Scuola di dottorato
Università degli Studi di Milano-Bicocca

Dipartimento di Informatica Sistemistica e Comunicazione – DISCo
PhD program in Computer Science – Cycle XXIX

Self-indexing for de novo assembly

PhD Thesis

Marco Previtali
704496

Advisors: Prof. Paola Bonizzoni
Prof. Gianluca Della Vedova

Tutor: Prof. Arcelli Fontana Francesca
Coordinator: Prof. Stefania Bandini

Academic Year 2015–2016

Acknowledgements

There are several people that I would like to thank for their support thorough the years that helped me to get to the end of this journey.

First and foremost, I would like to thank my parents and my family who always supported me no matter what and whose help has always been invaluable.

I would also like to thank my advisors, Paola and Gianluca, for their guidance, assistance, and encouragement and for showing me how to approach research, a completely new field for me. I'm also grateful to the AlgoLab research group's current and past members — Raffaella, Riccardo, Murray, Anna Paola, Luca D, and Mauricio — for the insightful suggestions and discussions and for making this period extremely enjoyable.

Special thanks go to Yuri, Stefano, Luca M, and Simone for sharing ideas, laughs, good and not-so-good moments, office space, and for cheering me up.

During my PhD I had the pleasure to spend some months abroad. In two different occasions I visited Prof. Veli Mäkinen's Genome-Scale Algorithmics in Helsinki where I met an extraordinaire group of researchers who helped me becoming a better one. Veli, Travis, Simon, Djamal, Alexandru, Leena, and Daniel: kiitos!

I also had the opportunity to visit Prof. Paul Medvedev's group at the Pennsylvania State University and I would like to thank all the group members — Paul, Kristoffer, Rahul, Chen, Ilia, Mayank, Bob — for hosting me.

Last but not least, I would like to thank Eliana who always reminds me what matters the most.

Contents

1 Introduction **7**

2 Preliminaries **15**

2.1 Computational complexity 15
2.2 Strings and Indexing Data Structures 16
2.3 Bioinformatics and Computational Biology 23
2.4 DNA and sequencing methods 24
2.5 Assembly Problem 27

I Indexing data structures for assembly graphs **33**

3 De Bruijn graphs representations **37**

4 Fully dynamic succinct de Bruijn graphs **43**

4.1 Static de Bruijn Graphs representation 44
4.2 Dynamic de Bruijn Graphs representation 47
4.3 Applications 50

5 Bidirectional succinct de Bruijn graphs **53**

5.1 Fixed- and variable-order `BOSS` 53
5.2 Bidirectional BOSS 55
5.3 Applications 63

II Building assembly graphs from self-indexes 65

6 Self-index based assembly framework 71

6.1 Preliminaries 71

7 Lightweight external memory assembly algorithm 81

7.1 Background 82

7.1.1 Definitions 83

7.2 Methods 85

7.2.1 Algorithm Engineering 91

7.3 Results and discussion 93

7.4 Conclusions and possible extensions 97

8 Parallel assembly algorithm 101

8.1 Definitions 102

8.2 Methods 103

8.2.1 Data representation 110

8.3 Results and discussion 111

8.4 Conclusions and possible extensions 114

A Appendix 117

A.1 Additional tables 117

Bibliography 123

1 Introduction

For at least three decades computer science approaches have proved to be of utmost importance for extrapolating knowledge from biological data. On one end the amount of data produced by sequencing technologies — Sanger [126], Next-Generation Sequencing, or Third Generation Sequencing — has increased more than exponentially [28,51,76,85], outpacing Moore's law [94] and making automatic analysis the only viable option. On the other end, computer science and electronic devices permeate our society and prove to be cost efficient and extremely helpful — if not essential — for scientific progress.

Raw biological data produced by sequencing technologies can easily be represented using computer science techniques. Indeed, for many applications, DNA and RNA can be represented as one-dimensional sequences of *nucleobases* (adenine, cytosine, guanine, and either thymine for DNA and uracil for RNA) that can be represented as a word which characters belong to an extremely small alphabet (`A`, `C`, `G`, and either `T` or `U`). The representation, analysis, and storage of words and texts is a widely studied field in computer science in which sequences of characters are usually referred as *strings*. Close collaboration between biologists and computer scientists is unavoidable and extremely beneficial for both fields. Indeed, on one end the former pose computational challenges to the latter stimulating and motivating research in the field whereas, on the other end, the latter help the former to extract biological meaning from a huge and unfathomable muddle of data.

This thesis proposes computer science methods for the representation, storage, and analysis of huge datasets of strings with applications in bioinformatics. Such datasets can require hundreds of GB of memory to be stored and therefore the design of efficient algorithms and data structures aimed to their analysis is not trivial. The main goal of this thesis is therefore to provide algorithms and data structures for the analysis of such huge amount of data, providing theoretical analysis of their performances, implementations

of them, and extensive experimental evaluation of the tools proposed.

One of the main problems tackled by computer scientists in this field is the *Sequence Assembly Problem* (SAP) which goal is to infer the original DNA or RNA sequence S from a set of much more shorter substrings of S, usually called *reads*. SAP can be further divided into two different problems, namely assembly *with reference* and *de novo* assembly. In the former a draft genome of the species analyzed is available, that is we have access to a genome highly similar to the one of the individual under analysis, whereas in the latter this information is not available. The presence or absence of such information leads to different formulations and solutions of SAP. More precisely, in assembly with reference the goal is to map the reads to the most probable locus from which they have been extracted whereas in de novo assembly the goal is to build a chain of reads that better represents the unknown original sequence. How to solve both problems is a widely studied yet lively field of research that crosses different computer science sub-fields such as stringology, algorithms, data structures, information theory, and software development to name a few.

Apart from pure scientific pleasure, de novo assembly's first driving force was the Human Genome Project (HGP), an extremely influential and well known scientific project of the last decade of the last millennium, whose goals were to determine the sequence of the humane genome, identify the genes of it, store the information acquired for public use, and transfer the developed technologies to the private sector [106]. Although HGP was successful, its cost was estimated to be between 2.7 and 3 billion dollars [66], it took ten years to complete, and it finally published the sequence of human genome at the turn of the millennium [78, 141]. Such tremendous costs made the procedures and technologies proposed by HGP hard to replicate in an everyday research setting, and therefore, a strong push to limit the cost of sequencing technologies began around the same time the first draft of the human genome was published.

By 2005 new sequencing technologies started to appear in the market with extremely interesting costs and performances. Indeed, with the advent of such technologies (Next-Generation Sequencing — NGS —) it became possible to sequence the genome of a given individual for approximately 10 million dollars [66], a two order of magnitude decrease with respect to HGP, and in a matter of days. The last ten years have witnessed a race

to the so-called $1000 genome, *i.e.*, sequencing a genome for only $1000, breaking such barrier in the last couple of years, making the sequencing process an affordable task for pretty much any research group, and enabling the usage of such technologies as clinical tools [48].

The astonishing cost drop of sequencing experiments and the amount of data produced by them (hundreds of GB of raw data per experiment) had consequences on SAP techniques too. Indeed, the throughput of sequencing experiments increased from mere millions of reads to billions of reads per run [56] making the dimension of the datasets almost unmanageable. Therefore, clever ways to efficiently store, represent, and query the data is, nowadays, mandatory.

Before discussing the methods proposed in this thesis we need to take a step back and understand what is our focus in de novo assembly. Almost every current de novo assembly method is based on a graph that represents the layout of the reads in the final assembly. More precisely, at the core of most of the assemblers lies a graph data structure that represents some kind of relation between reads or part of them. The two most well known and used graphs in this field are de Bruijn graphs [30] and Overlap graphs [97] (and their reduced form, String graph). The former is widely used — e.g. the well known assembler Velvet [144] uses them — and are tied to a value K, usually called their order. In these graphs nodes represent sequences of length K that appears in the reads and nodes are connected if the labels of the nodes shares as prefix or as suffix $K-1$ characters. De Bruijn graphs present an extremely nice and powerful formalism that was applied in multiple fields prior to their usage in bioinformatics and, hence, have a strong theoretical background. Nevertheless, one of the shortcomings of de Bruijn graphs is that they usually do not consider the whole information contained in the reads since they split them in smaller parts, increasing the consequences of short repetitions in the genome in the final assembly. The latter, String graphs, are in use since the dawn of assemblers [99] and have been later formalized in 2005 by Myers [97]. In these graphs nodes represent reads and nodes are connected if the reads related to them overlap, *i.e.*, if they share a prefix and a suffix. String graphs, therefore, avoid the problems of de Bruijn graphs since they do not split the reads but, on the other end, they require more time to be computed and more space to be stored. As for now, it is not completely clear what

are the advantages and disadvantages of one approach with respect to the other and, hence, both formalisms are used. Empirical analysis of the outcomes of the tools (e.g., the Assemblathon effort [33]) using both approaches suggest that using String graphs leads to outputs more faithful to the original genome but these results might be due to other steps of the pipelines analyzed (error correction, tip removal, bubble popping, scaffolding) and, therefore, it is still interesting to investigate de Bruijn graphs-based approaches.

This thesis focuses on this field and, more precisely, proposes methods for representing, building, and analyzing both String graphs and de Bruijn graphs. The contributions of our work can be divided in two directions that, although closely related, have slightly different goals and motivations. In particular, we will propose: (i) methods to store and query de Bruijn graphs efficiently and (ii) methods to analyze String graphs using succinct, *i.e.*, close to the theoretical space lower bound, representation of them. The contributions of this thesis intersect significantly with the study and development of the so-called indexes and self-indexes, fundamental and widely known tools used in computer science. Indexes are data structures built on top of a dataset — usually a text — that facilitate the execution of particular queries over it. Given a text T and a pattern P, common queries which performances are improved by indexes are `Find`, that is the process of deciding whether P appears in T, `Count`, that is the process of counting how many times P appears in T, and `Locate`, that is the process of detecting the starting positions of P in T. Indexes have found use in computer science for more than 40 years, starting with Suffix Trees [142] and eventually converging to Suffix Arrays [88], and are a well established and studied field. Self-indexes are *evolutions* of indexes; using this approach it is not required to store the original dataset but the whole information is store in them, allowing for a better level of compression of the data. An example of self-index is the well-known FM-index [41], an extremely efficient data structure that can store a text in optimal space by means of the Burrows-Wheeler Transform [19] (BWT) and that allows for fast `Find`, `Count`, and `Locate` queries.

In this thesis we will present two methods for storing de Bruijn graphs. Storing de Bruijn graphs can be done trivially either by storing a plain graph or by using some fast-access hash table but these approaches may require too much space to store graphs

that originate from a whole genome sequencing experiment due to the sheer amount of data produced. Therefore, efficient methods for storing, accessing, and querying the graphs are used. The literature presents some interesting approaches and, between them, the most efficient and interesting ones are the one based on Bloom filters [22] — a probabilistic membership data structure — and the one based on the Burrows-Wheeler Transform [14]. These two approaches were proposed simultaneously in 2012 and have different properties. The first one, indeed, is an extremely compact representation of de Bruijn graphs that is semi-dynamic in the sense that it is possible to add nodes to the graph but it is not possible to remove nodes without rebuilding the data structure from scratch. The second one, along with its extension proposed in 2015 [13], allows to store multiple de Bruijn graphs for the same dataset with a succinct data structure but allows to explore the graph only in one direction, *i.e.*, only visiting edges outgoing from the nodes, that can be seen as a limitation in some application.

Our first contribution in this direction is a new self-index for a de Bruijn graph based on Minimal Perfect Hashing. The main contribution of this method is a data structure that represents a de Bruijn graph in succinct space and allows to add and remove both edges and nodes without requiring to rebuild the whole data structure from scratch. Dynamic graphs can have different applications in bioinformatics and could be used in error correction tools for building graphs representing the whole data set and then removing edges and nodes scarcely supported by the data without requiring an excessive amount of memory nor time. We present a theoretical analysis of this method and provide a formal analysis of the properties of it.

The second contribution in this direction is a new self-index for de Bruijn graphs that removes the limitation of the one presented in [13], *i.e.*, it is able to represent a set of de Bruijn graphs for different orders and allows to test and visit both outgoing and incoming edges from a node. We present a theoretical analysis of this method and provide possible applications of it in genome assembly.

In this thesis we will also focus on efficient methods for analyzing and constructing String graphs and the relations of such graphs with the Burrows-Wheeler Transform. Previous works proposed in the literature (e.g. SGA [135]) showed that it was possible and efficient to use the BWT to build String graphs, highlighting strong relations between

these graphs and the FM-index.

In this thesis we will formalize a new framework for String graph-based assemblers in which String graphs nodes and edges are represented as intervals of the BWT requiring constant space to be stored. We will therefore present a theoretical analysis of such framework showing interesting connection between properties of the arcs of the graph and properties of portions of the BWT of the dataset that can be exploited in genome assembly and in particular in constructing the String graph of a dataset.

We will then study and show how this framework can be used to design String graph construction steps that either require an extremely low amount of main memory or an extremely low amount of time by external memory algorithms and usage of threads and careful operations organization, respectively. We will also provide implementations of both these approaches and perform an extensive experimental evaluation of the tools in comparison with state-of-the-art assemblers, such as SGA [134]. The experimental evaluation shows that the framework we propose and its implementations allow to design tools that can run efficiently either on simple machines or on extremely powerful server showing the pliability of our contribution.

We note that, once personalized medicine will be available, it will make sense to run assemblers even on not-so-powerful workstation, but most of the current tools require at least a mid-range server to complete such task. In our opinion, proposing tools that can work with a limited amount of main memory make sense in this direction.

The thesis is structured as follows. In Chapter 2 we review some basic notions computational complexity, indexing data structures, bioinformatics and computational biology, and the state-of-the-art of the assembly problem. The thesis is then divided in two main part: the first one presents methods for efficiently store sequencing data for the de novo assembly problem whereas the second one presents efficient algorithms for analyzing succinct data structures for the same problem. More precisely, in PartI we will present two data structures for representing and querying de Bruijn graphs. Although both data structures represent de Bruijn graphs, they have different purposes. In Chapter 4 we will present the first one which goal is to represent a de Bruijn graph in succinct space so that you can add and remove both edges and nodes. To the best of our knowledge, this is the first fully dynamic and succinct representation of these graphs. In Chap-

ter 5 we will present the second representation which goal is to represent multiple de Bruijn graphs of the same data set for various values of K in succinct space allowing free exploration of it. We will also point out possible application in bioinformatics — in particular in genome assembly — where such data structure could be useful. In Part II of the thesis we will present two new algorithms for computing a de novo assembly of a data set based on String graphs. In this part we will show that we can analyze and reduce these graphs by representing them using the so-called self-indexes of the reads, *i.e.*, representation of the information that allows for fast queries and, usually, can be compressed close to the theoretical lower bound. In Chapter 6 we will therefore show and proof strict relations between String graphs and the Burrows-Wheeler Transform. Moreover, in Chapter 7 and Chapter 8 we will show that these relations can be exploited to implement, respectively, external memory algorithms and parallel algorithms for the assembly problem.

2 Preliminaries

This chapter is devoted to the formal definition of the prerequisites that will be used through the rest of the thesis. In particular, this chapter will present notions on *computational complexity*, *indexing data structures*, *bioinformatics* and *computational biology*, and the *de novo assembly problem*. Note that we defer the definition of some concepts to the next sections, when closely related to the results of a single part of the thesis.

2.1 Computational complexity

Computational complexity theory is the study and the classification of computational problems. Borrowing the terminology from [3], a problem can be formalized as follows.

Definition 2.1 *(Problem) A* problem *is a mathematical relation $P \subseteq I \times S$ between a set I of* instances *and a set S of* solutions*.*

Given a problem $p \in P$, the set of acceptable solutions for an instance $i \in I$ is the set $\{s : (i, s) \in P\}$ and we call such set the *feasible solutions*. When the solution of a problem is binary, *i.e.*, it belongs to the set $\{\texttt{TRUE}, \texttt{FALSE}\}$, we say that the problem is a *decision* problem. In other cases we might want to rank the solutions of a problem based on *"how good"* they are with respect to a quality measure $c : S \to \mathbb{R}$ and either want to find the solution that *minimizes* or *maximizes* the quality measure. We call such problems *optimization problems* and, for each instance $i \in I$, $s^\star \in S$ is the *optimal solution* if $(i, s^\star) \in P$ and either $s^\star = \arg\min_{s \in S}\{c(s) : (i, s) \in P\}$ for minimization problems or $s^\star = \arg\max_{s \in S}\{c(s) : (i, s) \in P\}$ otherwise. The quality measure $c(s)$ of a feasible solution s is called *cost* of the solution, and the cost of an optimal solution is called *optimum* of the instance.

