \Sigma$ **do**
6: $[i_{c\alpha}^{\rightarrow}, j_{c\alpha}^{\rightarrow}], [i_{c\alpha}^{\leftarrow}, j_{c\alpha}^{\leftarrow}]$, := ExtendLeft($[i_\alpha^{\rightarrow}, j_\alpha^{\rightarrow}], [i_\alpha^{\leftarrow}, j_\alpha^{\leftarrow}], c)$
7: $[i_{\widetilde{c\alpha}}^{\rightarrow}, j_{\widetilde{c\alpha}}^{\rightarrow}], [i_{\widetilde{c\alpha}}^{\leftarrow}, j_{\widetilde{c\alpha}}^{\leftarrow}]$:= ExtendRight($[i_{\widetilde{\alpha}}^{\rightarrow}, j_{\widetilde{\alpha}}^{\rightarrow}], [i_{\widetilde{\alpha}}^{\leftarrow}, j_{\widetilde{\alpha}}^{\leftarrow}], \widetilde{c})$
8: **if** $[i_{c\alpha}^{\rightarrow}, j_{c\alpha}^{\rightarrow}], [i_{c\alpha}^{\leftarrow}, j_{c\alpha}^{\leftarrow}], [i_{\widetilde{c\alpha}}^{\rightarrow}, j_{\widetilde{c\alpha}}^{\rightarrow}], [i_{\widetilde{c\alpha}}^{\leftarrow}, j_{\widetilde{c\alpha}}^{\leftarrow}]$ is RC right-maximal **then**
9: push $\left([i_{c\alpha}^{\rightarrow}, j_{c\alpha}^{\rightarrow}], [i_{c\alpha}^{\leftarrow}, j_{c\alpha}^{\leftarrow}], [i_{\widetilde{c\alpha}}^{\rightarrow}, j_{\widetilde{c\alpha}}^{\rightarrow}], [i_{\widetilde{c\alpha}}^{\leftarrow}, j_{\widetilde{c\alpha}}^{\leftarrow}]\right)$ to stack
10: **end if**
11: **end for**
12: **end while**

4.5.2 Finding the intervals of k-mers

A common task in bioinformatics is to find all distinct k-mers (substrings of length k) of a string S. We define an equivalence relation on the suffixes such that two suffixes s_i and s_j are equivalent if and only if $|s_i| \geq k$, $|s_j| \geq k$ and $s_i[1..k] = s_j[1..k]$. This equivalence relation partitions the suffixes based on the first k characters. The set of suffixes in a single equivalence class are lexicographically adjacent, which means we can represent the class as a lexicographic interval. The whole partition can be conveniently represented as a single bit vector B of length $|S|$ such that $B[i] = 1$ if and only if the suffix with rank i is the lexicographically least of its equivalence class. In other words, the start of the interval of each class is marked with a 1.

The goal is to mark the intervals of all nodes which are at depth k or greater, and their parent is at depth $k-1$ or less. Visually, we are looking to find the "frontier" of suffix tree nodes that are at depth greater or equal to k.

The algorithm makes use of both the left and the right extension procedures. The left extension is used as a black box iterator to enumerate the lexicographic intervals of all right-maximal suffix tree nodes. The right extension is used in each node to find the lexicographic intervals of all children of the node in the suffix tree. If the depth of the current node is less than k, the starting points of the lexicographic interval of each of its children in the suffix tree are marked in the bit vector B. The algorithm works in time $O(n\sigma \log \sigma)$.

Algorithm 7 Marking all k-mers

1: stack $\leftarrow$ Empty iteration stack
2: push $([1,n],[1,n],0)$ to stack
3: $B :=$ bit vector of length n initialized to zeroes
4: **while** stack is not empty **do**
5: $([i^{\rightarrow}_{\alpha}, j^{\rightarrow}_{\alpha}], [i^{\leftarrow}_{\alpha}, j^{\leftarrow}_{\alpha}], \text{depth}) :=$ top of the stack
6: **if** depth $< k$ **then**
7: **for** all $c \in \Sigma$ **do**
8: $([i^{\rightarrow}_{\alpha c}, j^{\rightarrow}_{\alpha c}], [i^{\leftarrow}_{\alpha c}, j^{\leftarrow}_{\alpha c}]) :=$ ExtendRight$(([i^{\rightarrow}_{\alpha}, j^{\rightarrow}_{\alpha}], [i^{\leftarrow}_{\alpha}, j^{\leftarrow}_{\alpha}]), c)$
9: **if** $[i^{\rightarrow}_{\alpha c}, j^{\rightarrow}_{\alpha c}]$ is valid **then**
10: $B[i^{\rightarrow}_{\alpha c}] = 1$
11: **end if**
12: **end for**
13: **end if**
14: **for** all $c \in \Sigma$ **do**
15: $([i^{\rightarrow}_{c\alpha}, j^{\rightarrow}_{c\alpha}], [i^{\leftarrow}_{c\alpha}, j^{\leftarrow}_{c\alpha}]) :=$ ExtendLeft$(([i^{\rightarrow}_{\alpha}, j^{\rightarrow}_{\alpha}], [i^{\leftarrow}_{\alpha}, j^{\leftarrow}_{\alpha}]), c)$
16: **if** $([i^{\rightarrow}_{c\alpha}, j^{\rightarrow}_{c\alpha}], [i^{\leftarrow}_{c\alpha}, j^{\leftarrow}_{c\alpha}])$ is right-maximal **then**
17: push $([i^{\rightarrow}_{c\alpha}, j^{\rightarrow}_{c\alpha}], [i^{\leftarrow}_{c\alpha}, j^{\leftarrow}_{c\alpha}], \text{depth}+1)$ to stack
18: **end if**
19: **end for**
20: **end while**

4.5.3 Marking the intervals of k-submaximal repeats

k-submaximal repeats are a class of repeats with applications in metagenomic clustering [27]. The definition is as follows:

Definition 4.5.1. *k-submaximal repeat.* A substring α is a k-submaximal repeat if and only if α is a maximal repeat of length at least k and there does not exist a maximal repeat β such that $|\beta| \geq k$ and β is a substring of α.

The lexicographic intervals of all k-submaximal repeats can be found with two passes over the suffix link tree. First, we mark all right-maximal k-mers in a bit vector B using the iteration loop in Algorithm 5 by marking the start and the end point of the current interval with a 1 every time we are at depth k. On the second pass, we try to extend each right-maximal k-mer to the left such that it also becomes left-maximal, stopping if the current interval is a *subinterval* of the interval of a right-maximal k-mer. Every interval that was successfully extended to left maximality is an interval of a k-submaximal repeat.

Suppose we are extending the k-mer α to the left. Let M be the shortest maximal repeat such that α is the rightmost k-mer of M. By extending

α to the left we enumerate all lexicographic intervals of every suffix of M of length at least k. For each of these suffixes $\beta\alpha$ we check whether the lexicographic interval of $\beta\alpha$ is a subinterval of some right-maximal k-mer. This is equivalent to checking whether the parity of $i^{\rightarrow}_{\beta\alpha}$ in B is odd. If this happens, then the k-mer is a prefix of $\beta\alpha$, and thus M is not k-submaximal. If this does not happen for any suffix $\beta\alpha$ of M, then M is k-submaximal and we can report the interval of M.

Algorithm 8 Finding all k-submaximal repeats

```
1: B := bit vector of length n with right-maximal k-mers marked
2: stack := Empty iteration stack
3: push ([1, n], [1, n], 0) to stack
4: while stack is not empty do
5:     [i→_α, j→_α], [i←_α, j←_α], depth := top of the stack
6:     if depth ≥ k and LeftMaximal( [i→_α, j→_α]) then
7:         Report [i→_α, j→_α]
8:         continue
9:     end if
10:    for all c ∈ Σ do
11:        ([i→_cα, j→_cα], [i←_cα, j←_cα]) := ExtendLeft(([i→_α, j→_α], [i←_α, j←_α]), c)
12:        if ([i→_cα, j→_cα], [i←_cα, j←_cα]) is right-maximal then
13:            if depth + 1 ≤ k or rank_B(i→_cα) is even then
14:                push ([i→_cα, j→_cα], [i←_cα, j←_cα], depth + 1) to stack
15:            end if
16:        end if
17:    end for
18: end while
```

4.5.4 Locating suffixes by sampling the suffix array

All the algorithms described in this chapter report only the lexicographic intervals, not actual text positions of the desired suffixes. These can be easily converted to text positions by looking the positions up from the suffix array of S. However, the suffix array takes $O(n \log n)$ bits of space, which can be an order of magnitude greater than the $O(n \log \sigma)$ space taken up by the bidirectional index. Fortunately, there is a way to convert lexicographic ranks into text positions without storing the whole suffix array.

This is done as a separate step after the iteration of the suffix link tree. We modify suffix link tree iteration routine to mark all the lexicographic ranks of the interesting suffixes in a bit vector of length $|S|$. After this, we execute $|S|$

backward steps on the forward BWT starting from the empty suffix. In effect we walk through S backwards while maintaining the lexicographic rank of the suffix starting from the current position. This gives us the values $ISA[i]$ for $i = |S| \dots 1$, where ISA is the inverse suffix array. If $ISA[i] = k$, we can deduce that $SA[k] = i$. Walking the string backward in this way generates us every value of the suffix array, albeit in a unpredictable order. Every time we get a new suffix array sample, we check if the position is marked in the bit vector we built earlier, and report the current text position if this is the case.