An algorithm is a description of a procedure for computing a feasible solution of a problem in a finite number of steps using a mathematical model of computation. Multiple algorithms can solve the same problem, and comparing the performances of different algorithms to find the *best* one (for some definition of *best*) has a central role in computer science. Algorithms can be compared using different parameters, *time* required to compute the solution being one of the most used. When performing such comparison, time is not stated as number of seconds spent in order to compute the solution but, instead, as the number of elementary steps (single value assignments, value comparison, arithmetic operations, etc.). Since different instances of a problem require different times, algorithms are usually ranked in terms of *worst-case* time complexity. Given a problem and an algorithm solving it, the worst-case time complexity of the algorithm is a function $f : \mathbb{N} \rightarrow \mathbb{R}$ that specifies the maximum number of steps $f(n)$ that the algorithm requires to solve an instance of size n.

The worst-case time complexity is expressed using the so-called *big-O notation*, usually represented as $\mathcal{O}(t(n))$ where $t(n)$ is a function in n. Given an algorithm, we say that its worst-time complexity is $\mathcal{O}(t(n))$ if $t(n)$ is an asymptotic upper-bound of $f(n)$ when n — the size of the instance — grows to infinity. We say that an algorithm is a *polynomial algorithm* if its worst-time complexity function $t(n)$ can be represented as n^k for some constant k, we say that it is a *linear algorithm* if $t(n)$ can be represented as kn, and we say that it is an *exponential algorithm* if $t(n)$ can be represented as k^n. In the following chapters we will refer to this analysis as time complexity analysis, omitting the worst-case term since most of the time it is implied.

2.2 Strings and Indexing Data Structures

We begin this section by providing a formal definition that will be fundamental in this thesis.

Definition 2.2 *(String) Let Σ be an ordered finite alphabet of size σ. A* string *T is a finite sequence of characters $c_1, c_2, \ldots, c_n$ drawn from Σ. We denote by $T[i]$ the i-th symbol of T and by $|T|$ the length of T.*

Strings are extremely well studied data structures in computer science since they are

used to represent one of the most widely used mean of transmission of information: texts. We now give some additional fundamental definitions that we will use later.

Definition 2.3 *(Sub-string, Suffix, Prefix) Let Σ be an ordered finite alphabet of size σ, and let $T = c_1, c_2, \ldots, c_n$ be a string over it.*

Let i and j be two integers such that $1 \leq i \leq j \leq |T|$, we say that the sequence $c_i, c_{i+1}, \ldots, c_j$ is the sub-string of T starting in i and ending in j and we denote it by $T[i : j]$.

The suffix *and* prefix *of T of length K are respectively the sub-strings $T[|T| - K + 1 : |T|]$, denoted by $T[|T| - K + 1 :]$, and $T[1 : K]$, denoted by $T[: K]$.*

In the following chapters we will use two functions, namely `rank` and `select`, that are fundamental in the field this thesis is focused on. Both functions can be defined over *sequences* of elements but we report their definition over strings since we will use them in this case.

Definition 2.4 *(`rank` and `select`) Let T be a string over an alphabet Σ.*

`rank` is a function $\Sigma \times \mathbb{I} \rightarrow \mathbb{I}$ that, given a character $c \in \Sigma$ and an integer p such that $1 \leq p \leq |T|$, returns the number of occurrences of c in the prefix of length $p - 1$ of T.

`select` is a function $\Sigma \times \mathbb{I} \rightarrow \mathbb{I}$ that, given a character $c \in \Sigma$ and an integer i, returns the position p in T where the i-th occurrence of c appears. If i is greater than the number of occurrences of c in T, `select` is undefined.

One of the main problems with strings is the string searching problem. In this problem, given two strings T and P called the text and the pattern respectively, we want to know if, how many times, and where, P appears as a sub-string of T. We can formalize the three goals as follows.

Find: (also called **Exists**) a decision problem that returns `TRUE` if P is in T.

Count: a computational problem that returns the number of occurrences of P in T.

Locate: (also called **Report**) a computational problem that returns the set of positions in which P occurs in T.

There are different classic algorithms that solve this problems, the most known ones due to Knuth–Morris–Pratt [72], Karp–Rabin [68], and Boyer–More [15]. Although the first one is optimal if we want to search for a single pattern in the text, *i.e.*, it requires $\mathcal{O}(|T| + |P|)$ time, they are not suited to search multiple patterns in the same static text since they would require $\mathcal{O}(m(|T| + P^\star))$, where m is the number of the patterns and $P^\star$ is the average length of them, forcing us to analyze the text multiple times

Searching for patterns in a static text is a usual task in computer science, e.g., we can consider a book, a dictionary, a data set, or — to some extent — the web to be static texts. To overcome the limitations of searching multiple patterns in the same text, the concept of *index* was proposed.

Definition 2.5 *(Index) An index is a data structure built "on top" of the text that allows to search efficiently for patterns without scanning the whole text.*

An index is therefore a bit of information we add to the text in order to perform search queries efficiently. It is worth to note that the text has to be maintained in this case and that's why we say that the index is build "on top" of it. In some cases it is possible to design data structures that replace the text itself providing the same features of an index; in this case we call the data structure a *self-index* [103].

Definition 2.6 *(Self-index) A self-index is a data structure that provides the same features of an index but does not require to access or store the original text.*

The literature presents a plethora of indexes and self-indexes; in this section we will present the most prevalent ones, namely Suffix Trees, Suffix Arrays, and FM-index. For technical reasons, in all of these approaches, a single special character \$ — lexicographically smaller than all the characters in Σ — called the sentinel symbol is appended at the end of the text. This is required because most of these approaches are based on considering the set of the suffixes of the text and, in order to have a nice and functional formalism, we have to avoid suffixes that are prefixes of other suffixes. By concatenating \$ at the end of the string we are sure that no suffix will be a prefix of another one since \$ will appear only once. We will now devote a paragraph for each of the indexes cited before.

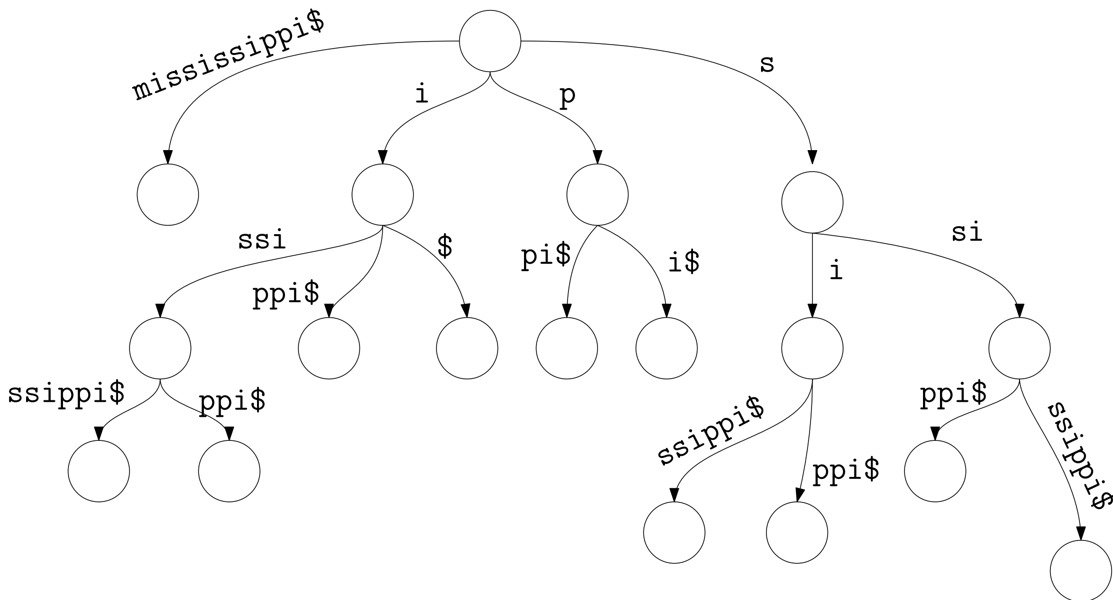


Figure 2.1: An example of a suffix tree for the text `mississippi`.

Suffix Trees Suffix trees were one of the first indexes for texts and were introduced by Weiner in 1973 [142]. It is easier to describe suffix trees as a contraction of another data structure called *Tries* [44] and we will therefore define them first.

Definition 2.7 *(Trie) Let S be a set of strings. The trie of S is a tree in which:*

- *there exits a node for each distinct prefix in S;*
- *the root is the node of the empty string ϵ;*
- *two nodes v_i and v_j are connected by a directed edge if and only if there exists a string X and a character c such that the prefix related to the former is X and the prefix related to the latter is Xc.*

If t is the trie built over the set of suffixes of a text T, we call t a *Suffix Trie*. Suffix tries can clearly contain simple paths (*i.e.*, non-branching paths), contracting such paths produces the so-called *Suffix Tree*. We can now present a formal definition of Suffix Trees.

Definition 2.8 *(Suffix Tree) Let T be a text, let S be the set of suffixes of T, and let t be the Trie of S. Let $t^\star$ be a copy of t in which we we replace simple paths by a single edge between the first node and the last of the path. We call $t^\star$ a Suffix Tree.*

GSA	Rotations	BWT
(5,0)	$missi	i
(6,1)	$ssippi	i
(4,0)	i$miss	s
(5,1)	i$ssipp	p
(1,0)	issi$m	m
(2,1)	issi$ss	s
(0,0)	missi$	$
(4,1)	pi$ssip	p
(3,1)	ppi$ssi	i
(3,0)	si$mis	s
(1,1)	sippi$s	s
(2,0)	ssi$mi	i
(0,1)	ssippi$	$

Figure 2.2: An example of Generalized Suffix Array and BWT for the texts `missi` and `ssippi`.

Suffix Trees are simple, elegant, and efficient data structures for storing dictionaries and performing search queries over them. A Suffix Tree can be built in $\mathcal{O}(|T|)$ time using $\mathcal{O}(|T| \lg |T|)$ bits of space [38, 91, 140], solves `find` and `count` queries in $\mathcal{O}(|P|)$, and `locate` queries in $\mathcal{O}(|P| + \texttt{occ})$ where `occ` is the number of occurrences of P. One of the biggest drawbacks of Suffix Tress is the space required to store them, up to $12|T|$ bytes even with careful implementations [49, 50], and practical implementation performances are dependent on the alphabet size. An example of the Suffix Tree of the text `mississippi` is shown in Figure 2.1.

Suffix Arrays To lower the amount of space required to store the Suffix Tree, Manber and Myers proposed in 1990 a new data structure called Suffix Array [88]. This data structure is a reduced form of the suffix tree that represents only the leaves of it, via pointers to the starting positions of all suffixes [103], requiring only $4|T|$ bytes to store the index. Different extensions of the Suffix Array have been proposed —- e.g. Generalized

Suffix Arrays [133], Enhanced Suffix Arrays [1], Compressed Suffix Arrays [59, 60, 121], and Dynamic Suffix Arrays [46, 124]— in this section we will consider and present the Generalized Suffix Array (GSA) that we will use in the next chapters of the thesis. Such data structure indexes a set of strings $T = \{s_1, \ldots, s_m\}$ and allows to search in all of them at the same time. Suffix Arrays, much like the Suffix Trees, are based on the set of suffixes of a text or on the set of rotations of the texts. For simplicity, we will now present their definition based on the rotations of the texts and thus will now introduce such concept.

Definition 2.9 *(Rotation) Let T be a text, we say that its i-th rotation is the concatenation of the suffix of length i of T (containing \$) and its prefix of length $|T| - i$.*

Note that a simple string T has exactly $|T|$ distinct rotations and none of them can be represented by the same string since the sentinel symbol \$ appears only once. We can now present the definition of Generalized Suffix Array.

Definition 2.10 *(Generalized Suffix Array) Let T be a set of strings terminated with \$ and let RT be the lexicographic ordered array of all the rotations of all the strings in T. The* Generalized Suffix Array (GSA) *of T is the array SA where each element $SA[i]$ is equal to (k, j) if and only if the k-rotation of string r_j is the i-th element of RT.*

In other words, the Generalized Suffix Array is a permutations of the suffixes of the texts of T such that they are lexicographically sorted.

Suffix Arrays are elegant and efficient data structures for searching through a text. A Suffix Array can be built in $\mathcal{O}(|T|)$ time [70], solves `find` and `count` queries in $\mathcal{O}(|P| \lg |T|)$ time, and `locate` queries in $\mathcal{O}(|P| \lg |T| + \texttt{occ})$. One of the major drawbacks of Suffix Arrays is that there is usually a significant amount of redundancy in them since big chunks of the array are copies of each other with values shifted by 1. Compressed Suffix Arrays [59, 60, 121] aims to reduce this redundancy and are able to lower the space required down to $2|T|$ bytes on top of the text at the cost of doubling the query time. Nevertheless, for some applications $2|T|$ extra bytes is still too much. An example of the Generalized Suffix Array for the texts `missi` and `ssippi` is shown in Figure 2.2.

FM-index To lower the memory requirements of Suffix Arrays, in 2000 Ferragina and Manzini [41] proposed a new (self) index which space occupancy is a function of the entropy of the underlying data set called the FM-index. The FM-index is a self-index that performs queries over the text by analyzing a permutation of the characters in it called the *Burrows-Wheeler Transform* [19] (BWT). BWT was firstly presented in 1994 for data compression purposes and is closely related to the Suffix Array data structure. We can formalize its definition as follows.

Definition 2.11 *(Burrows-Wheeler Transform) Let T be a set of strings terminated with \$ and let RT be the lexicographic ordered array of all the rotations of all the strings in T. The Burrows-Wheeler Transform of T is the sequence B such that $B[i] = r_j[|r_j|-k]$ if $SA[i] = (k, j)$ and $k < |r_j|$, or $B[i] = \$$ otherwise. Less formally, the BWT of T is the sequence B such that $B[i]$ is the last character of $RT[i]$.*

The Burrows-Wheeler Transform of a text T can usually be compressed better than the text itself since it groups together characters that appear before the same substring. Since texts usually follow grammatical rules, this grouping strategy leads to producing longer runs of characters that can be better compressed using state-of-the-art techniques [64,119,139]. It is easy to demonstrate that the Burrows-Wheeler Transform of a text can be stored in optimal space [41], equivalent to the empirical order-K entropy of the text [131,132], referred as $|T|H_K(T)$. Although BWT was presented more than twenty years ago, it is still a central tool in some compression software, like `bzip2` [130]. BWT alone was not intended as an index, indeed it lacks all the additional structures for computing efficiently `find`, `count`, and `locate` queries.

The FM-index closes this gap by adding two additional functions to it, namely $\texttt{C} : \Sigma \to \mathbb{I}$ and $\texttt{Occ} : \Sigma \times \mathbb{I} \to \mathbb{I}$. The first one, `C`, counts for each character c_i the occurrences of all the characters c_j smaller than it in the BWT and can easily be represented by an array of size σ that requires $\sigma \lg |T|$ bits. The second one, `Occ`, counts the occurrences of a given character $c \in \Sigma$ up to a given position i. Storing such table in plain form for constant time access would require $\sigma|T| \lg \|T|$ bytes and would be infeasible even for medium-sized texts. To overcome this limitation, such function is usually computed by mean of a Wavelet Tree, a tree data structure initially proposed to represent Compressed Suffix

Arrays [58] and later adapted to BWTs [42]. In brief, a Wavelet Tree is a tree shaped representation of the characters in a string based on bitvectors that "split" the string based on the lexicographic order of the characters that requires $|T|H_0(|T|) + o(|T|\sigma)$ bits to be stored. Since counting bits up to a given position in bitvectors requires constant time [104] and since a Wavelet Tree has as most $\lg \sigma$ levels, counting occurrences of a characters up to a position in a string represented as a Wavelet Tree requires $\mathcal{O}(\lg \sigma)$ time.

By means of these two additional data structures, the FM-index allows to search for a pattern P in a text T represented as a Burrows-Wheeler Transform in $\mathcal{O}(|P| \lg \sigma)$ time. We can summarize what an FM-index is in the following definition and defer a more in-depth discussion of the search procedure to Chapter 6.