4.5.5 Generalization to multiple strings

There is a standard trick to generalize all the algorithms described in this section from a single string S to a set of strings $\{S_1, \dots, S_m\}$. The trick is to concatenate all of the strings $S_1, \dots, S_m$, putting a separator character that does not appear in the alphabet of any of strings between the strings. In practice this can usually be the same "dollar" symbol that is used to mark the end of the string in a regular BWT. Then we modify the suffix link tree iteration algorithms such that the separator symbol is not considered a valid left- or right extension.

Another way to do the concatenation is to take two symbols that are not part of the alphabet of any of the strings, and separate the strings with a unique binary string formed out of the two external characters. This increases the length of the BWTs by $m \log m$, where m is the number of strings. The advantage is that now Algorithm 5 can be used without modifications, because the separators guarantee that there is no right-maximal substring that spans the separator into two strings. The disadvantage is that this requires some extra effort depending on the application to avoid reporting suffixes which start inside a separator sequence.

Chapter 5

Clustering metagenomic read sets

Finding groups of reads in a metagenomic sample which originate from the same species is called *metagenome clustering*. The task is closely related to metagenome assembly, which is the problem of reconstructing the prevalent genomes found in a metagenomic sample. Metagenomic clustering is a relaxation of the problem of assembly, where we are only interested if two reads originate from the same genome, but not in the order the reads appear in the genome.

Clustering algorithms are commonly classified into *supervised* and *unsupervised* algorithms. Supervised algorithms use some reference database describing the ground truth to aid in clustering, whereas unsupervised methods run without a need for external data. The algorithm described in this thesis is unsupervised. While unsupervised algorithms tend to be less accurate than supervised algorithms, a major problem with supervised methods is that the reference databases are incomplete, because most of the bacterial species are difficult to culture. [2] Therefore, unsupervised *de novo* methods which do not rely on outside information are needed. Unsupervised methods can work on datasets which are wildly different from the reference databases, and can discover new species.

Clustering of metagenomic reads has numerous applications. For example, the clusters can be used to estimate the number of species in a sample. The advantage of using clustering over assembly is that clustering is cheaper to run, because we do not have to spend time aligning and resolving conflicts. Assembly can to be slow and demand lots of memory [46].

The problem of metagenome assembly can be solved by reducing the problem of assembling a metagenome to the problem of assembling the individual clusters produced by a clustering algorithm. Single genome assembly has already been studied extensively in the literature, and there exists a multitude of tools for it [46].

Unsupervised clustering could also be helpful for annotating unknown reads. There exist regions in genomes which are highly preserved, but which differ between species. These sparse regions are called *barcode sequences*. If a cluster contains even a single barcode sequence, then the whole cluster can be annotated accordingly [35].

5.1 Challenges

The problem of clustering by species is ill-defined, because as we saw in Chapter 2, there is no exact definition of what constitutes a species based on DNA alone. Therefore any clustering algorithm can only approximate the official consensus taxonomy defined by biologists. Acknowledging this, there are good heuristics for the problem based on long shared k-mers and short k-mer distributions.

Imperfections of sequencing technologies create another set of problems. Various types of sequencing errors introduce noise into the reads. To deal with this, we detect and disregard reads which seem to contain many errors. Present technologies can also produce *chimeric reads*, which are formed by the concatenation of reads from multiple species. However, this is a relatively rare phenomenon [cite], so we make the simplifying assumption that a read can only originate from one species.

5.2 Existing tools

Numerous tools for the clustering problem have been proposed in the literature. We will present a brief survey of the most important unsupervised tools in the literature.

TETRA [35, 36] (2004) is one of the first algorithms dedicated to clustering metagenomes in the literature. The algorithm computes the distribution of tetranucliotides, i.e. 4-mers, for all input sequences. A k-mer is considered equal to its reverse complement. The choice of using precisely 4-mers instead of any other length k-mers is rationalized by arguing that shorter k-mers do not provide a strong enough phylogenetic signal, while the discriminatory power of distributions longer of k-mers is unclear. From the 4-mer vectors, the tool computes a vector of z-values for each input sequence measuring the statistical under- or over representation of each 4-mer in each input sequence. All z-value vectors are then compared pairwise using the Pearson correlation coefficient as the distance measure.

The method is only suitable for long sequences in the range of 40 thousand base pairs, because the phylogenetic signal carried by the 4-mer distribution is faint. It is also not suitable for large sets of input sequences, because it compares all pair of sequences pairwise, implying a quadratic running time with respect to the number of sequences.

CompostBin [6] (2008) is based on the euclidean distance on a low-dimensional projection of the space of k-mer distributions. This tool first finds the 6-mer distribution vector of every read in the read set. Since the number of dimensions 4^6 is very large, the vectors are projected into a three dimensional space using an abundance-weighted variant of principal component analysis. A nearest neighbor graph is then constructed such that every read has an edge to its six nearest neighbors in the three-dimensional feature space. The edges are weighted inverse exponentially with respect to the euclidean distance of the pair of feature vectors. Finally the clusters are iteratively cut into two pieces trying to maximize the weight of the edges crossing the cut, and minimize the sum of weights within each half of the cut, until the desired number of clusters is reached. Like all k-mer composition vector based methods, this tool is only shown to work for long reads of length 1000 base pairs.

LikelyBin [20] (2009) tries to simulaneously estimate the underlying k-mer distributions and the mixing ratios of the species in the read set using the heavy weight machinery of the Metropolis-Hastings Markov chain Monte Carlo algorithm.

The quality of the clustering starts to degrade when the read length decreases under 400 base pairs. The authors claim the quality, run time and memory to be on par with CompostBin. The time and space complexity of LikelyBin scales linearly with respect to the number of reads and number of species, but exponentially with respect to the k-mer length used. The authors write that values of k over 5 are infeasible to use.

SCIMM [19] (2010) models each species with an interpolated Markov model (IMM). An interpolated Markov model is a combination of multiple Markov models with varying degrees. The probability of a sequence given an IMM is computed as a weighted average of the probabilities of the constituent models. The algorithm alternates between two phases. In the first phase, the sequences are assigned to the most likely IMM, and in the second phase the parameters of the IMMs are re-computed to best match assignment in the second phase. These two steps are alternated until convergence.

The method is shown to work well for environments with ten or fewer species. The number of clusters k must be chosen by the user. Higher values of k will split dominant species into multiple clusters. However, the method does not work well for shorter reads, such as 100 base pairs, making it unsuitable for some high throughput sequencing platforms (see Table 2.1).

AbundanceBin [43] (2011) makes the assumption that two reads with long k-mers of similar frequency over the whole sample likely originate from the same species. The k-mers that occur only once are disregarded as they likely contain a sequencing error or come from a rare species. The algorithm clusters the k-mers using an expectation maximization algorithm, simultaneously optimizing the genome lengths and abundance ratios to best explain the data. When the expectation maximization has converged, each read is assigned to the k-mer cluster that has the highest probability of having generated the read under their model.

The authors show that their tool works even for very short reads down to 75 base pairs in length. They run the algorithm on simulated samples consisting of two species with varying abundance ratios, and conclude that the tool only works well if the abundance ratio of the two species is 2:1 or higher. They also show that the classification is better the higher the parameter k, and settle for a value of $k = 20$ in their experiments.

MetaCluster [23, 39, 40, 44, 45] (2010-2012) is a mature tool, which has been published five times with gradual improvements each time. It combines two insights. First, the observation that long k-mers are most likely contained in only one species in the sample and second, the short k-mer distribution of long segments (over 1000 base pairs) of a genome identify the genome. It solves the problem of the high dimensionality of the short k-mer composition vectors compared to read length by grouping reads into initial clusters on the grounds of long shared k-mers, and then running a variant of K-means on the short k-mer composition vectors of the initial clusters. The authors show that the tool can also find low abundance species by a two-round approach, where the most abundant species are identified first, and the same algorithm is then ran again on the remaining reads with slightly different parameters to identity rares species.

MBBC [38] (2015) is the latest tool for the problem. It combines k-mer abundance information much like AbundanceBin and Markov models much like SCIMM. The algorithm consists of two phases. First, an initial clustering is constructed using an expectation maximization utilizing the k-mer abundance information. Then in the second phase the clusters are refined by

constructing a Markov model out of the reads in each cluster, and iteratively reassigning reads to the most likely Markov model and retraining the models.

The authors show that their tools performs better than AbundanceBin and MetaCluster, but admit that the comparison might not have been totally fair, because they did not spend time choosing the parameters for the competing tools, and used the default parameters instead.

Summary. A variety of statistical methods have been successfully applied to the problem. However, none of the tools use sophisticated enough data structures to mention in the papers, which hints that there could be room for improvement in time and space complexity of the implementation. The paper of LikelyBin seem to be the only one which even mentions the algorithmic time and memory complexity of the implementation.

Technical performance, especially memory consumption, is a real issue in practice, since metagenomic datasets can be large. For example at the time of writing, the MG-Rast server [30] hosts 2450 projects of size larger than 10^9 base pairs and 90 projects of size larger than 10^{10} base pairs. For a dataset with $n = 10^{10}$ base pairs, merely storing n 32-bit pointers to the data will already take 40 gigabytes of memory, which is far beyond the capacity of a typical desktop computer at the time of writing. Pointerless data structures which take $O(n \log \sigma)$ bits of space instead of $O(n \log \sigma)$ have a significant advantage in these datasets, as σ is only four in DNA. Taking for example $n = 10^{10}$, we have $\log n \approx 33$ and $\log \sigma = 2$, a factor of 15 improvement.

The pipeline of MetaCluster is the best suited for the bidirectional BWT index described in this thesis. The other tools are more geared towards statistical estimation using counts of k-mers. While we might be able to implement most of the tools using our framework, MetaCluster stands out as the most interesting from a string algorithms and data structures point of view. Metacluster is also the most mature of the tools, having been published five times with incremental improvements. Therefore, in this work we focus on the pipeline of MetaCluster.

5.3 MetaCluster

We shall now describe the pipeline of MetaCluster in more detail. The pipeline starts with filtering reads which originate from a low-abundance species. This is detected by counting the frequencies of each k_1-mer in a read over the whole set of reads. It is implicit in all MetaCluster papers, that a k-mer is considered equivalent to its reverse complemented pair. If a read does not contain a k_1-mer of frequency τ_1 for some choice of threshold τ_1,

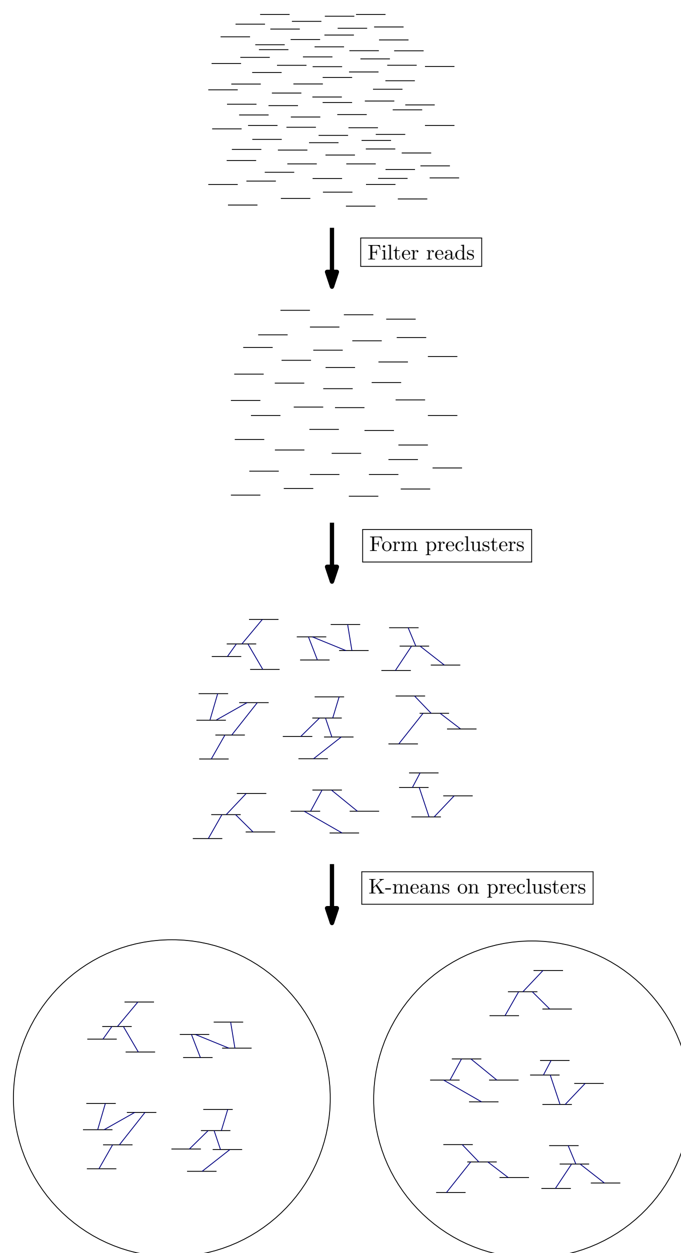

Figure 5.1: The pipeline of MetaCluster

then we discard the read. MetaCluster uses the value $k_1 = 16$ based on the data analysis in [12], and takes τ_1 as a user specified parameter. The MetaCluster 5 paper provides a way to calculate τ_1 to filter away all reads from all species with sequencing depth less than d given k_1, the average read length,

the average genome length, the error rate, and the tolerated probability of a mistake.

Figure 5.2 shows the distribution of the frequency of the most common 16-mer in a read set from a real human gut metagenome sequencing project [31]. The distribution is visibly bimodal. The interpretation is that the reads that contribute to the left peak come from a rare species, or possibly contain many errors, whereas the reads in the right peak correspond to reads from species with high abundance and few errors. Visually it seems from the figure that the threshold of $\tau_1 = 10$ would be the best to separate the two components. Determining this local minimum algorithmically would be possible, but it is not totally clear if the distribution has this bimodal shape for all datasets. The probabilistic estimation of τ_1 to filter away reads with a sequencing depth of less than 10 with a probability of 80% in the Metacluster 5 paper gives $\tau_1 = 4$, which is reasonable, although a bit low on the basis of figure 5.2. However improving the estimation of this parameter is outside of the scope of this thesis.

The next step in the pipeline is to form preclusters of reads that come from the same species with a very high probability. MetaCluster decides that two reads come from the same species if they share a k_2-mer of length at least 36. The preclusters containing two such reads are merged if the probability of a false positive merge does not exceed a user-defined threshold value. The parameter $k_2 = 36$ was chosen by probabilistically estimating the threshold at which two reads sampled in a nearby region of a genome are merged together in the same precluster with 99% accuracy.

Figure 5.3 shows experimental evidence that the choice of $k_2 = 36$ is reasonable. The Figure shows the distribution of the length of the average match over all starting positions from a reference genome of Escherichia coli to a reference genome of a species in the genus of Acidovorax from the NCBI databse [33]. These two genomes are far apart in the tree of life, which means that their evolutionary lines have diverged far away in the past, which in turn means that their genomes should have little similarity. The Figure shows that there are next to no matches of length 36 between the species. The lengths of both genomes are approximately 5 million base pairs, which means that the average match length would be $\log_4(5 \cdot 10^6) = 11.1$ if the strings were random. The peak of the histogram coincides with this value, which implies that most of the matches can be attributed to chance alone.

Finally, MetaCluster clusters the pre-clusters using a variant of the K-means algorithm using the Spearman footrule (Definition 5.3.1) on short k_3-mer distributions of the preclusters as a distance measure. The clusters produced by the K-means algorithm are further refined by joining two clusters if the Spearman footrule average distance between clusters is sufficiently low.

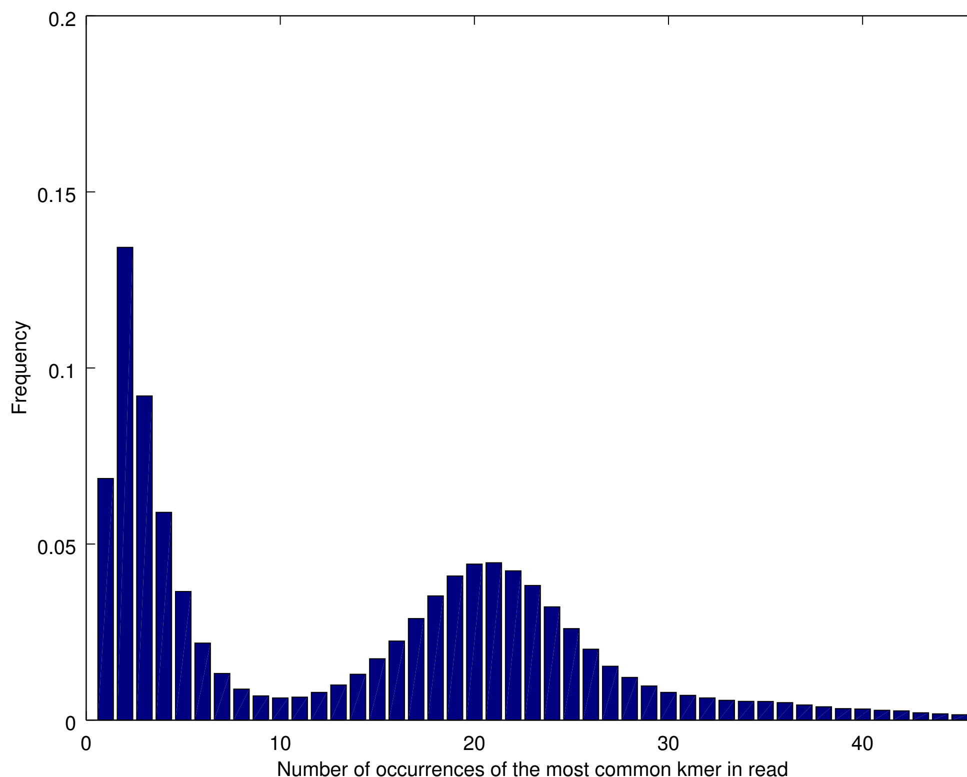

Figure 5.2

The value $k_3 = 5$ is chosen to keep the dimensionality of the distributions reasonable. Figure 5.1 shows a summary of the pipeline. The whole pipeline can be ran again on the reads that did not pass the filtering with slightly different parameters k_1, τ_1 and k_2 to attempt to cluster reads with lower abundances.

Definition 5.3.1. *Spearman footrule.* Let A and B be arrays of length n. Build arrays A' and B' such that $A'[i]$ is the number of positions j such that $A[j] < A[i]$, or $A[j] = A[i]$ and $j < i$, and similarly for B'. Let $f_{A'}(i)$ and $f_{B'}(i)$ be functions that return the index of value i in A' and B', respectively. The Spearman footrule distance between A and B is then $\sum_{i=1}^{n} |f_{A'}(i) - f_{B'}(i)|$.