Definition 2.12 *(FM-index) Let T be a set of strings terminated with \$ and let B be the Burrows-Wheeler Transform of the text. The FM-index of T is the union of B and two functions,* `C` *and* `Occ`*. The first one, given a character c_i, returns the number of characters smaller than it in B whereas the second one, given a character c_i and a position j returns the number of occurrences of c_i in B[1:j].*

For an in-depth analysis of Wavelet Trees we refer the reader to [102] and references therein. An example of the BWT of the text `mississippi` is shown in Figure 2.2.

2.3 Bioinformatics and Computational Biology

Bioinformatics and Computational biology are two research fields in which computer scientists and biologists collaborate to extract knowledge from biological data using computational methods. There is a partial overlap between them and the boundary between the two is, most of the time, blurry. In 2000 [63] the BISTIC Definition Committee released a formal definition of the two fields that we report here.

Definition 2.13 *(Bioinformatics) is* "Research, development, or application of computational tools and approaches for expanding the use of biological, medical, behavioral or health data, including those to acquire, store, organize, archive, analyze, or visualize such data."

Definition 2.14 *(Computational Biology) is* "The development and application of data-analytical and theoretical methods, mathematical modeling and computational simulation techniques to the study of biological, behavioral, and social systems."

This thesis focuses on the computer science side of bioinformatics and computational biology. More precisely, the main focus of this thesis is on algorithms and data structures for the de novo assembly problem and, thus, it is probably closer to bioinformatics than it is to computational biology. Nevertheless, as stated above, the boundary between the two fields is blurry and we reported both definitions for sake of completeness.

2.4 DNA and sequencing methods

As defined by the *Talking Glossary of Genetic Terms* of the National Institutes of Health [101], DNA (Deoxyribonucleic Acid) is the chemical name for the molecule that carries genetic instruction in a living things. It consists of two strands that wind around one another to form a shape known as a double helix, and each strand is an alternating sequence of sugar and phosphate groups. Attached to each sugar is one of four bases (also called base pairs in the context of this thesis): Adenine (`A`), Cytosine (`C`), Guanine (`G`), and Thymine (`T`).

The process of determining the exact sequence of bases in a DNA molecule is called DNA sequencing. It is worth to note immediately that, as for now, DNA cannot be sequenced in its entirety and only short sub-sequences of the molecule can be determined each time. The genome of a given individual is therefore sequenced small-piece-by-small-piece and the sequencing process produces as output a set of *reads*. During the last 40 years, multiple sequencing methods with different characteristics were developed; in this section we will give a brief overview of the different sequencing methods.

For a more thorough discussion of sequencing methods, we refer the reader to the following reviews [28, 51, 56, 76, 85].

Sanger sequencing In 1977, the first method for DNA sequencing was developed by Frederick Sanger [126] and was named after him. For 30 years Sanger sequencing was the only viable and used option to sequence genomes, it was one of the breakthrough

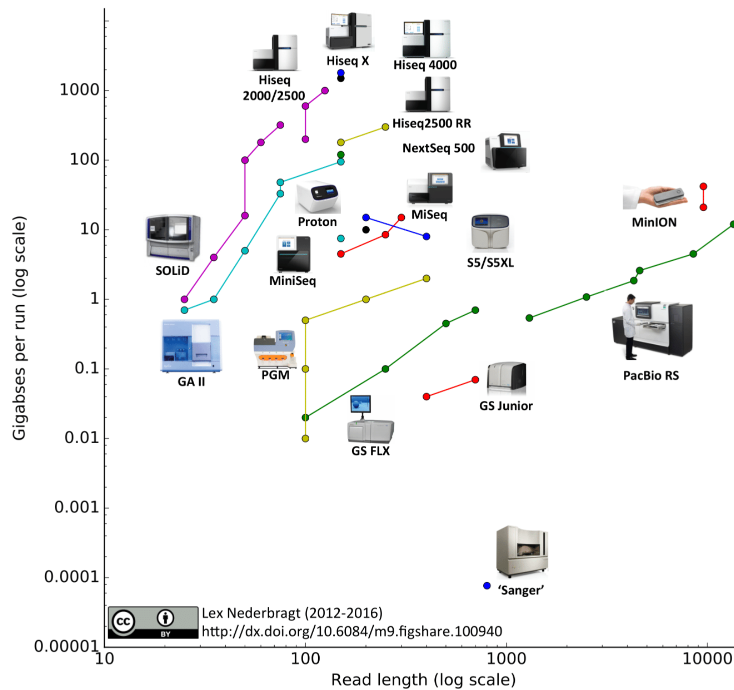


Figure 2.3: Overview of the current sequencing platforms ranked by data produced per run and length of reads produced. From [105].

discovery of the century since it helped, for the first time, to analyze and reconstruct the sequence of the human genome [78, 141].

Sanger sequencing comes with both advantages and disadvantages. On one end, this method produces the so-called *long* reads, of length up to ∼1000 base pairs, and has a low error rate, estimated to be between 10^{-4} and 10^{-5} [71]. On the other end, Sanger methods are slow since the average output is approximately 6 MB of DNA sequence per day, and expensive since the chemical consumables' cost amounts to about $500 per MB of data. A widespread use of this technology is therefore not possible and, to overcome this limitation, in the last two decades new sequencing methods have been developed and proposed.

Next-Generation Sequencing (NGS) New methods designed to replace the expensive Sanger methods usually fall under the name of *Next-Generation Sequencing Technologies* (NGS). The main goal of such methods is to lower the cost of a single sequencing experiment and to increase the amount of data generated by it. The first of these new technologies appeared in 2005 [83] and immediately changed the genomic research as it enabled to obtain millions of reads, at a fraction of the cost of Sanger sequencing.

NGS sequencing comes with both advantages and disadvantages too, but the former outshine the latter. Indeed, the biggest disadvantage is that the sub-sequences of the genome produced by these methods are shorter than the ones produced by the previous technologies. Such reads go by the name of *short* reads (in opposition to the long reads produced by Sanger methods), are usually in the range of 35 to 400 base pairs and have a higher error rate (betwen 0.1% and 1%). On the other end, NGS are extremely fast, cheap, and reliable [56], and has thus enabled the use of sequencing as a clinical tool [48]. Using sequencing as clinical tool is regarded by most as the next big step for medicine, unlocking the possibility to design drugs that can adapt to the individual will change healthcare in the next decades.

There are different companies producing of NGS machines (Illumina, Ion Torrent, SOLiD, 454); a thorough analysis of the options and the characteristics of each machine is out of the scope of this thesis and we refer the reader to the reviews in literature [28, 51, 56, 76, 85] and to Table A1 in the Appendix.

Third-Generation Sequencing Progress in sequencing didn't stop with NGS. Indeed, although affordable and disruptive, NGS technologies come at a cost, most notably the fact that extremely short reads cannot disambiguate repeats bigger than them in the sequenced genome. To tackle this limitation, since 2011 new technologies, usually referred as Third-Generation Sequencing or Future-Generation Sequencing, have been proposed.

As their predecessors, these new methods, which most notable implementations are PacBio [116] and Oxford Nanopore [35, 93], have both advantages and disadvantages. The most notable advantage of this technologies is the length of the reads produced which are usually longer than 10000 bp and can reach even 200000 bp. On the other

end, Third-Generation Sequencing technologies produce reads with much higher error rate, between 10% and 15% [56].

The combination of these two characteristics makes the analysis of such data challenging and it is thus common practice nowadays to use them in combination with NGS technologies, either by correcting the error-prone long reads using the more accurate short reads [123] or using hybrid approaches [87].

2.5 Assembly Problem

Between the problems bioinformatics faces, inferring the DNA sequence of an individual from the set of sub-sequences produced by the sequencing technologies is central. Such problem is one of the first ever considered in the field and is called the *Sequence Assembly Problem* (SAP).

SAP can be further divided into two sub-problems that differ by the information available, namely assembly *with reference* and *de novo* assembly. The main difference between the two is that in the former a draft genome of the species analyzed is available whereas in the latter only the data produced by the sequencing machines is. This leads to different formulations and solutions of SAP. More precisely, in assembly with reference (`AR`) the goal is to map each read to the most probable locus from which they have been extracted whereas in de novo assembly (`DA`) the goal is to build a chain of reads that better represents the unknown original sequence. Thus, although solving the same problem, `AR` and `DA` approaches are deeply different and this thesis focuses on the latter.

De novo Assembly From a computer science point of view, `DA` was initially approximated to the *Shortest Common Superstring Problem* (SSP) [69, 95, 98] that can be formalized as follows.

Definition 2.15 *(Shortest Common Superstring Problem) Given as input a set of strings* $T = \{s_1, \ldots, s_m\}$*, find the shortest sequence* $s' = c_1, \ldots, c_n$ *such that for each* s_k *in* T *there exists two integers* i *and* j*, with* $1 \leq i < j \leq n$*, such that* $s_k = s'[i : j]$.

SSP provides an elegant formal definition of SAP but it is known to be reducible to the traveling salesman problem, a problem known to be $\mathcal{NP}$-complete (see [25, Sec-

tion 34.5.4] or [29, pp.250]), and is therefore $\mathcal{NP}$-complete [120]. For a more thorough description of SSP we refer the reader to [137, Section 18.9].

To overcome this issue, many approaches to solve SSP (and, in turn, `DA`) aim to provide a near-optimal and approximated solution lowering the time complexity of the method to polynomial time. The most prominent approximate formulation of SSP for de novo assembly is the graph-based formulation which goal is to produce a graph in which a generalized Euler tour is the correct assembly. Finding an Euler tour of the graph can be performed in linear time [34, 37] and therefore it is possible to use this approach to produce the assembled genome.

It is worth to note that building the graph and extracting a single path in it is not the only step of a de novo sequence assembler. Indeed, errors, repetition in the structure of the genome, and read orientation play a major role in many assemblers currently available. Nevertheless, we can identify a number of steps shared by the majority of the approaches proposed in the literature. More precisely:

Error correction that is the process of removing artifacts from the sequencing data introduced by the sequencing machines.

Graph construction and contraction that is the process of constructing a graph representing the relations between the reads and compacting the simple paths (also known as unitig).

Tip removal and bubble popping that is the process of removing short unitigs (tips) branching out from the graph and short branching-and-joining paths. Tips are usually due to errors in the data set not corrected by the error correction step whereas bubbles can be due to errors not corrected or heterozygous loci in the individual under investigation.

Scaffolding that is the process of linking together non-contiguous sequences in the graph usually performed exploiting the mate-pair information.

All these steps together allow to produce reasonable results and to deal with the limitation of the technologies currently in use.

In this thesis we will focus on the graph construction, representation, and analysis, *i.e.*, the second step presented before. Two main type of graphs are used for `DA`, namely *de*

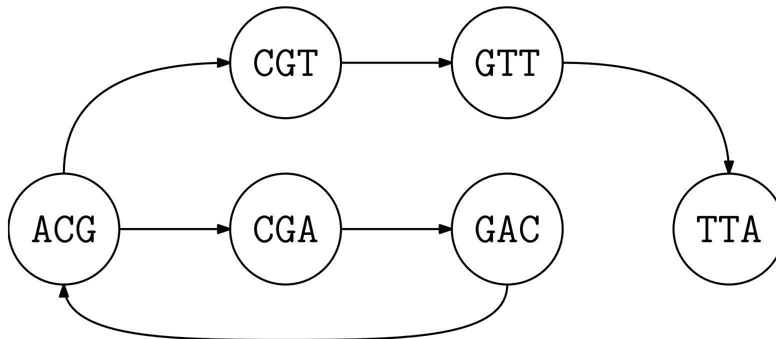


Figure 2.4: An example of a node centric de Bruijn graph of order 3 of the set of strings $T = \{\texttt{ACGT}, \texttt{CGAC}, \texttt{GTTA}\}$.

Bruijn graphs and *Overlap/String graphs*; we will now present both of them highlighting their differences.

The concept of de Bruijn graph was firstly introduced by de Bruijn [30] and Good [55] in 1946 as a mathematical concept. Their definition in our setting can be formalized as follows.

Definition 2.16 *(de Bruijn graphs) Let $T = \{s_1, \ldots, s_m\}$ be a set of strings and let K be a positive integer value. The de Bruijn graph of order K of T is a directed graph $G = (V, E)$ such that (i) for any distinct sub-string k_i of length K of any string $s_j \in T$ there exists a vertex v_i in V labeled with k_i and there exists a directed edge $e = (v_i, v_j)$ in E either if (ii-a) the $(K-1)$-long suffix of v_i is equal to the $(K-1)$-long prefix of v_j or (ii-b) property (ii-a) holds and there exists a string s_h in T such that the concatenation of the label of v_i and the last character of v_j is a sub-string of it.*

Depending on which property between (ii-a) and (ii-b) holds, de Bruijn graphs are referred with different names in computer science, namely *node-centric de Bruijn graphs* (`n-dBG`) and *edge-centric de Bruijn graphs* (`e-dBG`). We defer the discussion of their differences to the introduction of Part I in which we will present methods for representing these graphs. De Bruijn graphs are a powerful formalism that is used in fields far apart from bioinformatics, e.g. Cellular Automata Theory [90, 138].

A second formalism used to represent assembly graphs is the so-called Overlap/String graph. These graphs are in use since at least two decades in bioinformatics [96, 99] but are still central nowadays [134, 135]. Overlap graphs are graphs that represent the

overlap relation between the reads whereas String graphs are a reduction of the Overlap graphs in which redundant information is removed. We can formalize the definition of the first as follows.

Definition 2.17 *(Overlap graphs) Let $T = \{s_1, \ldots, s_m\}$ be a set of strings and let τ be a positive integer value. The overlap graph $G_O = (V, E)$ of T is a directed graph whose vertices are the strings in T, and there is an arc in E if and only if the strings represented by the two connected nodes share a prefix and a suffix of at least τ characters, i.e., if and only if they overlap by at least τ characters.*

The overlap graph contains redundant information. More precisely, the information provided by some arcs is not required to reconstruct the original genome of the dataset T. Indeed, if we consider three strings $s_1 = \alpha\beta\gamma$, $s_2 = \beta\gamma\delta$, and $s_3 = \gamma\delta\lambda$ such that all the factors have length at least τ, and build their overlap graph we can note that we will obtain three edges — (s_1, s_2), (s_2, s_3), and (s_1, s_3) — and one of them, (s_1, s_3), does not add any information to our graph since the path spelled by its traversal, *i.e.*, the genome represented by that path, is the same as the one represented by another path that starts in s_1, visits s_2, end ends in s_3. We call such uninformative arcs *reducible* or *transitive* and we refer to the others as *irreducible* or *non-transitive*. Removing reducible arcs from the graph can thus be performed without loss of information and we call such reduced graph a string graph. We can formalize its definition as follows.

Definition 2.18 *(String graphs) Let $T = \{s_1, \ldots, s_m\}$ be a set of strings, let τ be a positive integer value, and let $G_O = (V, E)$ be the overlap graph of T with minimum overlap τ. The String graph $G_S = (V^\star, E^\star)$ of T is a directed graph where $V^\star = V$ and $E = \{e : e \in E \cap e \text{ is not reducible}\}$.*

In bioinformatics, using String graphs is preferable due to the sheer amount of data needed to represent and, indeed, overlap graphs are more resource intensive then their reduced counterparts. A graphical representation of the overlap and string graph of strings s_1, s_2, and s_3 can be found in Figure 2.5.

De Bruijn and String Graphs are both widely used data structures for the de novo sequence assembly problem. The former is probably the most well-known one follow-

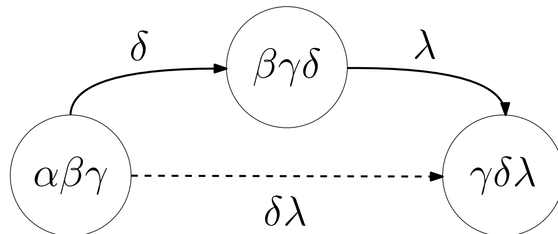


Figure 2.5: An example of an overlap graph. Irreducible arcs are represented by a straight line, reducible arcs by a dashed line. For readability, the arcs are labeled with only one of the two labels. Note that both paths spell the same string ($\alpha\beta\gamma\delta\lambda$).

ing its application in extremely influential articles [65, 111] whereas the latter regained relevance in the last years [134] after being in use twenty years ago [141]. The two implementations of assembly graphs present differences — most notably the way the nodes are connected — but an in-depth theoretical analysis of the repercussion they have on the final assembly is lacking. Indeed, the usual comparisons between the results are on an experimental basis and consider the performances of the whole pipeline with no particular focus on the consequences of choosing a graph over the other, e.g. Assemblathon [33], Assemblathon 2 [16], and GAGE [125] emphasize — reasonably — the overall performances of the tools examined.

Nevertheless, it is possible and easy to highlight the most obvious differences between the two graphs. String graphs can deal with repetitions in the original genome better than de Bruijn graphs. Indeed, since the whole information of the reads is considered in its construction, it is possible to avoid collapsing repetitions together when their length is smaller than the shortest read in the dataset, whereas de Bruijn graph can distinguish only repetition smaller than K. On the other end, de Bruijn graphs are easier to store and compute since the intrinsic characteristics of the nodes and of the edges can be exploited, whereas String graphs do not present such nice and consistent characterization of them. For the same reasons, de Bruijn graphs show extremely simple structures in presence of tips or bubbles that can simplify the tip removal and bubble popping steps of assemblers.

Presenting all the assemblers in the literature is out of the scope of this thesis. Nevertheless, we will report an incomplete list of assemblers proposed in the last years based

either on de Bruijn graphs or on String graphs to show the liveliness of this research field.

de Bruijn graph based assemblers Velvet [144], MetaVelvet [100], MetaVelvet-SL [127], SPAdes [4], IDBA [109], IDBA-UD [110], Abyss [136], Trans-Abyss [118], Trinity [57], SOAPdenovo [86], SOAPdenovo-Trans [143], Cortex [67], EULER [111], Oases [129], Minia [22];

String graph based assemblers SGA [134], Readjoiner [54], LSG [11], FSG [12], Celera [99], Edena [61], AMOS [117], MAP [75].

Finally, we want to note that there exist approaches that mix the usage of the two assembly graphs [20, 62], and that, at the best of our knowledge, only one assembler (MaSuRCA) [145] implements it.

Part I

Indexing data structures for assembly graphs

Context and motivations

In this part of the thesis two different approaches for efficiently storing and querying de Bruijn graphs (`dBG`) will be presented. De Bruijn graphs are widely used data structures for the sequence assembly problem, their main goal is to represent shared substring between the reads so that a path in the graph represents the DNA sequence of the sequenced individual. For a given set of reads $R = \{r_1, r_2, \ldots, r_n\}$ output by a biological experiment and a value K (usually referred as the *order* of the graph), a `dBG` is defined as $G_K = (V, E)$ where, as usual, V is the set of nodes of G and E is a set of edges $\{e_1 = (v_{i_1}, v_{j_1}), e_2 = (v_{i_2}, v_{j_2}), \ldots, e_m = (v_{i_m}, v_{j_m})\}$. The literature presents two slightly different types of de Bruijn graphs, namely *edge-centric* `dBG` and *node-centric* `dBG` in which the main difference between the two is the set of edges E. In the following we will refer to them as `e-dBG` and `n-dBG`, respectively. In both definitions, for any distinct sub-string of length K k_i (called a K-mer) in any $r_j \in R$, V will contain a vertex v_i labeled with k_i. In `e-dBG`, E will contain an edge e that connects two vertices v_i and v_j with labels k_i and k_j, respectively, if and only if: (i) k_i and k_j share a prefix and a suffix of length $K-1$ (*i.e.*, $k_i[2:K] = k_j[1:K-1]$ or $k_j[2:K] = k_i[1:K-1]$) and (ii) the $(K+1)$-mer $k_i \cdot k_j[K]$ appears in some input string $r \in R$. Conversely, edges in `n-dBG` have less strict constraints. Indeed, in this case E will contain an edge only if property (i) is met, *i.e.*, only if the two K-mer share a prefix and a suffix of length $K-1$ without forcing and constraint on the consecutiveness of the two K-mers.

Given the set R, since $K \in \mathbb{N}$, an unlimited number of `dBG` can be defined. For ease of presentation, we will refer to each of these by their order K and, more precisely, we will refer to the `dBG` of order K as $\texttt{dBG}_K$. Clearly, for each $K > \max\{|r| : r \in R\}$ there is no K-mer of length K in R and therefore all the $\texttt{dBG}_K$ will be the empty graph (*i.e.*, a graph with no vertices and no edges). Instead, for each $1 \leq K \leq \max\{|r| : r \in R\}$ the $\texttt{dBG}_K$ exists.

It is worth to note that, when considering the assembly problem, different $\texttt{dBG}_K$ yield different inferred sequences and choosing the correct order is fundamental for producing good assemblies. Choosing the *best* order is a research topic by itself and is required in different fields such as de novo genome assembly [21], de novo transcriptome assembly [32], and error correction [128]. Note that this thesis is not focused on this topic (choosing the best order) and we refer the reader to the related literature [4, 21, 47, 92]. A different approach is to use multiple orders either incrementally [110] or at the same time [81] to exploit the information provided by all the graphs and thus — ideally — producing better outputs.

In this part of the thesis we will present two different contribution to the field of indexes for de Bruijn graphs. The first one is a new deterministic succinct representation that has better space bounds than the ones currently presented in the literature when the number of connected components in in the graph is small and is also the first known succinct representation that is fully dynamic (*i.e.*, it allows to add *and* remove nodes and edges from the graph without rebuilding it from scratch). The second one falls in the field of Burrows-Wheeler Transform inspired indexing data structures and allows to traverse both incoming and outgoing edges of a node. This second data structure permits to efficiently compute unitigs in multiple orders of dBG storing only $\mathcal{O}(n \lg K)$ bits (where n is the total number of nucleotides in R) instead of $\mathcal{O}(Kn)$ bits. Moreover, possible applications of this data structure (more precisely, unitig construction from multiple dBG) will be shown.

This part is divided in three main chapters. The first one (Chapter 3) presents the approaches proposed in the literature, the second one (Chapter 4) presents the dynamic indexing data structure, and the third one (Chapter 5) presents the BWT-inspired indexing data structure.

3 De Bruijn graphs representations

In this section we will focus on presenting the main approaches for representing a `dBG` proposed in the literature. In theory, one can represent a `dBG` naïvely using general purpose data structures such as hash-tables and storing, for each node, the label and the adjacency list of its neighborhood. Indeed, early implementations of `dBG`-based assemblers developed simple representations based on distributed hash tables [136]. Although feasible on powerful servers or clusters, running those tools requires hundreds of GiB of main memory to store the informations included in a whole genome sequencing experiment and is unfeasible on resource-limited servers and PCs. To cope with the limited resources available, different methods have been proposed to lower the amount of RAM required to store these informations by either representing implicitly some data or using compressed data structures.

In this section we will briefly introduce three data structures proposed in the literature. The first one by Conway and Bromage [24] is a succinct data structure based on sparse bit-vectors, the second one by Chikhi and Rizk [22] (further improved by Salikhov et al. [122]) is based on a probabilistic membership data structure called Bloom filter, and the third one by Bowe et al. [14] (further improved by Boucher et al. [13]) is a deterministic data structure loosely inspired by the Burrows-Wheeler Transform.

Sparse bit-vector based data structure A first take on efficiently storing `dBG` was presented in 2011 by Conway and Bromage [24]. In this approach nodes are not stored explicitly and can be inferred from the edges. More precisely, nodes are sorted in lexicographic order and for each node v_i with label k_i a vector of $|\Sigma|$ (4 in the case of DNA) bits is created such that each bit is associated with a character in the alphabet. In order to represent existence or absence of edges, the bit associated to the character $c \in \Sigma$ is set to `1` (*i.e.*, `True`) if the string $k_i[2 : K] \cdot c$ is in the dataset (*i.e.*, there is a vertex labeled

with the concatenation of the $(K-1)$-suffix of k_i and c in `dBG`) and `0` (*i.e.*, `False`) otherwise. All the vectors are then concatenated into a single bit-vector B, compressed to near-optimal space using Elias-Fano coding [36], and, by efficient implementations [107] of `rank` and `select` queries it is possible to move from the bits related to a given node to the ones related to one of its neighbors and therefore explore the graph.

This data structure was the first one that reached space requirements close to the theoretical bounds and can be used to represent either n-dBG or e-dBG, albeit the description we just outlined is clearly focused on the latter. Nevertheless, it is trivial to store a `n-dBG` using such representation since we only need to store some additional edges by setting to `1` the corresponding bits in B.

A Bloom filter based data structure A second succinct encoding of `dBG` was presented in 2012 by Chikhi and Rizk [23]. This representation is based on the so called *Bloom filters* [9], an efficient and compact membership data structure based on hash functions. A Bloom filter can be defined as the union of a bit-vector B of length n (user defined) and a set of hash functions $\{h_1, h_2, \ldots, h_m\}$ mapping elements (in our case nodes in `dBG`) to the range $[1 \ldots n]$. When we want to add an element x to our set all the positions $h_1(x), h_2(x), \ldots, h_m(x)$ are computed and the corresponding bits in B are set to `True`. When we want to test if an element y is in our set, the positions $h_1(y), h_2(y), \ldots, h_m(y)$ are computed and the element is reported as absent if at least one position is set to `False` in B. This data structure is probabilistic since it produces *false positives* because hash functions produce collisions and an element can be reported as present due to a combinations of collisions with other elements actually in the set. We refer the reader to Figure 3.1 for an example of false positive produced by a Bloom filter.

Chikhi and Rizk proposed to use a Bloom filter to compactly represent nodes of the `dBG` and to store the list of false positives nodes reported as present in the graph by the Bloom filter alongside it. In order to limit the memory usage of such list (storing all the false positives could require too much space), the authors proposed to store only the false positive nodes *reachable* from nodes in the graph called *critical false positives*. Doing so, it is possible to have a deterministic data structure that requires an extremely low space, if we can assure that the analysis of the graph starts from a node that is

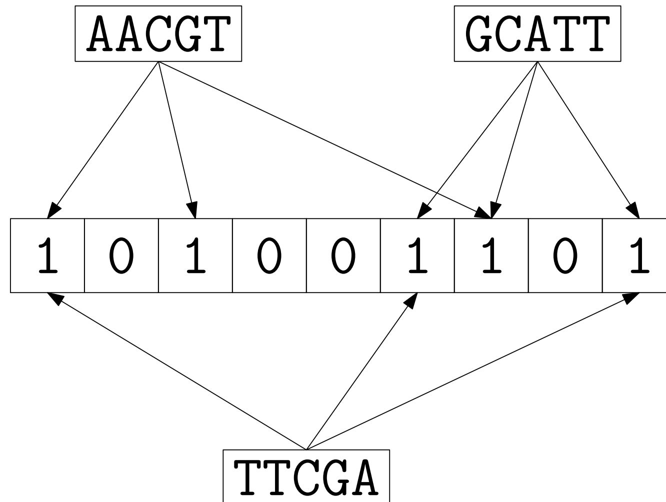


Figure 3.1: An example of Bloom filter with bitvector of length 9 and three hash functions. Sequences AACGT and GCATT are inserted in the in the Bloom filter and then the sequence TTCGA is tested producing a false positive.

actually in it.

It is worth to note that this data structure only represents nodes and does not stores edges. Indeed, since nodes connected by an edge share a prefix and a suffix of length $K-1$, it is possible to test efficiently if all the nodes that can be reached are in the set. Thus, this data structure represents a n-dBG.

In 2013, Salikhov et al. [122] improved this approach by noting that also the set of false positives can be represented by a Bloom filter. Obviously, this will produce false positives of the false positive set that, in turn, can be represented by mean of a (smaller) Bloom filter. Reiterating this method until the set of false positive of a Bloom filter is small enough leads to even greater memory savings than using a single Bloom filter.

A BWT-based data structure A third succinct data structure encoding dBG was presented by Bowe et al. in 2012 [14]. In the following, we will refer to this data structure as BOSS (from the concatenation of the first letter of each author's surname: Bowe, Onodera, Sadakane, and Shibuya).

This representation is heavily inspired by the Burrows-Wheeler Transform and the related literature (*i.e.*, BWT-like representation of labeled trees [40, 89]) and falls in the

well established field of compressed deterministic self-indexes. In this approach the goal is to produce some kind of linearization of the graph (*i.e.*, a 1-dimension representation of the information contained in it) such that it is possible to apply efficient algorithms designed to work on strings. In particular, `BOSS` represents the information of the labels of the edges of the `dBG` and does not store the labels of its nodes since they can be inferred by the former. In brief, given a `dBG`, its `BOSS` representation is a string produced by the concatenation of the labels of its edges sorted by the lexicographic order of the reverse of the label of the source node, *i.e.*, edges (v_i, v_j) are sorted by the reverse of the label of v_i. More precisely, in order to compute the `BOSS` representation of a `dBG` $G = (V, E)$, we first compute the set $N = \{(\mathtt{l}(e), \mathtt{reverse}(\mathtt{l}(\mathtt{source}(e))) : e \in E\}$ where, for an edge e in E, $\mathtt{l}(e)$ is its label, $\mathtt{source}(e)$ is the source node of e, and $\mathtt{reverse}(s)$ produces the reverse of the string s. We then sort the pairs in N by the lexicographic order of their second elements and create a string W of length $|N|$ by concatenating the first element of each pair. In addition to W, `BOSS` is composed by a bit-vector L of length $|N|$ used to group together labels of edges outgoing from the same node. Indeed, note that, since N is sorted by the lexicographic order of the reverse of the label of the source node, pairs related to edges outgoing from the same node will be consecutive in N. Therefore, it is possible to group together edges outgoing from the same node using a single bit for each element and, in particular, $L[i]$ will be set to `False` (or 0) if the i-th element of N has the same source node as the $(i-1)$-th. The overall space required by this data structure is $4|N| + o(|N|)$, since storing N requires 3 bits per edge and storing L requires 1 bit per edge. We point out that, albeit the DNA alphabet has size 4, for technical reasons its dimension is doubled since for each c in Σ a new character c^- is added to it. Therefore, the size of this extended alphabet is 8 and 3 bits are required to index each element in it.

As presented by Bowe et al., `BOSS` allows to represent a single order `e-dBG`. In many applications like genome assembly, multiple order of the same graph are used either at the same time [81] or iteratively [109]. Using a simple `BOSS`, this would require to compute a data structure for each order. In order to lower the amount of memory required, in 2015 Boucher et al. [13] extended this approach to represent multiple orders at the same time by storing an additional integer vector representing the Longest Common Suffix Array

(LCSA) of the labels of the nodes[1]. This way it is possible to represent all the graphs between order 1 and K using $\mathcal{O}(|N| \lg K) + 4|N| + o(|N|)$ bits instead of $K(4|N| + o(|N|)$.

[1]Note this means to compute the Longest Common Prefix Array of the second elements of the pairs in N.

4 Fully dynamic succinct de Bruijn graphs

In this section we will present a fully dynamic succinct `dBG` representation. This data structure is based on a combinations of Karp-Rabin hashing [68] and Minimal Perfect Hashing, that is similar to that using Bloom filters but has better theoretical bounds when the number of connected component in the graph is small, and is fully dynamic: *i.e.*, we can both insert and delete nodes and edges efficiently, whereas implementations based on Bloom filters are usually semi-dynamic and support only insertions.

We can summarize the results of this section with the two following technical lemmas:

Lemma 4.1 *Given a static set $S = \{k_1, k_2, \ldots, k_n\}$ of n K-tuples over an alphabet Σ of size σ, with high probability in $\mathcal{O}(kn)$ expected time we can build a function $f : \Sigma^k \to \{0, \ldots, n-1\}$ with the following properties:*

- *when its domain is restricted to S, f is bijective;*
- *we can store f in $\mathcal{O}(n + \log K + \log \sigma)$ bits;*
- *given a K-tuple k_i, we can compute $f(k_i)$ in $\mathcal{O}(K)$ time;*
- *given k_i and k_j such that the suffix of k_i of length $K-1$ is the prefix of k_j of length $K-1$, or vice versa, if we have already computed $f(k_i)$ then we can compute $f(k_j)$ in $\mathcal{O}(1)$ time (i.e., f is a Karp-Rabin — or rolling — hashing function).*

Lemma 4.2 *If S is dynamic then we can maintain a function f as described in Lemma 4.1 except that:*

- *the range of f becomes $\{0, \ldots, 3n-1\}$;*
- *when its domain is restricted to N, f is injective;*

- *our space bound for f is $\mathcal{O}(n(\log\log n + \log\log\sigma))$ bits with high probability;*
- *insertions and deletions take $\mathcal{O}(K)$ amortized expected time.*
- *the data structure may work incorrectly with very low probability (inversely polynomial in n).*

Suppose $G = (V, E)$ is a `dBG`. In Section 4.1 we show how we can store $\mathcal{O}(n\sigma)$ more bits than Lemma 4.1 such that, given a pair of K-tuples (k_i and k_j) of which at least one appears in G, we can check whether the edge (k_i, k_j) is in the graph. This means that, if we start with a K-tuple whose node is in V, then we can explore the entire connected component containing that K-tuple in the underlying undirected graph. On the other hand, if we start with a K-tuple not in V, then we will learn that fact as soon as we try to cross an edge to a K-tuple that is in V. To deal with the possibility that we never try to cross such an edge, however — *i.e.*, that our encoding as described so far is consistent with a graph containing a connected component disjoint from V — we cover the vertices with a forest of shallow rooted trees. We store each root as a K-tuple, and for each other node we store $1 + \lg\sigma$ bits indicating which of its incident edges leads to its parent. To verify that a K-tuple we are considering is indeed in the graph, we ascend to the root of the tree that contains it and check that K-tuple is what we expect. The main challenge for making our representation dynamic with Lemma 4.2 is updating the covering forest. In Section 4.2, we show how we can do this efficiently while maintaining our depth and size invariants. Finally, in Section 4.3 we observe that our representation can be easily modified for other applications by replacing the Karp-Rabin hash function by other kinds of hash functions.