The algorithmically challenging parts of the pipeline are the filtering and the precluster building, as these require searching for k-mers over the whole

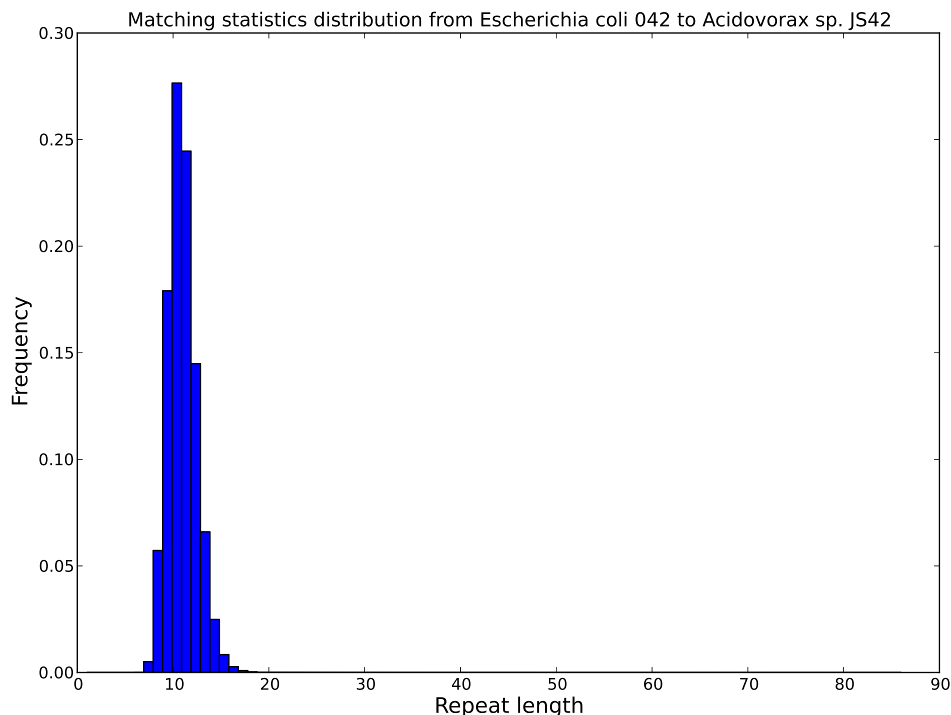

Figure 5.3

read set. The K-means phase involves only finding all k-mers locally within a precluster, which can be easily done without sophisticated pointerless data structures, because preclusters can be processed independently of each other and the sizes of the preclusters are small in practice. In this work we focus on the problems of filtering and precluster building. We now formalize the two problems in the language of string algorithms. First, the filtering problem:

The filtering problem: Given a set of strings R, find all strings $s \in R$ such that s contains a k-mer α such that the combined number of occurrences of α and $\widetilde{\alpha}$ in R is at least τ. In the case where $\alpha = \widetilde{\alpha}$, we do not double count the occurrences.

Next, we formulate the precluster problem without including the probabilistic reasoning in MetaCluster about not merging too large preclusters if the probability of a false positive merge is too high. The reason for this is to be able to formulate a cleaner, purely string-algorithmic problem. Given a

solution to the string algorithmic problem, incorporating the probabilistic reasoning into the solution is not difficult.

The precluster problem: Given a set of strings R, partition R into equivalence classes such that two strings $s_1, s_2 \in R$ are in the same class if and only if there is a k-mer α such that α or $\widetilde{\alpha}$ is a substring of s_1 and α or $\widetilde{\alpha}$ is a substring of s_2 .

5.4 Solving the filtering problem with the bidirectional index

The goal is to remove all reads from the read set that do not contain a k_1-mer that appears in at least τ_1 times in the read set. The index we use is a bidirectional BWT index built for the concatenation of all reads where the reads are delimited with a special unique separator character outside of the alphabet of the read set.

The first step is to find and compute the lexicographic intervals of all RC right-maximal (see section 4.5.1) k-mers. We will fill these intervals with ones in a bit vector B of length equal to the length of the BWT. The intervals can be marked by extending the iteration framework in Algorithm 6. In addition to the four interval pairs, we also store the string length of the substrings corresponding to the intervals in the stack.

Suppose we take the stack frame corresponding to a k_1-mer α from the stack. The number of occurrences of α and $\widetilde{\alpha}$ combined in equal to the size of the union of the intervals $[i_\alpha^\rightarrow, j_\alpha^\rightarrow]$ and $[i_{\widetilde{\alpha}}^\rightarrow, j_{\widetilde{\alpha}}^\rightarrow]$. Since intervals of distinct k-mers do not overlap, this is equal to $(j_\alpha^\rightarrow - i_\alpha^\rightarrow + 1) + (j_{\widetilde{\alpha}}^\rightarrow - i_{\widetilde{\alpha}}^\rightarrow + 1)$ unless $\alpha = \widetilde{\alpha}$, in which case the intervals are equal, and the combined size is just $(j_\alpha^\rightarrow - i_\alpha^\rightarrow + 1)$. If the combined size is greater than τ_1, we fill both intervals with ones in B. After all k-mer have been iterated, we walk the concatenation of reads backwards while maintaining the lexicographic rank of the suffix starting from the current position using the technique described in section 4.5.4. Whenever the lexicographical rank of the current suffix is marked in B, we flag the current read to be kept, and output the read once we reach the start of the read.

It may not be immediately clear why it is enough to only consider the RC right-maximal k_1-mers. We will now argue the correctness of the algorithm using a short Lemma.

Lemma 3. *A read contains a k_1-mer α such that the combined frequency of α and $\widetilde{\alpha}$ is at least τ_1 if and only if the read contains a RC right-maximal*

k_1-mer β such that the combined frequency of β and $\widetilde{\beta}$ is at least τ_1.

Proof. ($\Rightarrow$) Suppose a read r has a k_1-mer α such that α and $\widetilde{\alpha}$ have at least τ occurrences combined. Extend the α to the right until it is not RC right-maximal anymore. The number of occurrences remains the same after doing this. Then contract the substring from the left until it has length k_1 again. The number of occurrences can only increase while doing this, and the final substring β remains RC right-maximal and by construction r contains β.
($\Leftarrow$) Trivial, as a RC right-maximal k_1-mer is a special case of a k_1-mer. □

Lemma 3 says that if a read should be kept, then it has a RC right-maximal k_1-mer of combined frequency τ_1, and on the other hand, if it should not be kept, then there is no such k_1-mer, and therefore we need to only consider the RC right-maximal subset of k_1-mers for the purpose of filtering.

The total space needed by the algorithm is equal to the size of the bidirectional index and the bit vector B combined. The number bits needed is therefore $2|S|\log\sigma + |S| + o(|S|\log\sigma)$, where the first term comes from the two Burrows-Wheeler transforms, the second term comes from the bit vector B and the last term comes from the rank support structures (wavelet trees) of the Burrows-Wheeler transforms.

5.5 Solving the precluster problem with the bidirectional index

Now we show how to solve the precluster problem using the bidirectional index built for the concatenation of the read set. We define that the rank of a read is the number of reads that come before it in the concatenation, plus one to make the ranks start from one. We solve the problem in two phases. First, we compute the equivalence classes of the relation where two reads are related if and only if there is a k_2-mer that is contained in both reads. Next we will merge all equivalence classes C_1 and C_2 such that there is a k_2-mer α such that α is contained in some read in C_1, and $\widetilde{\alpha}$ is contained in some read in C_2.

To implement the merging operations, we use a Union-Find data structure (see section 3.7) as a black box with operations find(r) which gives a handle to the class containing the read r, union(C_1, C_2), which merges the classes C_1 and C_2 and and size(C), which gives the size of the class C. We initialize the structure such that every read is in a separate equivalence class initially.

In the first step, we iterate and mark the start and end points of the lexicographic intervals of all right-maximal k_2-mers in a bit vector B of length

equal to the size of the BWT using Algorithm 5. We index B for rank-queries (see section 3.4) so that we can check whether an index i is inside a marked interval by checking whether $B[i] = 1$ or the $rank_B(i)$ is odd. Each marked interval is associated with the set of reads at the start points of the suffixes in the interval. For each marked interval we store the class handle of one of these reads. The handles are stored in an array H of length equal to the number of marked intervals such that if position i is inside a marked interval in B, the class handle corresponding to the interval can be found at $H[rank_B(\lceil \frac{i}{2} \rceil)]$. We initialize the array H with null values.

After initializing H, we walk the concatenation backwards while maintaining the lexicographic rank $i^{\rightarrow}$ of the suffix starting from the current position and the rank r of the read containing the suffix using the backward step of the forward BWT. At each step, we check whether the current lexicographic rank $i^{\rightarrow}$ is in a marked interval in B. If so, let $\ell = rank_B(\lceil \frac{i^{\rightarrow}}{2} \rceil)$ be the index of the class handle corresponding to the interval containing $i^{\rightarrow}$ in H. If $H[\ell]$ is null, we set $H[\ell] = \text{find}(r)$, else we execute union($H[\ell]$, find(r)).

After this is done, we merge all classes C_1, C_2 that share a reverse complemented k_2-mer. At this point we free the bit vector B and the array H from memory and initialize another bit vector B_2 of length equal to the length of the BWT with zeroes. We iterate all RC right-maximal k-mers using algorithm 6 and at each such k-mer α, if the interval $[i^{\rightarrow}_{\tilde{\alpha}}, j^{\rightarrow}_{\tilde{\alpha}}]$ is non empty, we know we should merge the class of the interval $[i^{\rightarrow}_{\alpha}, j^{\rightarrow}_{\alpha}]$ with the class of the interval $[i^{\rightarrow}_{\tilde{\alpha}}, j^{\rightarrow}_{\tilde{\alpha}}]$. However we can not do it yet, because we do not know the corresponding cluster handles, so pick arbitrary positions $i \in [i^{\rightarrow}_{\alpha}, j^{\rightarrow}_{\alpha}]$ and $j \in [i^{\rightarrow}_{\tilde{\alpha}}, j^{\rightarrow}_{\tilde{\alpha}}]$, mark $B_2[i] = 1$ and $B_2[j] = 1$ and write the pair of positions i, j to disk for future reference. When finished, we index B_2 for rank queries.