4.1 Static de Bruijn Graphs representation

Let G be a de Bruijn graph of order K, let $V = \{v_1, \ldots, v_n\}$ be the set of its nodes, and let $E = \{a_1, \ldots, a_e\}$ be the set of its edges. We call each v_i either a node or a K-tuple, using interchangeably the two terms since there is a one-to-one correspondence between nodes and labels.

We maintain the structure of G by storing two binary matrices, IN and OUT, of size $n \times \sigma$. For each node, the former represents its incoming edges whereas the latter represents its outgoing edges. In particular, for each K-tuple $v_i = c_1c_2 \ldots c_{K-1}a$, the former stores a row of length σ such that, if there exists another K-tuple $v_j = bc_1c_2 \ldots c_{K-1}$ and an edge from v_i to v_j, then the position indexed by b of such row is set to 1. Similarly, OUT contains a row for v_i and the position indexed by a is set to 1. As previously stated, each K-tuple is uniquely mapped to a value between 0 and $n-1$ by f, where f is as defined in Lemma 4.1, and therefore we can use these values as indices for the rows of the matrices IN and OUT, *i.e.*, in the previous example the values of $\texttt{IN}[f(v_i)][b]$ and $\texttt{OUT}[f(v_j)][a]$ are set to 1. We note that, e.g., the SPAdes assembler [4] also uses such matrices.

Suppose we want to check whether there is an edge from bX to Xa. Letting $f(bX) = i$ and $f(Xa) = j$, we first assume bX is in G and check the values of $\texttt{OUT}[i][a]$ and $\texttt{IN}[j][b]$. If both values are 1, we report that the edge is present and we say that the edge is *confirmed* by IN and OUT; otherwise, if any of the two values is 0, we report that the edge is absent. Moreover, note that if bX is in G and $\texttt{OUT}[i][a] = 1$, then Xa is in G as well. Symmetrically, if Xa is in G and $\texttt{IN}[j][b] = 1$, then bX is in G as well. Therefore, if $\texttt{OUT}[i][a] = \texttt{IN}[j][b] = 1$, then bX is in G if and only if Xa is. This means that, if we have a path P and if all the edges in P are confirmed by IN and OUT, then either all the nodes touched by P are in G or none of them is.

We now focus on detecting false positives in our data structure maintaining a reasonable memory usage. Our strategy is to sample a subset of nodes for which we store the plain-text K-tuple and connect all the unsampled nodes to the sampled ones. More precisely, we partition nodes in the undirected graph G' underlying G into a forest of rooted trees of height at least $K \lg \sigma$ and at most $3K \lg \sigma$. For each node we store a pointer to its parent in the tree, which takes $1 + \lg \sigma$ bits per node, and we sample the K-mer at the root of such tree. We allow a tree to have height smaller than $K \lg \sigma$ when necessary, *i.e.*, if it covers a connected component. Figure 4.1 shows an illustration of this idea.

We can therefore check whether a given node v_i is in G by first computing $f(v_i)$ and then checking and ascending at most $3K \lg \sigma$ edges, updating v_i and $f(v_i)$ as we go. Once we reach the root of the tree we can compare the resulting K-tuple with the one

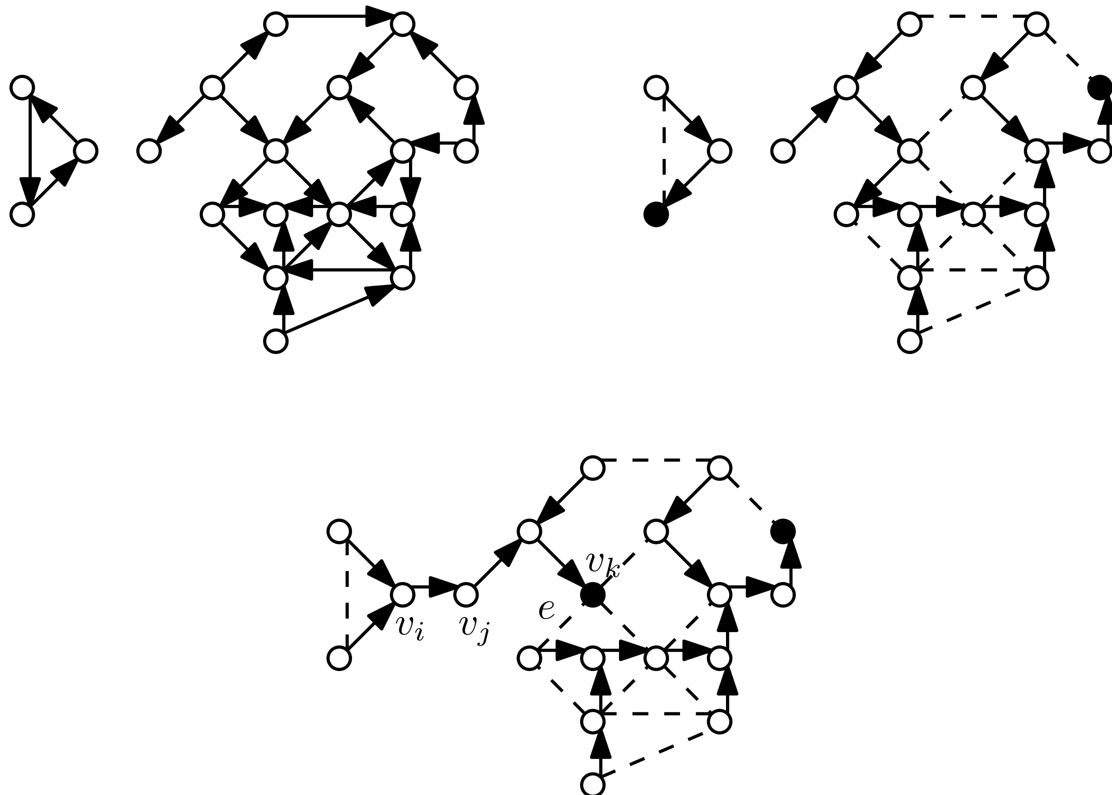


Figure 4.1: Given a de Bruijn graph (top left), we cover the underlying undirected graph with a forest of rooted trees of height at most $3K \lg \sigma$ (top right). The roots are shown as filled nodes, and parent pointers are shown as arrows; notice that the directions of the arrows in our forest are not related to the edges' directions in the original de Bruijn graph. We sample the K-tuples at the roots so that, starting at a node we think is in the graph, we can verify its presence by finding the root of its tree and checking its label in $\mathcal{O}(K \log \sigma)$ time. The most complicated kind of update (bottom) is adding an edge between a node v_i in a small connected component to a node v_j in a large one, v_j's depth is more than $2K \lg \sigma$ in its tree. We re-orient the parent pointers in v_i's tree to make v_i the temporary root, then make v_i point to v_j. We ascend $K \lg \sigma$ steps from v_j, then delete the parent pointer e of the node v_k we reach, making v_k a new root. (To keep this figure reasonably small, some distances in this example are smaller than prescribed by our formulas.)

sampled to check if v_i is in the graph. This procedure requires $\mathcal{O}(K \lg \sigma)$ time since computing the first value of $f(v_i)$ requires $\mathcal{O}(K)$, ascending the tree requires constant

time per edge, and comparing the K-tuples requires $\mathcal{O}(K \lg \sigma)$.

Construction algorithm We now describe a Las Vegas algorithm for the construction of this data structure that requires, with high probability, $\mathcal{O}(Kn + n\sigma)$ expected time. We recall that V is the set of input nodes of size n. We first select a function f and construct a bitvector B of size n initialized with all its elements set to `0`. For each element v of V we compute $f(v) = i$ and check the value of $B[i]$. If this value is `0` we set it to `1` and proceed with the next element in V, if it is already set to `1`, we reset B, select a different function f, and restart the procedure from the first element in V. Once we finish this procedure — *i.e.*, we found that f does not produce collisions when applied to V — we store f and proceed to initialize `IN` and `OUT` correctly. This procedure requires with high probability $\mathcal{O}(Kn)$ expected time for constructing f and $\mathcal{O}(n\sigma)$ time for computing `IN` and `OUT`. Notice that if V is the set of K-tuples of a single text sorted by their starting position in the text, each $f(v_i)$ can be computed in constant time from $f(v_{i-1})$ except for $f(v_1)$ that still requires $\mathcal{O}(K)$. More generally, if V is the set of K-tuples of t texts sorted by their initial position, we can compute $n - t$ values of the function $f(v_i)$ in constant time from $f(v_{i-1})$ and the remaining t in $\mathcal{O}(K)$. In this case the construction requires, with high probability, $\mathcal{O}(Kt + (n - t) + n\sigma) = \mathcal{O}(Kt + n\sigma)$ expected time.

Combining our forest with Lemma 4.1, we can summarize our static data structure in the following theorem:

Theorem 4.3 *Given a static σ-ary Kth-order de Bruijn graph G with n nodes, with high probability in $\mathcal{O}(Kn + n\sigma)$ expected time we can store G in $\mathcal{O}(\sigma n)$ bits plus $\mathcal{O}(K \log \sigma)$ bits for each connected component in the underlying undirected graph, such that checking whether a node is in G takes $\mathcal{O}(K \log \sigma)$ time, listing the edges incident to a node we are visiting takes $\mathcal{O}(\sigma)$ time, and crossing an edge takes $\mathcal{O}(1)$ time.*

4.2 Dynamic de Bruijn Graphs representation

In the previous section we presented a static representation of de Buijn graphs, we now show how we can make this data structure dynamic. In particular, we will show how we can insert and remove edges and nodes and that updating the graph reduces to managing

the covering forest over G. In this section, when we refer to f we mean the function defined in Lemma 4.2. We first show how to add or remove an edge in the graph and will later describe how to add or remove a node in it. The updates must maintain the following invariant: any tree must have size at least $K \log \sigma$ and height at most $3K \log \sigma$ except when the tree covers (all nodes in) a connected component of size at most $K \log \sigma$.

Let v_i and v_j be two nodes in G, $e = (v_i, v_j)$ be an edge in G, and let $f(v_i) = i'$ and $f(v_j) = j'$.

Suppose we want to add e to G. First, we set to `1` the values of $\texttt{OUT}[i'][a]$ and $\texttt{IN}[j'][b]$ in constant time. We then check whether v_i or v_j are in different components of size less than $K \lg \sigma$ in $\mathcal{O}(K \lg \sigma)$ time for each node. If both components have size greater than $K \lg \sigma$ we do not have to proceed further since the trees will not change. If both connected components have size less than $K \lg \sigma$ we merge their trees in $\mathcal{O}(K \lg \sigma)$ time by traversing both trees and switching the orientation of the edges in them, discarding the samples at the roots of the old trees and sampling the new root in $\mathcal{O}(K)$ time.

If only one of the two connected components has size greater than $K \lg \sigma$ we select it and perform a tree traversal to check whether the depth of the node is less than $2K \lg \sigma$. If it is, we connect the two trees as in the previous case. If it is not, we traverse the tree in the bigger components upwards for $K \lg \sigma$ steps, we delete the edge pointing to the parent of the node we reached creating a new tree, and merge it with the smaller one. This procedure requires $\mathcal{O}(K \lg \sigma)$ time since deleting the edge pointing to the parent in the tree requires $\mathcal{O}(1)$ time, *i.e.*, we have to reset the pointer to the parent in only one node.

Suppose now that we want to remove e from G. First we set to `0` the values of $\texttt{OUT}[i'][a]$ and $\texttt{IN}[j'][b]$ in constant time. Then, we check in $\mathcal{O}(K)$ time whether e is an edge in some tree by computing $f(v_i)$ and $f(v_j)$ checking for each node if that edge is the one that points to their parent. If e is not in any tree we do not have to proceed further whereas if it is we check the size of each tree in which v_i and v_j are. If any of the two trees is small (*i.e.*, if it has fewer than $K \lg \sigma$ elements) we search any outgoing edge from the tree that connects it to some other tree. If such an edge is not found we conclude that we are in a small connected component that is covered by the current tree and we sample a node in the tree as a root and switch directions of some edges if necessary. If such an edge

is found, we merge the small tree with the bigger one by adding the edge and switch the direction of some edges originating from the small tree if necessary. Finally if the height of the new tree exceeds $3K \log \sigma$, we traverse the tree upwards from the deepest node in the tree (which was necessarily a node in the smaller tree before the merger) for $2K \lg \sigma$ steps, delete the edge pointing to the parent of the reached node, creating a new tree. This procedure requires $\mathcal{O}(K \lg \sigma)$ since the number of nodes traversed is at most $O(K \lg \sigma)$ and the number of changes to the data structures is also at most $O(K \lg \sigma)$ with each change taking expected constant time.

It is clear that the insertion and deletion algorithms will maintain the invariant on the tree sizes. It is also clear that the invariant implies that the number of sampled nodes is $O(n/(K \log \sigma))$ plus the number of connected components.

We now show how to add and remove a node from the graph. Adding a node is trivial since it will not have any edge connecting it to any other node. Therefore adding a node reduces to modify the function f and requires $\mathcal{O}(K)$ amortized expected time. When we want to remove a node, we first remove all its edges one by one and, once the node is isolated from the graph, we remove it by updating the function f. Since a node will have at most σ edges and updating f requires $\mathcal{O}(K)$ amortized expected time, the amortized expected time complexity of this procedure is $\mathcal{O}(\sigma K \lg \sigma + K)$.

Combining these techniques for updating our forest with Lemma 4.2, we can summarize our dynamic data structure in the following theorem:

Theorem 4.4 *We can maintain a σ-ary Kth-order de Bruijn graph G with n nodes that is fully dynamic (i.e., supporting node and edge insertions and deletions) in $\mathcal{O}(n(\log \log n + \sigma))$ bits (plus $\mathcal{O}(K \log \sigma)$ bits for each connected component) with high probability, such that we can add or remove an edge in expected $\mathcal{O}(K \lg \sigma)$ time, add a node in expected $\mathcal{O}(K + \sigma)$ time, and remove a node in expected $\mathcal{O}(\sigma K \lg \sigma)$ time, and queries have the same time bounds as in Theorem 4.3. The data structure may work incorrectly with very low probability (inversely polynomial in n).*

4.3 Applications

Karp-Rabin hash functions implicitly divide their domain into equivalence classes — *i.e.*, subsets in which the elements hash to the same value. In this chapter we have chosen Karp-Rabin hash functions such that each equivalence class contains only one K-tuple in the graph. Most of our efforts have gone into being able, given a K-tuple and a hash value, to determine whether that K-tuple is the unique element of its equivalence class in the graph. In some sense, therefore, we have treated the equivalence relation induced by our hash functions as a necessary evil, useful for space-efficiency but otherwise an obstacle to be overcome. For some applications, however — *i.e.*, parameterized pattern matching, circular pattern matching or jumbled pattern matching — we are given an interesting equivalence relation on strings and asked to preprocess a text such that later, given a pattern, we can determine whether any substrings of the text are in the same equivalence class as the pattern. We can modify our data structure for some of these applications by replacing the Karp-Rabin hash function by other kinds of hash functions.