Finally after all RC right-maximal k-mers have been iterated, we find the read ranks corresponding to all marked positions in B_2 by backward stepping through the BWT. We can store these in an array H_2 of length equal to the number of marks in B_2 such that $H_2[i]$ contains the read rank of the marked position with rank i in B_2. Then we proceed to stream the position pairs previously written to disk. For each pair i, j, we look up the read ranks r_i and r_j from $H_2[rank_{B_2}(i)]$ and $H_2[rank_{B_2}(j)]$, respectively, and execute the command union(find(r_i), find(r_j)). After all is done, the classes in the union-find structure are exactly the desired equivalence classes.

In practice, MetaCluster sets a maximum size threshold for the sizes of the preclusters, because the probability of a false-positive merge due to shared long k_2-mers in distinct species increases quickly as the sizes of the equivalence classes to be merged increases. This can be implemented by simply querying the sizes of the equivalence classes to be merged, and skipping the merge operation if the sum of the sizes is above a specified threshold.

In summary, the algorithm requires the bidirectional index, the union-find structure and the auxiliary data arrays B, B_2, H and H_2. However only a subset of the arrays is stored in memory at any given time. Figure 5.4 shows an overview of the structures stored in memory over time, and the space complexities of each of the structures. The line next to each structure represents the time period the structure is held in memory. The symbols λ and λ_{rc} denote the number of right-maximal and RC right-maximal k_2-mers in $|S|$, and the symbol R denotes the number of reads. The sublinear terms in the time complexities come from the rank support structures, and the $2R \log R$ space complexity of the union find structure comes from the fact that for each read we need to store a pointer to its parent in the structure, and the size of the subtree attached to it to maintain the balance of the trees. The peak of the memory usage occurs the latter half of the algorithm, at which we need a total of $2|S| \log \sigma + (2R + \lambda_{rc}) \log R + |S| + o(|S| \log \sigma)$ bits of space. In practice for real datasets, our experiments suggested that this is usually less than $16|S|$, i.e. two bytes per character.

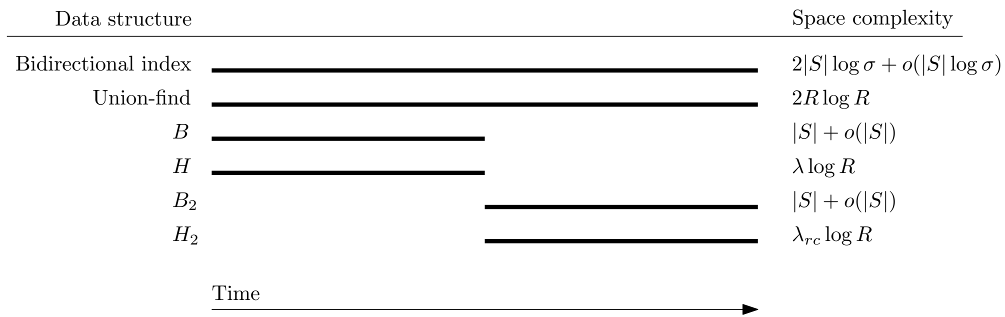

Figure 5.4: Overview of the data structures used during the preclustering algorithm.

Chapter 6

Implementation

The bidirectional index was implemented in C++ using the succinct data structures library by Simon Gog et al. [13]. The implementation is available at `https://github.com/jnalanko/BD_BWT_index`. The interface of the index along with the associated time complexities are shown in Table 6.1.

Operation	Input	Output	Time Complexity
Extend left	**Interval pair** of a substring α and a **symbol** $c \in \Sigma$	**Interval pair** of the substring $c\alpha$	$O(\sigma \log \sigma)$
Extend right	**Interval pair** of a substring α and a **symbol** $c \in \Sigma$	**Interval pair** of the substring αc	$O(\sigma \log \sigma)$
Right maximality	**Interval pair** of a substring α	**Boolean value** whether α is right-maximal	$O(\sigma \log \sigma)$
Left maximality	**Interval pair** of a substring α	**Boolean value** whether α is left-maximal	$O(\sigma \log \sigma)$
Backward step	**Lexicographic rank** r	**Lexicographic rank** of $S[(SA[r]-1)..\vert S \vert]$	$O(\log \sigma)$

Table 6.1: Supported operations

A complete clustering pipeline including filtering, preclustering and K-means in available at `https://github.com/jnalanko/bwtCluster`. The implementation goes by the name bwtCluster. The pipeline was implemented with support for multithreaded processing using the threading support in the C++11 standard library.

Before building the BWTs, the read set is sorted. The reads are concatenated and a dollar character is placed in between each read, and at the start and the end of the whole concatenation. This gives the concatenation the useful property that the suffix starting at the dollar preceding the i-th read in the concatenation has lexicographic rank $i + 1$, and the last dollar has lexicographic rank 1. The ropebwt library [24] was used for the parallel construction of the Burrows-Wheeler transforms.

The BWT inversion (Algorithm 2) was straightforward to modify for parallel processing. Given T threads, we simply divide the text into T parts of approximately equal size, and backward step through each part in parallel.

If there are m dollars in the text, we set the i-th thread to start at the dollar with rank $\lfloor m/T \rfloor i$, and end where the previous thread starts. Because of the sorting of the read set, we know the lexicographic ranks of the dollars needed to start the threads. The parts are completely independent, so no synchronization primitives are needed.

The suffix link traversal (Algorithm 5) based algorithms were more challenging to parallelize. We assign a disjoint subtree of the suffix link tree for each thread. The problem with this is that the subtrees might not be of equal size. To address this issue, we have a simple mechanism to balance work between threads. Whenever a thread has finished processing its subtree, it sets a flag to indicate it has run out of work. At every iteration, every thread checks this flag, and if it is set, it gives the stack frame at the bottom of its iteration stack to the new thread, which will then start iterating the subtree attached to that stack frame. This results in some synchronization overhead, but in practice the suffix link tree traversal takes only a small fraction of the total running time of the pipeline, so we chose not to spend time on implementing more sophisticated synchronization mechanisms.

Our pipeline does not exactly replicate the pipeline of MetaCluster. For instance, our pipeline does not allow mismatches in reads, while MetaCluster allows one mismatch, and our tool does not try to estimate the number of clusters in K-means and instead takes the number of clusters as a user specified parameter. Replicating MetaCluster completely turned out to be difficult, because what is actually implemented in MetaCluster is a mixture of the methods presented in the MetaCluster papers [23, 39, 40, 44, 45], and finding out the details directly from the source code seemed difficult and time consuming. We chose not to spend too much time on this, because the focus of the thesis is not to replicate MetaCluster down to the details. A proper comparison to MetaCluster is deferred to an upcoming scientific publication on the project.

6.1 Correctness

The correctness of our tool was tested on low complexity simulated data. We ran three experiments, each on a different level of taxonomy. All test datasets contain exactly two species, with tenfold coverage with paired end reads of length 100 base pairs and no errors. The first dataset contains the species Vibrio cholerae and Vibrio vulnificus, both from the genus Vibrio. We call this the species level dataset. The second dataset contains Vibrio cholerae and Photobacterium gaetbulicola, both from the family Vibrionaceae, but different genuses. This is called the genus level dataset. The third dataset

contains Vibrio cholerae and Escherichia coli, both from the class Gammaproteobacteria, but different families. This is called the family level dataset. The source genomes were taken from the NCBI databse. We used the same same parameters as MetaCluster, namely $k = 16, \tau = 4$ for filtering, $k = 36$ for preclustering and $k = 4$ for K-means. The number of clusters in K-means was set to 2, as our tool can not estimate the number of clusters by itself.

For each dataset, we measured the average *purity* of the preclusters, and the final clusters. The purity of a cluster containing n_1 reads from one species of the dataset and n_2 reads from the other species is defined to be $\max(n_1, n_2)/(n_1 + n_2)$, so a perfect cluster has purity 1, and a completely mixed cluster purity 0.5. Each experiment was ran 10 times and the average results were computed. The results are listed in Table 6.2. It is shown that the preclusters are good quality on all tested levels of taxonomy, but the final clusters are reasonable only starting at the genus level. We tried to run the same test cases with MetaCluster, but the program fails and prints an error message saying that the number of preclusters is too small because the coverage is too low. We were only able to run MetaCluster successfully on the datasets used in the MetaCluster papers, but for those we do not know the true species of the reads, so they could not be used to assess the correctness of the clusters.

Dataset	**Precluster purity**	**Final cluster purity**
Species level	0.97197	0.63395
Genus level	0.99951	0.94117
Family level	0.99395	0.98287

Table 6.2: Results of low complexity data

6.2 Performance

The peak memory and running time of our tool was compared to MetaCluster and MBBC. The experiments were conducted on A machine with 32GB of RAM and two Intel Xeon E5540 2.53GHz CPUs with 4 cores each. Figure 6.4 shows the memory usage of both MetaCluster and bwtCluster plotted as a function of time on a sample from a human gut sequencing project [31] (sample id ERR011087), which is one of the benchmark datasets used in the paper presenting MetaCluster 5.0 [40]. The memory usage was measured at intervals of 100 milliseconds using the resident memory size reported by the Unix `ps` command. Our implementation was 10 times faster and used 10

times less memory at peak. However, here it is important to keep in mind the fact that our algorithm does not do everything that MetaCluster does, as explained in the introduction of this chapter.