For indexed jumbled pattern matching [2,18,73] we are asked to pre-process a text such that later, given a pattern, we can determine quickly whether any substring of the text consists of exactly the same multiset of characters in the pattern. Consider fixed-length jumbled pattern matching, when the length of the patterns is fixed at pre-processing time. If we modify Lemmas 4.1 and 4.2 so that, instead of using Karp-Rabin hashes in the definition of the function f, we use a hash function on the histograms of characters' frequencies in K-tuples, our function f will map all permutations of a K-tuple to the same value. The rest of our implementation stays the same, but now the nodes of our graph are multisets of characters of size K and there is an edge between two nodes v_i and v_j if it is possible to replace an element of v_i and obtain v_j. If we build our graph for the multisets of characters in K-tuples in a string T, then our process for checking whether a node is in the graph tells us whether there is a jumbled match in T for a pattern of length K. If we build a tree in which the root is a graph for all of T, the left and right children of the root are graphs for the first and second halves of T, etc., as described by Gagie et al. [45], then we increase the space by a logarithmic factor but we can return the locations of all matches quickly.

Theorem 4.5 *Given a string $T[1..n]$ over an alphabet of size σ and a length $K \ll n$, with high probability in $\mathcal{O}(Kn + n\sigma)$ expected time we can store $(2n \log \sigma)(1 + o(1))$ bits such that later we can determine in $\mathcal{O}(K \log \sigma)$ time if a pattern of length K has a jumbled match in T.*

5 Bidirectional succinct de Bruijn graphs

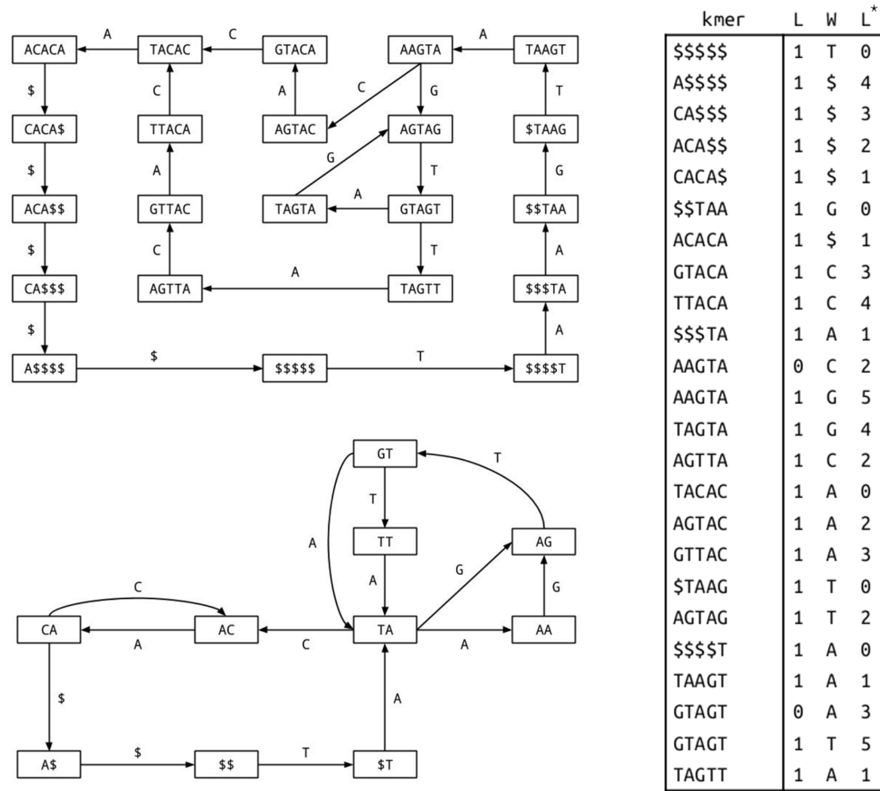
In this section we will present a succinct data structure for representing a set of de Bruijn graphs for various order that permits to move backwards in them. In particular, given a set of texts S and an integer K, this data structure, called bidirectional variable order BOSS (`biBOSS` for short), represents all the `dBG`s of S of order between 1 and K and their reverse. The main goal of this data structue is to overcome the limitations of what was presented in 2015 by Boucher et al. [13] when describing approaches for storing multiple `dBG`s at the same time. Supporting variable-orders is useful when non-uniform sampling of the reads leads to some parts of the graph being sparser than others but it's an asymmetric representation. The `BOSS` representation is asymmetric in the sense that, when visiting a node v_i, it takes much longer to follow the edge with a given label arriving at v_i than it does to follow the edge with that label leaving v_i.

The contribution of this chapter is a space-efficient bidirectional variable-order `BOSS` representation. Although being motivated partly by aesthetics — symmetry is beautiful — such representation is useful when building unitigs considering multiple orders at the same time.

The rest of this chapter is laid out as follows: in Section 5.1 we briefly review the definitions of fixed- and variable-order `BOSS` (see Section 3 for a more thorough description), in Section 5.2 we describe our bidirectional variable-order `BOSS` representation, and in Section 5.3 we briefly show possible applications of this data structure.

5.1 Fixed- and variable-order `BOSS`

We recall that the `BOSS` representation of a K-th order `dBG` is essentially the union of two vectors. The first one, W — also called edge-BWT —, is the concatenation of the labels of the edges of the graph sorted by the lexicographic order of the reverse of the



kmer	L	W	L*
$$$$$	1	T	0
A$$$$	1	$	4
CA$$$	1	$	3
ACA$$	1	$	2
CACA$	1	$	1
$$TAA	1	G	0
ACACA	1	$	1
GTACA	1	C	3
TTACA	1	C	4
$$$TA	1	A	1
AAGTA	0	C	2
AAGTA	1	G	5
TAGTA	1	G	4
AGTTA	1	C	2
TACAC	1	A	0
AGTAC	1	A	2
GTTAC	1	A	3
$TAAG	1	T	0
AGTAG	1	T	2
$$$$T	1	A	0
TAAGT	1	A	1
GTAGT	0	A	3
GTAGT	1	T	5
TAGTT	1	A	1

Figure 5.1: A BOSS representation of a dBG. On the left two the bruijn graph for the same dataset are reported: the one on top is of order 5 and the one on the bottom is of order 2. On the right the variable-order BOSS representation is reported. Note that K-mers are reported but are not part of the data structure.

label of their source node. The second one, L, is used to group together consecutive edges in W outgoing from the same node.

Bowe et al. [14] described a number of queries for traversing the graph, all of which can be implemented in terms of the following three basic queries, with at most an $\mathcal{O}(\sigma)$-factor slowdown:

- `forward`(v, a) returns the node u reached from v by an edge labelled a, or NULL if there is no such node;
- `backward`(v) lists the nodes u with an edge from u to v;
- `lastchar`(v) returns the last character of v's label.

In `BOSS`, nodes correspond to the intervals of the edge-BWT. Boucher et al. [13] augment the `BOSS` representation of the original graph, by storing the length of the longest common suffix of each consecutive pair of nodes, to support the following three queries:

- `shorter`(v, K') returns the node whose label is the last K' characters of v's label;
- `longer`(v, K') lists nodes whose labels have length $K' \leq K$ and end with v's label;
- `maxlen`(v, a) returns some node in the graph of maximum order whose label ends with v's label, and that has an outgoing edge labelled a, or NULL if there is no such node.

Together, these operations allow the order of the de Bruijn graph to be changed on the fly. The main addition to `BOSS` is a wavelet tree over the array L^* storing the length of the longest suffix common to each row in the `BOSS` matrix and the preceding row.

5.2 Bidirectional BOSS

`BOSS` allows us to move backward in the graph, accurately using `rank` and `select` on W and L. This procedure has a major drawback, which is that we cannot read which is the label of the edge we are traversing backward but can only read labels when we traverse edges forward. Indeed, it is possible to read the labels outgoing from a node by following an edge and using the function `lastchar(v)` (that returns the last character of a node in constant time). Nevertheless, moving backward in `BOSS` is performed "blindly", *i.e.*, without reading the label of the edge since the function `backward(v)` only returns a list of nodes and there is no `firstchar(v)` function.

A naïve yet inefficient solution would be to traverse K' edges backward in order to retrieve the character in the first position of the current K'-mer. Note that this way we *shift* the label of the current vertex until the first character becomes the label of the outgoing edge of the reached vertex. This procedure clearly requires $\mathcal{O}(K')$ for each incoming edge of the source node. Moreover, if the current node has j incoming edges we should perform $j \times K'$ backward steps in order to find all the possible backward

labels of those edges. Therefore, for an alphabet of size σ this procedure would require $\mathcal{O}(\sigma \times K')$.

For fixed order `BOSS` we can avoid backtracking in the graph by storing the first character of each K-mer, although this approach is not viable in variable-order `dBG` since it requires to store the whole set of K-mers. Indeed, note that storing the first character of the K-mers allows us to gather the backward label of an edge in the graph of order K but we need to store the second character in order to gather the backward label of an edge in the graph of order $K-1$, the third character for the graph of order $K-2$, and so on. We now introduce an elegant and efficient approach to move backward and forward in `BOSS`, namely bidirectional `BOSS` (`biBOSS` for short). This idea is loosely inspired by bidirectional BWT.

First, note that if we build the `dBG` of order K' for a set of strings and their reverse, we obtain two isomorphic graphs; we refer to the former as $\texttt{dBG}^f_{K'}$ and to the latter as $\texttt{dBG}^r_{K'}$. For each vertex v^f_i with label l_i in $\texttt{dBG}^f_{K'}$ there is a vertex v^r_i with label $\texttt{reverse}(l_i)$ in $\texttt{dBG}^r_{K'}$ and for each edge $e^f_h = (v^f_i, v^f_j)$ labeled with $\sigma^f = v^f_j[K']$ in $\texttt{dBG}^f_{K'}$ there is an edge $e^r_h = (v^r_j, v^r_i)$ labeled with $\sigma^r = v^r_i[K'] = v^f_i[1]$ in $\texttt{dBG}^r_{K'}$. Therefore, if we can maintain a link between the nodes and the edges in the two graphs we can easily retrieve the forward and backward labels simply by looking at e^f_h and e^r_h.

Moreover, note that outgoing edges from v^f_i in $\texttt{dBG}^f_{K'}$ are edges incoming to v^r_i in $\texttt{dBG}^r_{K'}$ and, conversely, edges outgoing from v^r_i are incoming to v^f_i. This remark clearly points out that we can simulate a backward step in $\texttt{dBG}^f_{K'}$ with a forward step in $\texttt{dBG}^r_{K'}$ without any need for further backtracking in neither $\texttt{dBG}^f_{K'}$ nor $\texttt{dBG}^r_{K'}$.

A `biBOSS` for a set of strings S is therefore a data structure composed by two `BOSS`. The first one, $\texttt{BOSS}^f$, is the `BOSS` data structure for S whereas the second one, $\texttt{BOSS}^r$, is the `BOSS` data structure for $S^r = \{\texttt{reverse}(s_i) : s_i \in S\}$. Each node v^f_i in $\texttt{dBG}^f_{K'}$ is defined in $\texttt{BOSS}^f$ as an interval $I^f_i = [b^f, e^f]$ over W, L, and L^* (W^f, L^f, and L^{*f} from now on). Conversely, each node v^r_i in $\texttt{dBG}^r_{K'}$ is defined in $\texttt{BOSS}^r$ as an interval $I^r_i = [b^r, e^r]$ over W^r, L^r, and L^{*r}.

Therefore, in order to support forward and backward navigation of the DBG we propose to maintain a pair of intervals (I^f_i, I^r_i), one to describe the vertex v^f_i (I^f_i) and the other to describe the vertex v^r_i (I^r_i). From now on we will use v^f_i and v^r_i to define vertices

and intervals on their respective BOSS interchangeably.

As described in section 5.1, Boucher et al. [13] described three main functions for a variable order BOSS, namely `shorter`, `longer`, and `maxlen` in order to move upwards and downwards in the orders of the dBGs and to explore the graph. Obviously we want to support the same set of functions so we will provide the biBOSS versions of the first two. We will not define `maxlen` for biBOSS since it is related to a specific direction of the graph and wouldn't make too much sense in this case. The examples in Figures 5.2–5.4 show a biBOSS graphical representation for a dBG of maximum order 5. We will use this example to show the execution of the new procedures graphically.

Shorter Let v_i^f and v_i^r be two vertices of order K' labeled by l_i and $\texttt{reverse}(l_i)$ and let K be the maximum order of the variable order BOSS. We define `bi-shorter` as $\texttt{bi-shorter}(v_i^f, v_i^r) = (v_j^f, v_j^r)$ such that the labels of v_j^f and v_j^r are respectively $l_i[2:K']$ and $\texttt{reverse}(l_i[2:K']) = \texttt{reverse}(l_i)[1:K'-1]$. This function returns the node in $\texttt{dBG}^f_{K'-1}$ which label is the last $K'-1$ character of l_i and its linked node in $\texttt{dBG}^r_{K'-1}$. Note that we can swap the two vertices in order to move downward between the orders of any $\texttt{dBG}^r_*$.

Computing v_j^f is straightforward since we only need to compute $\texttt{shorter}(v_i^f, K'-1)$ as defined in section 5.1. This procedure requires $\mathcal{O}(\lg K)$ time [13].

Computing v_j^r is more challenging since we need to remove the last character of $\texttt{reverse}(l_i)$. Nevertheless we can easily find a node in the graph with maximum order K that ends with $\texttt{reverse}(l_i)[1:K'-1]$ by selecting any position in the interval v_i^r and applying `backward` as defined in Bowe et al. [14] and then moving downwards in the order of the graph. More formally we can say that $v_j^r = \texttt{shorter}(\texttt{backward}(\texttt{maxlen}(v_i^r, *)), K'-1)$. This procedure requires $\mathcal{O}(\lg K)$ time since we can compute `maxlen` and `backward` in constant time and `shorter` in $\mathcal{O}(\lg K)$.

The example in Figure 5.2 shows the computation of `bi-shorter` on the pair of linked vertices $v_i^f = AAGTA$ and $v_i^r = ATGAA$. The vertex $v_j^f = AGTA$ is computed by finding the L^{*f}-interval of order 4 that contains v_i^f. In order to compute v_j^r we first gather any node in the original graph contained in v_i^r (in this case the same node since v_i^r has maximum order 5) and, by applying `backward` to it, obtain a node in which the

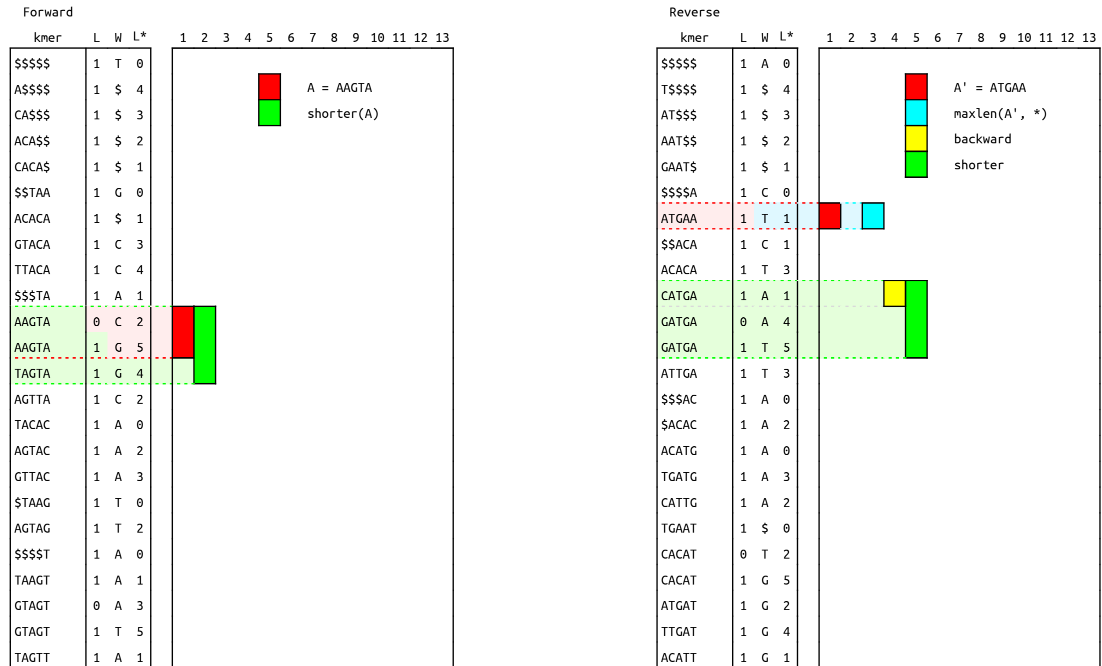


Figure 5.2: A graphical example of `bi-shorter`.

reversed label of v_j^f is a suffix ($CATGA$). Finally we move downwards in the order of $\texttt{BOSS}^r$ obtaining the vertex $v_j^r = ATGA$.

Note that `bi-shorter` and `shorter` are slightly different since the former only allows to move between two adjacent orders (*i.e.*, between K and $K-1$) whereas the latter allow to move between multiple orders at the same time. Nevertheless, `bi-shorter` can simulate the behavior of `shorter` by performing repeatedly the same operations (increasing the time complexity to $\mathcal{O}(J \lg K)$ where J is the difference between the two orders of the `dBG`s).

Longer Let $\sigma_k \in \Sigma$ be a character in the alphabet; we define `bi-longer` as

$$\texttt{bi-longer}(v_i^f, v_i^r, \sigma_h) = (v_j^f, v_j^r)$$

such that the labels of v_j^f and v_j^r are respectively $\sigma_h \cdot l_i$ and $\texttt{reverse}(l_i) \cdot \sigma_h$, thus the two computed vertices have reversed labels. Note that `bi-longer` is slightly different than

its corresponding function in variable order `BOSS`, namely `longer`. Indeed, applying the latter function (`longer`) to a vertex $v \in \texttt{dBG}^f_i$ in order to gather the nodes of order j, returns a list of vertices $V \in \texttt{dBG}^f_j$ such that each vertex $v_h \in V$ has a label that ends with the label of v. `bi-longer`, instead, allow us to select the character we want to concatenate to the labels. Nevertheless, gathering all the vertices from `bi-longer` to simulate `longer` is straightforward.

Clearly, we cannot directly compute v^f_j using `longer`. Our goal is therefore to provide a method that allows us to select the correct vertex from the list produced by `longer`, i.e. the one labeled with $\sigma_h \cdot l_i$. First, note that if V is the list returned by $\texttt{longer}(v^f_i, K'+1)$ then the labels of the vertices in V end with l_i and have length equal to $|l_i|+1$. Moreover, the vertices in V are sorted by lexicographic order of the reverse of the labels and for each vertex v_h in V the first character of its label is in the interval v^r_i. Indeed, since v_h is a backward extension of v^f_i its first character is a label of an edge outgoing from v^r_i. This remark hints that we can analyze v^r_i in order to correctly label the vertices in V. It is easy to prove that $|V|$ is equal to the cardinality of the set of the distinct characters in v^r_i and, since the elements of V are sorted, we can easily link each vertex with its first character and select the vertices that starts with σ_h (if it exists).

Computing v^r_j is straightforward, we need only to follow an edge labeled by σ_h (if it exists) and then find the vertex of order $K'+1$. More formally, if t is the cardinality of the set $\{c : c \in v^r_i \cap c < \sigma_h\}_{\neq}$, then v^f_j and v^r_j can be computed respectively as $\texttt{longer}(v^f_i, K'+1)[t+1]$ and $\texttt{shorter}(\texttt{forward}(\texttt{maxlen}(v^r_i, \sigma_h)), K'+1)$.

This procedure requires $\mathcal{O}(\sigma \lg \sigma + |V| \lg K)$ time since computing t requires us to compute the `rank` values for each character in the alphabet at the beginning and at the end of the interval of v^r_i ($\mathcal{O}(\sigma \lg \sigma)$), `longer` takes $\mathcal{O}(|V| \lg K)$, `forward` takes constant time, `maxlen` takes $\mathcal{O}(\lg \sigma)$, `shorter` takes $\mathcal{O}(\lg K)$, and selecting an element from a list takes constant time. When $\sigma = \mathcal{O}(1)$, therefore, `bi-longer` takes $\mathcal{O}(\lg K)$ time.

The example in Figure 5.3 shows the computation of `bi-longer` on the pair of linked vertices $v^f_i = TA$ and $v^r_i = AT$ with $\sigma_h = T$. The interval v^f_i is first split into the 3 possible vertices of order 3 using L^{*f}. By analyzing v^r_i we find that the characters at the beginning of each 3-mer are respectively \$, G, and T. We therefore select the third interval since $\$ < G < T$. In order to compute v^r_j we first select an edge in v^r_i labeled

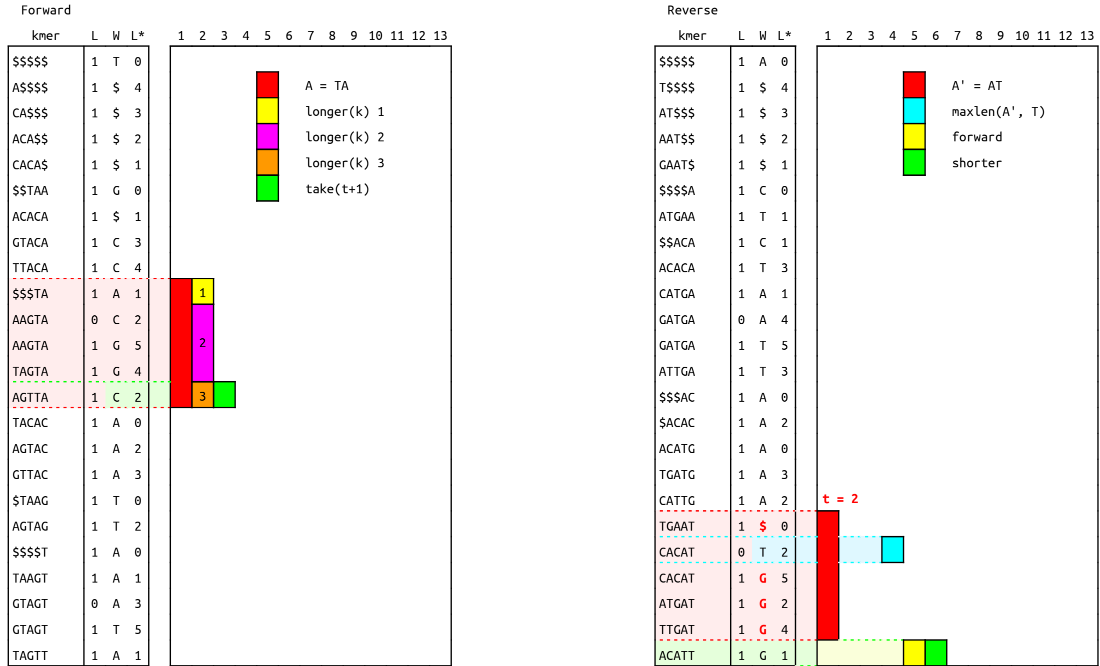


Figure 5.3: A graphical example of `bi-longer`.

with T (the one outgoing from $CACAT$) and follow it, obtaining a vertex in the graph of maximum order 5 that has the label of v_j^r as suffix ($ACATT$). As a last step we move upward in the order of the graph using the information stored in L^{*r} and obtaining the correct vertex in $\texttt{BOSS}^r$ (ATT, represented by the same interval).

Forward-Backward Until now we have described how we can move between the different orders described by the variable order `BOSS` maintaining the link between the nodes. We will now show how to move forward in either of the two graphs and maintain the link with the vertex in the other one with reversed label. Note that this actually means we move backward in one of the two graphs by selecting the correct graph in which to perform the forward step.

Let σ_h be a character in the alphabet; we define $\texttt{FwdBwd}(v_i^f, v_i^r, \sigma_h) = (v_j^f, v_j^r)$ such that the labels of v_j^f and v_j^r are respectively $l_i[2 : K'] \cdot \sigma_h$ and $\sigma_h \cdot \texttt{reverse}(l_i)[1 : K'-1] = \texttt{reverse}(l_i[2 : K'] \cdot \sigma_h)$.

Computing v_j^f is straightforward since we only need to compute $\texttt{forward}(v_i^f, \sigma_h)$ as

defined in Section 5.1. This step requires $\mathcal{O}(\lg K)$.

Computing v_j^r is mostly a combination of the previous two functions. Indeed, if we consider the labels of v_i^r (l_i) and v_j^r (l_j) we can note that, for some $\sigma_l \in \Sigma$, $l_i \cdot \sigma_l = \sigma_h \cdot l_j$, that is, in order to obtain v_j^r we must remove the last character of v_i^r (σ_l) and concatenate σ_h at the beginning of the obtained label. This two-step description clearly highlights the connections with `bi-shorter` (delete the last character) and `bi-longer` (concatenate a character at the beginning).

The proposed method is as follows. First we compute the interval for $\texttt{reverse}(l_i[2 : K'])$ by applying `backward` to v_i^r; note that this step produces a node (v_t^r) in $\texttt{dBG}^r_{K'-1}$ which label is a suffix of the label of v_j^r. At this point it is easy to see that we can apply `longer` to v_p^r in order to find the list of vertices V that share the label of v_p^r as suffix; clearly v_j^r will be in V by definition. Selecting the correct vertex can be done similarly as in `bi-longer`, the vertices are sorted by lexicographic order of the reverse of their labels and we can access the different characters by analyzing $\texttt{shorter}(v_i^f, K'-1)$.

More formally, if t is the cardinality of the set $\{c : c \in \texttt{shorter}(v_i^f, K'-1) \cap c < \sigma_h\}_{\neq}$, we can compute v_j^f and v_j^r using the following formulas:

$$v_j^f = \texttt{forward}(v_i^f, \sigma_h)$$

and

$$v_j^r = \texttt{longer}(\texttt{shorter}(\texttt{backward}(\texttt{maxlen}(v_i^r, *)), K'-1), K'+1)[t]$$

This procedure requires $\mathcal{O}(\sigma \lg \sigma + |V| \lg K)$ time where $|V| \leq \sigma$ is the number of nodes returned by `longer`. Computing t requires us to compute `shorter` ($\mathcal{O}(\lg K)$) and perform the same `rank` operation as for `bi-longer` ($\mathcal{O}(\sigma \lg \sigma)$), computing v_j^f requires $\mathcal{O}(\lg K)$, and computing v_j^r requires $\mathcal{O}(|V| \lg K)$ since `maxlen` and `backward` can be computed in constant time, `shorter` requires $\mathcal{O}(\lg K)$, and `longer` requires $\mathcal{O}(|V| \lg K)$. When $\sigma = \mathcal{O}(1)$, therefore, `FwdBwd` takes $\mathcal{O}(\lg K)$ time.

The example in Figure 5.4 shows the computation of `FwdBwd` on the pair of linked vertices $v_i^f = \texttt{GTA}$ and $v_i^r = \texttt{ATG}$ with $\sigma_h = G$. First we gather the index t, that we will use in the last step, by applying `shorter` to v_i^f (obtaining the vertex `TA`) and counting the number of distinct character smaller than σ_h. We then compute v_j^f by selecting an

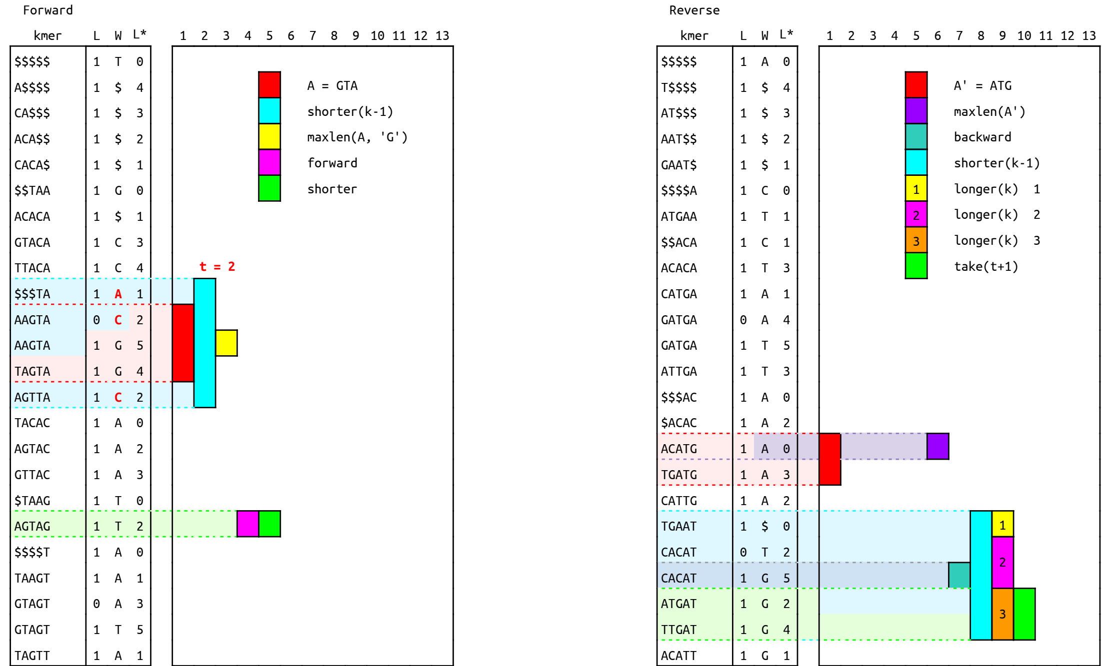


Figure 5.4: A graphical example of `FwdBwd`.

edge in v_i^f labeled with σ_h and following it, obtaining the vertex in the original graph labeled with `AGTAG`. We then apply `shorter` to gather the vertex v_j^f of order 3. In order to compute v_j^r we first select any edge in v_i^r and traverse it backward obtaining a vertex with suffix `AT`. We then compute v_j^r by applying `shorter` (order $K' - 1$) and `longer` (order K') and selecting the t-th interval computed. This example clearly shows why we cannot directly compute `shorter` of order K'. When we select a random edge in v_i^r we cannot directly access the K'-mers so we may concatenate a character $\sigma_l \neq \sigma_h$ at the beginning of the label of our vertex (in our example $\sigma_l =$ `C`). We therefore need to get rid of this character by moving downward between the orders obtaining the node `AT` and then select the correct vertex by moving upward using the information of the edges outgoing from its reverse (`TA` $=$ `shorter`$(v_i^f, K' - 1)$).

We can summarize the results of this section in the following theorem.

Theorem 5.1 *When $\sigma = \mathcal{O}(1)$, we can store a variable-order de Bruijn graph of maximum order K in $\mathcal{O}(n \lg K)$ bits on top of the* `BOSS` *representation of the order-K graph,*

where n is the number of nodes in the order-K graph, such that incrementing or decrementing the order and forward or backward traversals take $\mathcal{O}(\lg K)$ time.

5.3 Applications

The `biBOSS` data structure and the functions we have described in this chapter can be applied to the sequence assembly problem. First of all, we note once again that different approaches build unitigs by considering multiple orders of the de Bruijn graphs either at the same time or iteratively. For example, the IDBA [110] metagenome assembler iteratively builds de Bruijn graphs for increasing orders, compute the unitigs, and use them when computing unitigs in higher order `dBG`. Although reasonable, such approach requires to build the `dBG` multiple times and, most importantly, it only considers orders in one direction, *i.e.*, only for increasing order value. At the best of our knowledge, no method proposed in the literature allows to build unitigs by increasing or decreasing the context — *i.e.*, the order of the `dBG` — at will. Doing so, while analyzing the graph it could be possible to jump between the orders selecting the *best* one at each extension of the unitig, possibly considering the both the length of the context and the average coverage of it.

`biBOSS` has all the characteristics to unlock this opportunity. Indeed, starting from a given seed, with this data structure we can extend a pattern, *i.e.*, an unitig, in one direction using `bi-longer` if there is only one possible choice, append the character to the unitig and repeat such process. Once `bi-longer` cannot be performed (*i.e.*, when there are no possible extensions of the unitig or the coverage of the extended unitig — easily inferred by the width of the interval — is too low), we can shorten the context considered by applying `bi-shorter` and repeat the process until an unavoidable branch is reached. Finally, we can perform the same process on the reverse of the seed switching the graphs used and the intervals of the seed.

Note that we cannot use a simple variable order `BOSS` with this approach since, in order to guarantee that we compute unitigs, we have to test that there is a single incoming edge in the current node. Indeed, a variable order boss `BOSS` only allows to "see" in one direction and checking the label of an incoming edge would require some additional and

probably convoluted method.

We note that unitig is probably an abuse of terminology in this case but we chose to use it to better explain the goal of this procedure.

Part II

Building assembly graphs from self-indexes

Context and motivation

In this part of the thesis we will present a new *framework* for the genome assembly problem. We will show relations between indexing data structure and assembly graphs — more precisely string graphs — with an in-depth theoretical analysis and, after that, we will show how these properties can be exploited to implement either external memory assembler or parallel assemblers.