A similar comparison was conducted against the MBBC software. The test dataset was an example dataset provided by the MBBC package. The memory consumption as a function of time is shown in Figure 6.5. Our implementation was over 5 times faster, and uses 50 times less space.

The parallelization of the key algorithmic components, i.e. the BWT inversion and the suffix link traversal with and without reverse complements were benchmarked on a machine with two Intel Xeon E5-2420 CPUs, each with 6 cores running at 1.90GHz and 124GiB of DDR3 RAM clocked at 1333 MHz. The machine can run 24 parallel threads with hyperthreading. A 42MB freshwater sample from MG-Rast was chosen as the test dataset, because its moderate size allows running the algorithms multiple times to even out the variance in the running time. Figure 6.3 shows the time and the speedup factor over a single thread as the function of number of threads used for the BWT inversion. Figures 6.2 and 6.1 show the same plots for a full traversal of the suffix link tree without reverse complements (Algorithm 5) and with reverse complements (Algorithm 6), respectively. As expected, we see that the BWT inversion parallelizes much better than the suffix link tree traversal. The best running time for inversion is achieved with 24 threads, which is exactly the number of parallel threads our benchmarking machine is able to run with hyperthreading. However, the speedup is only 14-fold, even though in theory it could be up to 24-fold. This is most likely due to low level hardware details.

The performance for the suffix link tree traversal however saturates at 5 threads already. This is because there is now synchronization between the threads. It is interesting to note that when the number of threads is increased over 12, which is the number of cores in the machine, the performance starts to *degrade*, stabilizing at around 18 threads. It seems that the gains from hyperthreading are overshadowed by the overhead of context switching. The performance for the reverse complement enhanced traversal is slightly better, because the backward step operation is heavier, which means a smaller portion of the time is spent dealing with the synchronized parts of the code.

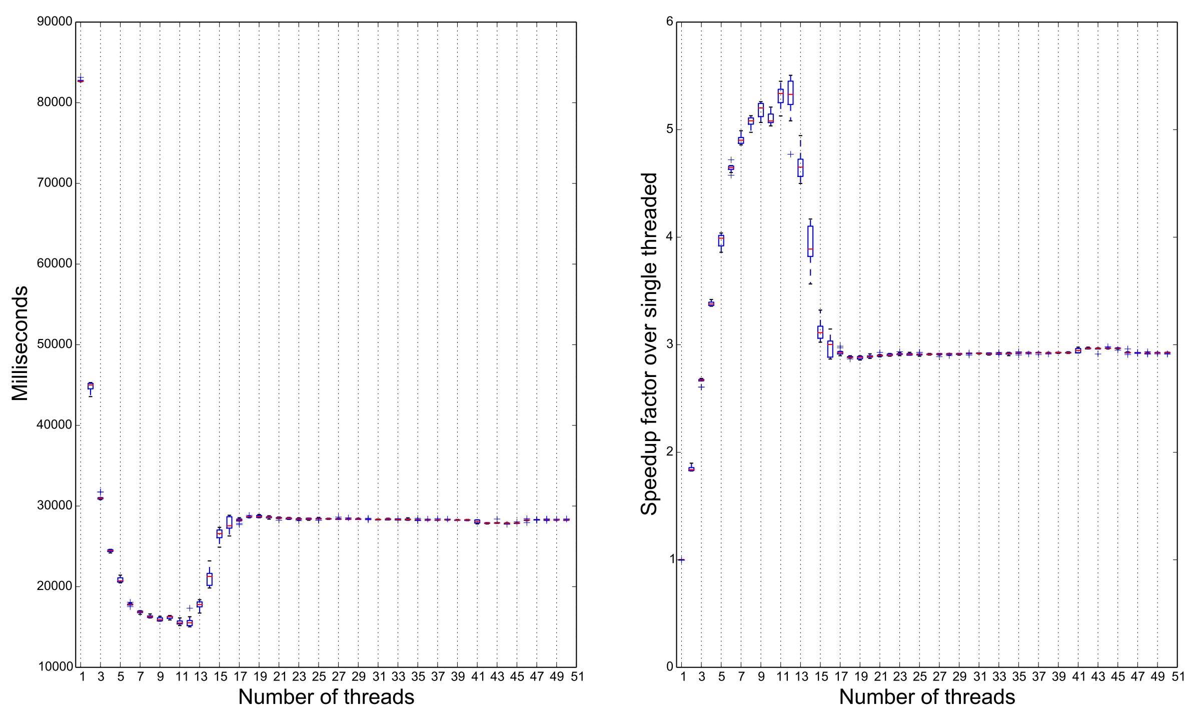

Figure 6.1: Full suffix tree traversal with reverse complements

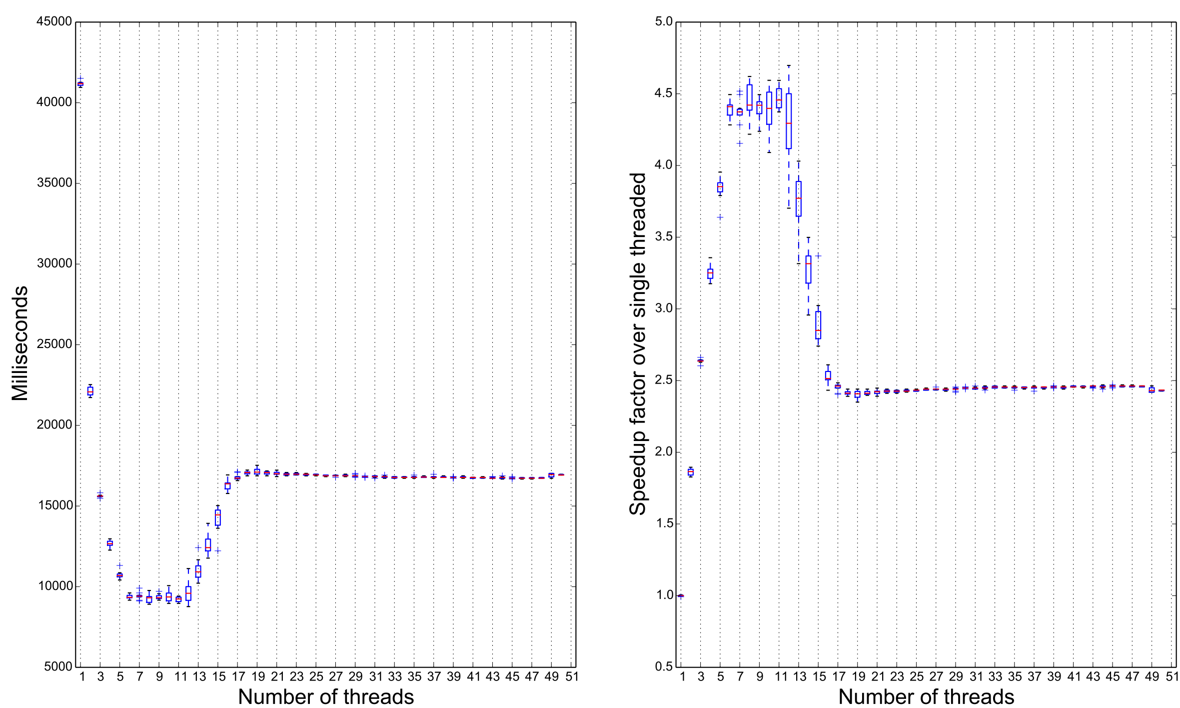

Figure 6.2: Full suffix tree traversal without reverse complements

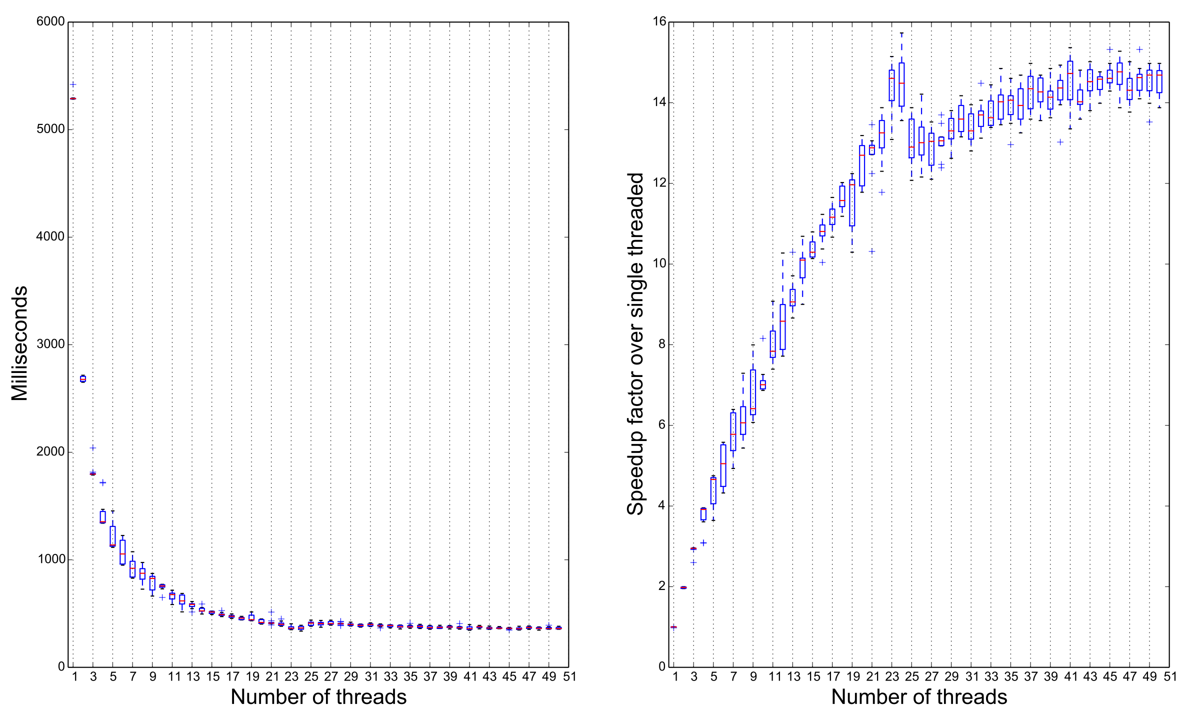

Figure 6.3: BWT inversion

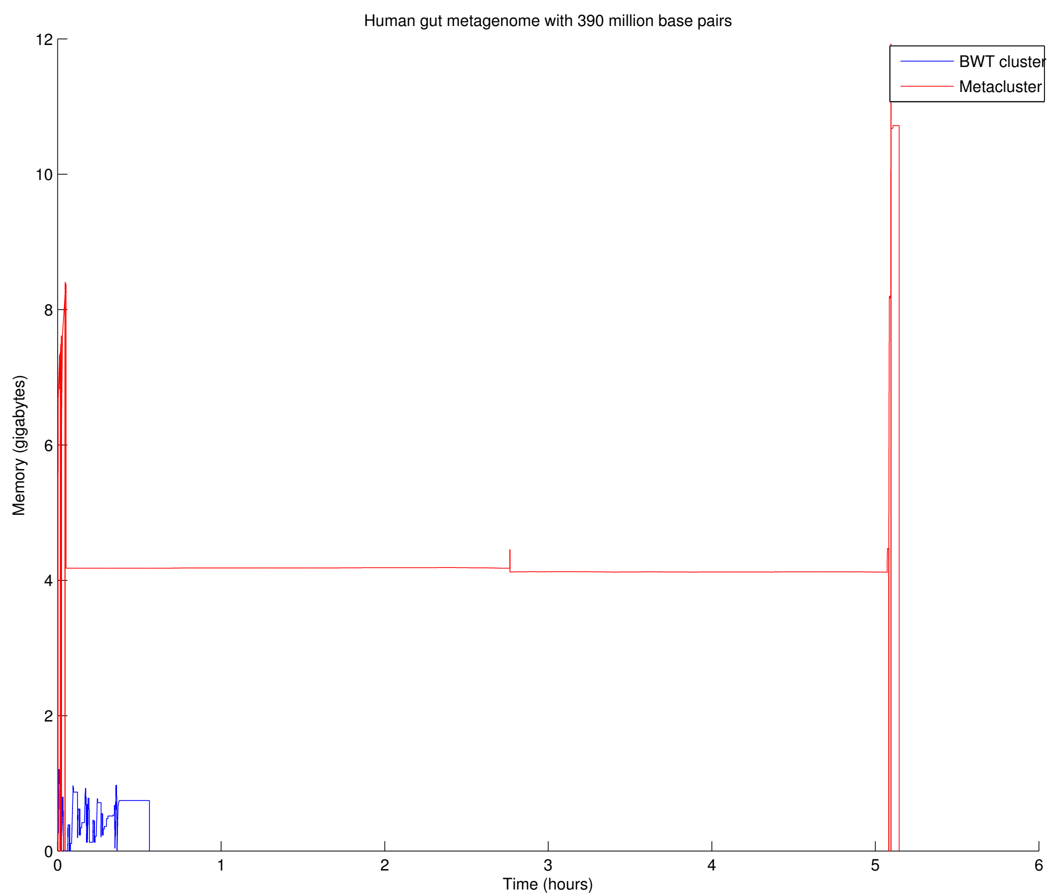

Figure 6.4: MetaCluster vs BWTcluster

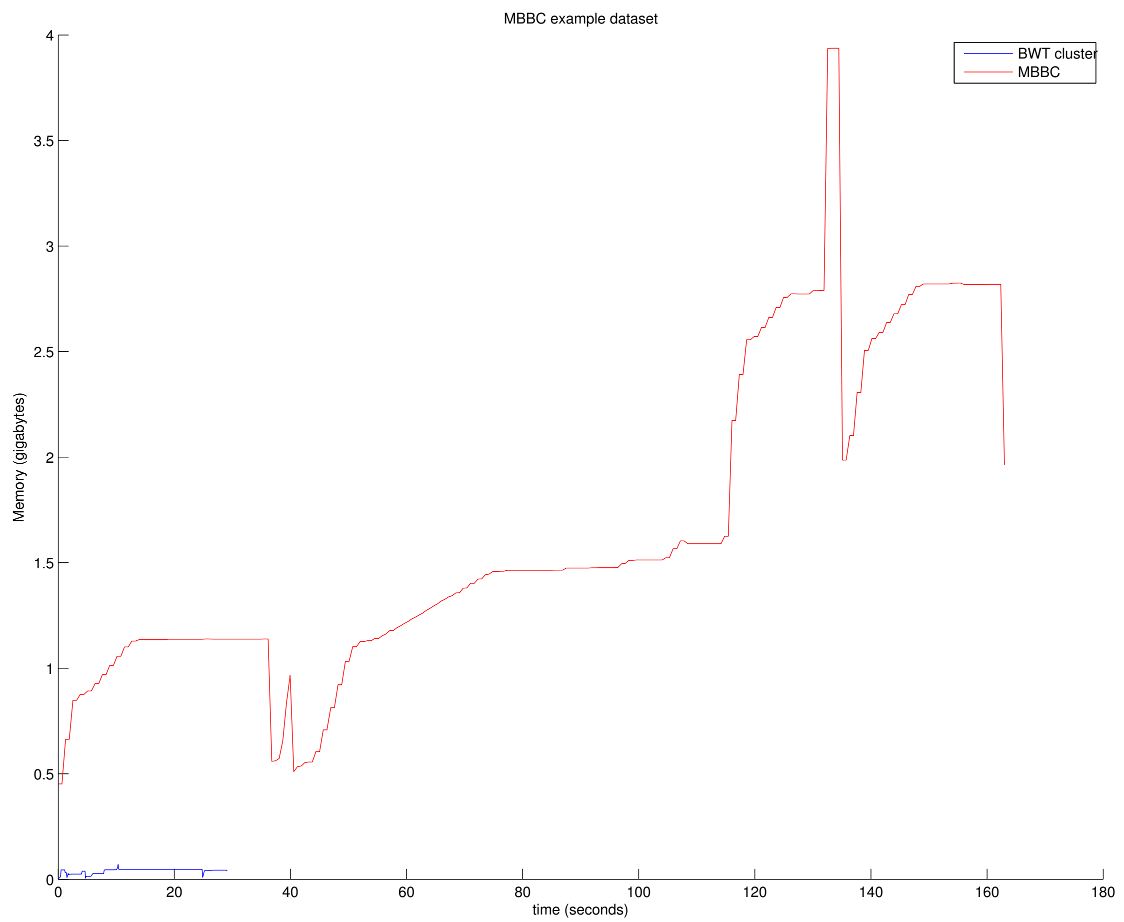

Figure 6.5: MBBC vs metacluster

Chapter 7

Conclusion

The purpose of the thesis was to apply recent research on Burrows-Wheeler type indices in the field of metagenomics. The pipeline of the clustering tool MetaCluster was determined to be suitable to implement on top of the bidirectional Burrows-Wheeler index. A new theoretical concept, reverse complement right-maximality, was introduced. The filtering and the clustering problem in MetaCluster were solved using the bidirectional index with the notion of RC right-maximality. The algorithms were implemented with support for parallel processing using C++ on top of the SDSL library. Experiments showed that our implementation performs better in both time and space consumption in practice. Preliminary experiments suggest that the quality of the clustering is good for samples with two species and no errors, given the species are from a different genus. Unfortunately, a proper comparison in the quality versus MetaCluster could not be conducted because of problems running the MetaCluster software.

The work was presented in September 2015 in London in the workshop on compression, text and algorithms (WCTA), organized in conjunction with the 22nd edition of the International Symposium on String Processing and Information Retrieval (SPIRE).

Many theoretical and practical questions were left unanswered. Perhaps most important of all on the theory side, the question whether the algorithms presented can be modified for efficient approximate matching, such that two substrings are considered a match if they differ only in a limited number of positions. There is a folklore trick to tolerate one mismatch, but tolerating multiple mismatches is difficult. The need to also take into account reverse complemented repeats further complicates the problem. The observation to tolerate one mismatch is that the mismatch could be in the first half of the string, or in the second half of the string but not both. To find all matches of a substring α tolerating one mismatch, one can find the lexicographic

interval of all exact matches of the first half of α, and from there explore all the right halves of α, allowing branching on one mismatch, and then do the same in the reverse direction, matching the second half of α exactly and allowing one mismatch in the first half. This has a chance to be effective in practice, because if α is long, the number of exact occurrences of the first and the second half of α is often small, and therefore the branching caused by allowing the mismatch is manageable. Implementing a mismatch-tolerant pipeline would make our tool better comparable to MetaCluster. Further research could be done on how to tolerate a limited number of insertions or deletions between strings, an adapt those algorithms to take into account reverse complement matches.

On the more practical side of things, the evaluation of the purity of the clusters of our tool could be done in greater depth. Interesting experiments to run include finding out how the tool works when errors are introduced, the number of species is increased and the abundance ratios of the species is varied. The results could be then compared to those of the state-of-the-art tools. It would also be interesting to run the clustering on real metagenomic samples, and try get a new biological insight on the structure of an environment.

Another direction would be to try to apply the clustering to compress read sets. A cluster could be encoded by building a single reference genome for the cluster, and representing the reads as pointers to this reference genome, encoding the mismatches separately. The idea exists in the literature [16], but it has not been explored in depth. For this however we would need to implement the heuristics of MetaCluster to estimate the number of clusters for the final K-means phase.

Bibliography

[1] AllSeq knowledge bank. `http://allseq.com/knowledgebank/sequencing-platforms/pacific-biosciences`. Accessed: 10.9. 2015.

[2] Amann, R. I., Ludwig, W., and Schleifer, K.-H. Phylogenetic identification and in situ detection of individual microbial cells without cultivation. *Microbiological reviews 59*, 1 (1995), 143–169.

[3] Belazzougui, D. Linear time construction of compressed text indices in compact space. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing* (2014), ACM, pp. 148–193.

[4] Belazzougui, D., Cunial, F., Kärkkäinen, J., and Mäkinen, V. Versatile succinct representations of the bidirectional burrows-wheeler transform. In *Algorithms–ESA 2013*. Springer, 2013, pp. 133–144.

[5] Burrows, M., and Wheeler, D. J. A block-sorting lossless data compression algorithm. *Technical Report 124 Palo Alto, CA: Digital Equipment Corporation* (1994).

[6] Chatterji, S., Yamazaki, I., Bai, Z., and Eisen, J. A. Compostbin: A dna composition-based algorithm for binning environmental shotgun reads. In *Research in Computational Molecular Biology* (2008), Springer, pp. 17–28.

[7] Clark, D. *Compact Pat Trees*. PhD thesis, University of Waterloo, 1998.

[8] Cormen, T. H. *Introduction to algorithms*. MIT press, 2009.

[9] Federhen, S. The ncbi taxonomy database. *Nucleic acids research 40*, D1 (2012), D136–D143.

[10] Ferragina, P., and Manzini, G. Opportunistic data structures with applications. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on* (2000), IEEE, pp. 390–398.

[11] Ferragina, P., and Manzini, G. Indexing compressed text. *Journal of the ACM (JACM) 52*, 4 (2005), 552–581.

[12] Fofanov, Y., Luo, Y., Katili, C., Wang, J., Belosludtsev, Y., Powdrill, T., Belapurkar, C., Fofanov, V., Li, T.-B., Chumakov, S., et al. How independent are the appearances of n-mers in different genomes? *Bioinformatics 20*, 15 (2004), 2421–2428.

[13] Gog, S., Beller, T., Moffat, A., and Petri, M. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)* (2014), pp. 326–337.

[14] Grossi, R., and Vitter, J. S. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing 35*, 2 (2005), 378–407.

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

[16] Hach, F., Numanagić, I., Alkan, C., and Sahinalp, S. C. Scalce: boosting sequence compression algorithms using locally consistent encoding. *Bioinformatics 28*, 23 (2012), 3051–3057.

[17] Jacobson, G. J. *Succinct static data structures.* PhD thesis, Carnegie Mellon University, 1988.

[18] Kärkkäinen, J., and Sanders, P. Simple linear work suffix array construction. In *Automata, Languages and Programming.* Springer, 2003, pp. 943–955.

[19] Kelley, D. R., and Salzberg, S. L. Clustering metagenomic sequences with interpolated markov models. *BMC Bioinformatics 11*, 1 (2010), 544.

[20] Kislyuk, A., Bhatnagar, S., Dushoff, J., and Weitz, J. S. Unsupervised statistical clustering of environmental shotgun sequences. *BMC Bioinformatics 10*, 1 (2009), 316.

[21] Ko, P., and Aluru, S. Space efficient linear time construction of suffix arrays. In *Combinatorial Pattern Matching* (2003), Springer, pp. 200–210.

[22] Lam, T. W., Li, R., Tam, A., Wong, S., Wu, E., and Yiu, S.-M. High throughput short read alignment via bi-directional bwt. In *Bioinformatics and Biomedicine, 2009. BIBM'09. IEEE International Conference on* (2009), IEEE, pp. 31–36.

[23] Leung, H. C., Yiu, S.-M., Yang, B., Peng, Y., Wang, Y., Liu, Z., Chen, J., Qin, J., Li, R., and Chin, F. Y. A robust and accurate binning algorithm for metagenomic sequences with arbitrary species abundance ratio. *Bioinformatics 27*, 11 (2011), 1489–1495.

[24] Li, H. Fast construction of fm-index for long sequence reads. *Bioinformatics* (2014), 3274–3275.

[25] Liu, L., Li, Y., Li, S., Hu, N., He, Y., Pong, R., Lin, D., Lu, L., and Law, M. Comparison of next-generation sequencing systems. *BioMed Research International 2012* (2012).

[26] Lukjancenko, O., Wassenaar, T. M., and Ussery, D. W. Comparison of 61 sequenced escherichia coli genomes. *Microbial ecology 60*, 4 (2010), 708–720.

[27] Mäkinen, V., Belazzougui, D., Cunial, F., and Tomescu, A. I. *Genome-Scale Algorithm Design.* Cambridge University Press, 2015.

[28] Manichanh, C., Rigottier-Gois, L., Bonnaud, E., Gloux, K., Pelletier, E., Frangeul, L., Nalin, R., Jarrin, C., Chardon, P., Marteau, P., et al. Reduced diversity of faecal microbiota in crohn's disease revealed by a metagenomic approach. *Gut 55*, 2 (2006), 205–211.

[29] Mayr, E. *Systematics and the origin of species, from the viewpoint of a zoologist.* Harvard University Press, 1942.

[30] Meyer, F., Paarmann, D., D'Souza, M., Olson, R., Glass, E. M., Kubal, M., Paczian, T., Rodriguez, A., Stevens, R., Wilke, A., et al. The metagenomics rast server–a public resource for the automatic phylogenetic and functional analysis of metagenomes. *BMC bioinformatics 9*, 1 (2008), 386.

[31] Qin, J., Li, R., Raes, J., Arumugam, M., Burgdorf, K. S., Manichanh, C., Nielsen, T., Pons, N., Levenez, F., Yamada, T., et al. A human gut microbial gene catalogue established by metagenomic sequencing. *nature 464*, 7285 (2010), 59–65.

[32] Sadakane, K. Compressed suffix trees with full functionality. *Theory of Computing Systems 41*, 4 (2007), 589–607.

[33] Sayers, E. W., Barrett, T., Benson, D. A., Bolton, E., Bryant, S. H., Canese, K., Chetvernin, V., Church, D. M., DiCuccio, M., Federhen, S., et al. Database resources of the national center for biotechnology information. *Nucleic acids research 39*, suppl 1 (2011), D38–D51.

[34] Staley, J. T. Biodiversity: are microbial species threatened?: Commentary. *Current Opinion in Biotechnology 8*, 3 (1997), 340–345.

[35] Teeling, H., Meyerdierks, A., Bauer, M., Amann, R., and Glöckner, F. O. Application of tetranucleotide frequencies for the assignment of genomic fragments. *Environmental microbiology 6*, 9 (2004), 938–947.

[36] Teeling, H., Waldmann, J., Lombardot, T., Bauer, M., and Glöckner, F. O. Tetra: a web-service and a stand-alone program for the analysis and comparison of tetranucleotide usage patterns in dna sequences. *BMC bioinformatics 5*, 1 (2004), 163.

[37] Vandamme, P., Pot, B., Gillis, M., De Vos, P., Kersters, K., and Swings, J. Polyphasic taxonomy, a consensus approach to bacterial systematics. *Microbiological reviews 60*, 2 (1996), 407–438.

[38] Wang, Y., Hu, H., and Li, X. Mbbc: an efficient approach for metagenomic binning based on clustering. *BMC Bioinformatics 16*, 1 (2015), 36.

[39] Wang, Y., Leung, H. C., Yiu, S.-M., and Chin, F. Y. Metacluster 4.0: a novel binning algorithm for ngs reads and huge number of species. *Journal of Computational Biology 19*, 2 (2012), 241–249.

[40] Wang, Y., Leung, H. C., Yiu, S.-M., and Chin, F. Y. Metacluster 5.0: a two-round binning approach for metagenomic data for low-abundance species in a noisy sample. *Bioinformatics 28*, 18 (2012), i356–i362.

[41] Wayne, L. G., and ICSB, J. C. International committee on systematic bacteriology: announcement of the report of the ad hoc committee on reconciliation of approaches to bacterial systematics. *Zentralblatt für Bakteriologie, Mikrobiologie und Hygiene. Series A: Medical Microbiology, Infectious Diseases, Virology, Parasitology 268*, 4 (1988), 433–434.

[42] Wu, S., Zhu, Z., Fu, L., Niu, B., and Li, W. Webmga: a customizable web server for fast metagenomic sequence analysis. *BMC genomics 12*, 1 (2011), 444.

[43] Wu, Y.-W., and Ye, Y. A novel abundance-based algorithm for binning metagenomic sequences using l-tuples. *Journal of Computational Biology 18*, 3 (2011), 523–534.

[44] Yang, B., Peng, Y., Leung, H., Yiu, S.-M., Qin, J., Li, R., and Chin, F. Y. Metacluster: unsupervised binning of environmental genomic fragments and taxonomic annotation. In *Proceedings of the First ACM International Conference on Bioinformatics and Computational Biology* (2010), ACM, pp. 170–179.

[45] Yang, B., Peng, Y., Leung, H. C., Yiu, S.-M., Chen, J.-C., and Chin, F. Y. Unsupervised binning of environmental genomic fragments based on an error robust selection of l-mers. *BMC bioinformatics 11*, Suppl 2 (2010), S5.

[46] Zhang, W., Chen, J., Yang, Y., Tang, Y., Shang, J., and Shen, B. A practical comparison of de novo genome assembly software tools for next-generation sequencing technologies. *PloS one 6*, 3 (2011), e17915.