We note that indexing data structure based assemblers have been proposed in the past [134] and proved to be extremely efficient for genome assembly [135]. Although those methods hint to underlying relations between the Burrows-Wheeler Transform and assembly graph and exploit them, no formal and thorough analysis is performed. In the first chapter of this section (Chapter 6) we will fill this gap by a complete and accurate theoretical investigation of it. In the successive chapters we will show that these properties can be efficiently exploited in order to design both external memory (Chapter 7) and parallel (Chapter 8) methods and tools that can suite different requirements and will compare their performances with state-of-the-art tools.

String Graph based assembly

We are currently witnessing the rise of different Next-Generation Sequencing (NGS) technologies, each with its own peculiarities, especially with respect to read lengths that typically range between 50 and 150 bases [125]. De novo sequence assembly is still one of the most fundamental problems in Bioinformatics and is continuously receiving much attention. Most of the available assemblers [4, 110, 136] are built upon the notions of de Bruijn graphs and of k-mers (short k-long substrings). To analyze datasets coming from different technologies, hence with a large variation of read lengths, an approach based on same-length strings is likely to be limiting.

The amount of data necessary to assemble a (meta)genome emphasizes the need for algorithmic solutions that are time and space efficient. For example, over 500 gigabases of data have been analyzed to start a catalogue of the human gut microbiome [113]. To manage such a dataset, it is important to reduce main memory usage while keeping a reasonable time efficiency. Such challenge is usually tackled with the introduction of compact data structures for indexing reads, on which it is possible to build an efficient assembler. Papers applying the notion of Bloom filter [23, 108] to genome assembly follow this direction. For instance, a standard representation of the de Bruijn graph for the human genome when $k = 27$ requires 15GB (and is unfeasible in metagenomics), while a probabilistic version requires less than 4GB [23]. Larger values of k increase the memory requirements; other common values of k are 55 and 77 [112].

Alternative approaches have been developed recently, mostly based on the idea of *string graph*, initially proposed by Myers [97] before the advent of NGS technologies and further developed [134, 135] to incorporate some advances in text indexing, such as the FM-index [42]. These methods build a graph whose vertices are reads and the paths allow to assemble overlapping reads into larger contigs.

Since the optimal value of k for k-mers is usually less than half the read length [21], string graphs can immediately disambiguate some repeats that methods based on de Bruijn graphs might resolve only at later stages. Even without repetitions, analyzing only k-mers instead of the longer reads can result in some information loss, since bases of a read that are more than k positions apart are not part of the same k-mer. Moreover, string graphs methods can deal easily with mixed length reads. On the other hand, string graphs are more computationally intensive to compute [135].

The most used string graph assembler is SGA [134]. The main technical devices of SGA are the construction of the BWT [19] and of the FM-index of a set of strings, and their use to efficiently compute the arcs of the string graph (connecting overlapping reads). The original SGA has estimated requirement of 700GB of main memory to assemble the human genome [134].

Another string graph assembler is Fermi [79] which is inspired by SGA and tailored for variant calling. The main data structure of Fermi is a single bidirectional FM-index.

A number of recent works face the problem of designing efficient algorithmic strategies

or data structures for building string graphs. Among those works we can find a string graph assembler [7], based on a careful use of hashing and Bloom filters, with performance comparable with the first SGA implementation [134]. Another important alternative approach to SGA is Readjoiner [54] which is based on an efficient computation of a subset of exact suffix-prefix matches of the overlap graph and by subsequent round of suffix sorting, scanning, and filtering outputs the irreducible edges of the graph.

All the string graph based assemblers rely on both efficient indexing data structures and the original reads set in order to detect prefix-suffix overlaps between the elements. Since self-indexing data structures such as FM-index represents the whole information of the original dataset, an interesting problem is to design efficient algorithms for the construction of string graphs that only require the index and do not directly access the reads set. Improvements in this direction have both theoretical and practical motivations. Indeed, detecting prefix-suffix overlaps by the analysis of the (compressed) index alone is an almost unexplored problem and managing such data structure is usually more efficient since it is more compressible than the original dataset (thus requiring less memory than the whole reads set) and can be copied between machines faster.

6 Self-index based assembly framework

In this section we focus on a thorough presentation of the tight relations between self-indexes and overlap graphs. The main goal is to present a new representation of such graphs that includes all the information required to remove redundant information from them by storing a constant amount of memory per edge.

6.1 Preliminaries

In the following we will consider a collection $T = \{r_1, \ldots, r_m\}$ of m distinct strings (also called *reads*) over a finite alphabet Σ. As usual we consider each string in T to have a special character \$, smaller than any character in Σ, at the end. Furthermore, we assume that T is *substring-free*, that is, no string in T is a substring of another string in it. We denote by n the total number of characters in the input strings plus the m sentinels and by $\ell_{\max}$ the maximum length of a string, that is $n = \sum_{i=1}^{m}(|r_i| + 1)$ and $\ell_{\max} = \max_{i=1}^{m}\{|r_i|\}$.

Generalized Suffix Arrays, Longest Common Prefix arrays, and the Burrows-Wheeler Transform (BWT) are well-known index structures for collections of strings based on the lexicographic ordering of all their suffixes (see Chapter 2). In order to preserve the identifiability of each string of the collection, it is often assumed that all of them are terminated with distinct sentinels symbols. However, this approach has the drawback of making the alphabet too large. To overcome this technical difficulty, we use a single sentinel symbol and we define the index structures on the lexicographic ordering of all the rotations of all the strings of the collection. This allows to uniquely identify the resulting indexing structures independently from the order of the strings even if a single sentinel symbol is used. Furthermore, at the end of this section we will show how these indexing structures can be easily mimicked using those based on the ordering of the

i	$RT[i]$	$B[i]$	$SA[i]$	$L[i]$
1	$ATT	T	(0, 4)	-1
2	$CAT	T	(0, 3)	0
3	$CCA	A	(0, 2)	0
4	$GCA	A	(0, 1)	0
5	A$CC	C	(1, 2)	0
6	A$GC	C	(1, 1)	1
7	AT$C	C	(2, 3)	1
8	ATT$	$	(3, 4)	2
9	CA$C	C	(2, 2)	0
10	CA$G	G	(2, 1)	2
11	CAT$	$	(3, 3)	2
12	CCA$	$	(3, 2)	1
13	GCA$	$	(3, 1)	0
14	T$AT	T	(1, 4)	0
15	T$CA	A	(1, 3)	1
16	TT$A	A	(2, 4)	1

$q(\texttt{A})$ (rows 5–8), $q(\texttt{AT})$ (rows 7–8)

Figure 6.1: The rotations RT, BWT, GSA, and LCP array of the reads $r_1 = \texttt{GCA}$, $r_2 = \texttt{CCA}$, $r_3 = \texttt{CAT}$, and $r_4 = \texttt{ATT}$. The column RT lists all rotations in lexicographic order, presenting the k-suffixes in black and the remaining prefixes in grey.

suffixes (that are computed by most existing methods).

In the following, we will refer to the list of the sorted rotations of T as RT, to the generalized suffix array of T as SA, to the LCP array of T as L, and to the BWT of T as B. Figure 6.1 shows an example of these data structures over the set $R = \{\texttt{GCA}, \texttt{CCA}, \texttt{CAT}, \texttt{ATT}\}$.

Given a string $Q \in (\Sigma \cup \{\$\})^*$ containing at most a single sentinel symbol \$, notice that all rotations of all the strings in T whose prefix is Q appear consecutively in RT. We call *Q-interval* [6,27] on T (or, simply, Q-interval) the maximal interval $q(Q) = [b, e)$ such that Q is a prefix of $RT[i]$ for each i, $b \leq i < e$. For instance, on the example in Figure 6.1, the interval $q(\texttt{A})$ is $[5, 9)$ and the interval $q(\texttt{AT})$ is $[7, 9)$. It is worth noting that two strings Q_1 and Q_2 might be distinct but $q(Q_1) = q(Q_2)$ (see, for example, $q(\texttt{GC})$

and $q(\texttt{GCA})$ in Figure 6.1). We define the *length* and *width* of the Q-interval $[b, e)$ as $|Q|$ and the difference $e - b$, respectively. Notice that the width of the Q-interval is equal to the number of occurrences of Q as a substring of some string $r \in T$. Whenever the string Q is not specified, we will use the term *string-interval.* For simplicity we assume that $q(\epsilon) = [1, n + 1) = [1, |RT| + 1)$. Nonempty string-intervals $q(Q\$)$ or $q(\$Q)$ have some important properties. In the first case, the string Q is a suffix of some strings in T. Indeed, given the $Q\$$-interval $[b, e)$, then $RT[i][: |Q| + 1] = Q\$$ for each i, $b \leq i < e$. For this reason, we will say that a $Q\$$-interval is associated with the set $T^s(Q)$ of the reads (strings) with suffix Q. In the latter case, the string Q is a prefix of some strings in T. Indeed, given the $\$Q$-interval $[b, e)$, then $RT[i][: |Q| + 1] = \$Q$ for each i, $b \leq i < e$. For this reason, we will say that the $\$Q$-interval is associated with the set $T^p(Q)$ of the reads with prefix Q. Moreover, given a $\$Q$-interval $[b, e)$, we have that $1 \leq b < e \leq m + 1$ since \$ is smaller than any element in Σ, and T contains m reads. Notice that the width of a $Q\$$-interval ($\$Q$-interval, resp.) is equal to the number of reads having Q as suffix (prefix, resp.). Since we are not interested in circular patterns (*i.e.*, strings which match a rotation of a string in T), in the following we will assume that all the string-intervals refers to strings of the form Q, $\$Q$, or $Q\$$ for some $Q \in \Sigma^*$.

Since RT, B, SA, and L are all closely related, a string-interval can be viewed as an interval on any of those arrays. There are some interesting relations between string-intervals and the LCP array. The interval $[i, j)$ of the LCP array is called a *lcp-interval* of value K (shortly K-interval) [1], if $L[i], L[j] < K$, while $L[h] \geq K$ for each h with $i < h < j$ and there exists $L[h] = K$ with $i < h < j$. An immediate consequence is the following proposition.

Proposition 6.1 *Let S be a string over Σ and let $[b, e)$ be the S-interval. Then $L[h] \geq |S|$ for each h with $b < h < e$. Moreover, if $S \in \Sigma^*$ is the longest string whose S-interval is $[b, e)$, then $[b, e)$ is a lcp-interval of value $|S|$.*

Proposition 6.1 relates the notion of string-intervals with that of lcp-intervals. It is immediate to associate to each K-interval $[b, e)$ the string S consisting of the common K-long prefix of all strings $RT[i]$ with $b \leq i < e$. Such string S is called the *representative* of the K-interval. Moreover, given a K-interval $[b, e)$, we will say that b is its *opening position* and that e is its *closing position.*

String-intervals form an inclusion hierarchy, that is no string-interval can partially overlap another one, as showed by the following proposition.

Proposition 6.2 *Let S_1, S_2 be two strings such that the S_2-interval $[b_2, e_2)$ is nonempty, and let $[b_1, e_1)$ be the S_1-interval. Then S_1 is a proper prefix of S_2 if and only if $[b_1, e_1)$ contains $[b_2, e_2)$ and $|S_1| < |S_2|$.*

Proof The only-if direction is immediate, therefore we only consider the case when $[b_1, e_1)$ contains $[b_2, e_2)$ and $|S_1| < |S_2|$. If the containment is proper, the proof is again immediate from the definition of string-interval. Therefore assume that $b_1 = b_2$ and $e_1 = e_2$. Since both S_1 and S_2 are prefixes of all the rotations in $RT[b_1, e_1)$ and since $|S_1| < |S_2|$, we have that S_1 is a proper prefix of S_2. □

Notice that Proposition 6.2 is restricted to nonempty S_2-intervals, since the Q-interval is empty for each string Q that is not a substring of a read in T. Therefore relaxing that condition would falsify Proposition 6.2.

Let B^{rev} be the BWT of the set $T^{rev} = \{r_i^{rev} : r_i \in R\}$, let $[b, e)$ be the Q-interval on T for some string Q, and let $[b^r, e^r)$ be the Q^{rev}-interval on T^{rev}. Then, $[b, e)$ and $[b^r, e^r)$ are called *linked* string-intervals. The linking relation is a 1-to-1 correspondence and two linked intervals have same width and length, hence $e - b = e^r - b^r$.

We recall from the literature [41] that, given a Q-interval for some $Q \in \Sigma^*$ and a symbol $\sigma \in \Sigma^{\$}$, the *backward σ-extension* of the Q-interval is the σQ-interval. We say that a Q-interval has a *nonempty* (*empty*, respectively) backward σ-extension if the resulting interval has width greater than 0 (equal to 0, respectively). Conversely, the *forward σ-extension* of a Q-interval is the $Q\sigma$-interval. Moreover, two important results [42, 77] showed that, given an S-interval on T and the linked S^{rev}-interval on T^{rev}, it is possible to compute the backward σ-extension of the former *and* the forward σ-extension of the latter using only the functions `C` and `Occ` computed on B (hence avoiding to store `C` and `Occ` computed on B^{rev}). We recall that `C` is a function that, given a character as input, counts the number of the characters lexicographically smaller than it in a given set, and that `Occ` is a function that, given a character and a position on a string, count the number of occurrences of that character up to the given position in the string. The following proposition summarizes those results.

Proposition 6.3 *Let S be a string, $q(S) = [b, e)$ be the S-interval on T, $q^{rev}(S^{rev}) = [b^r, e^r)$ be the S^{rev}-interval on T^{rev}, and let σ be a character. Then, the backward σ-extension of $q(S)$ and the forward σ-extension of $q^{rev}(S^{rev})$ are:*

$$q(\sigma S) = [\mathtt{C}(\sigma) + \mathtt{Occ}(\sigma, b) + 1, \mathtt{C}(\sigma) + \mathtt{Occ}(\sigma, e) + 1) \tag{6.1}$$

$$q^{rev}(S^{rev}\sigma) = [b^r + \Sigma_{c<\sigma}(\mathtt{Occ}(c, e) - \mathtt{Occ}(c, b)), b^r + \Sigma_{c\leq\sigma}(\mathtt{Occ}(c, e) - \mathtt{Occ}(c, b)))\,. \tag{6.2}$$

It is immediate to obtain from $\mathtt{C}$ a function $\mathtt{C}^{-1}(i)$ that returns the first character of $RT[i]$ (*i.e.*, $\mathtt{C}^{-1}(i) = \arg\max_{\sigma\in\Sigma^{\$}}\{\mathtt{C}(\sigma) \leq i\}$).

Self-indexes rotations and suffixes -based Notice that, up to now, we have described B, SA, and L on the rotations of the input strings, but most existing methods compute these data structures based on the K-suffixes (for all K) of the input strings. The two definitions are almost equivalent and, indeed, the resulting data structures are almost identical. The only difference between sorting suffixes and rotations is due to the order of identical suffixes. In fact, the traditional definition does not impose an order in that case, whereas we impose a specific order of identical suffixes. This order guarantees that Proposition 6.2 holds also when $\sigma = \$$. Moreover, the order of identical suffixes has no impact if we focus our attention only on string-intervals, since the definition of string-interval does not depend on such order. To show the equivalence between the two definitions, we will now show a simple yet efficient linear time algorithm that will allow us to modify the data structures produced considering the suffixes in order to correctly identify each string r_j of T by performing a backward \$-extension of the $r_j\$$-interval.

The goal is to (efficiently) sort the suffixes that are equal to the sentinel \$ (corresponding to the positions i such that $SA[i] = (0, \cdot)$) according to the lexicographic order of the reads. In other words, we enforce that, for each i_1, i_2 where $SA[i_1] = (0, j_1)$, $SA[i_2] = (0, j_2)$, if $i_1 < i_2$ then r_{j_1} lexicographically precedes r_{j_2}. Since the reads are already listed in lexicographic order in the GSA, we can reconstruct such order with a coordinated scan of the GSA and of the BWT, exploiting the fact that $B[i] = \$$ and $SA[i] = (k, j)$ iff the read r_j is k long (see Algorithm 1). Since we are not interested in searching patterns that include the sentinel symbol in the middle (*i.e.*, circular patterns), this procedure, which is executed just after the construction of SA and B, updates SA

Algorithm 1: Reorder the first m entries of SA according to the lexicographic order of the input reads.

Input : The BWT B and the GSA SA of the set R.
Output: An updated SA such that for each i_1, i_2 where $SA[i_1] = (0, j_1)$, $SA[i_2] = (0, j_2)$, if $i_1 < i_2$ then r_{j_1} lexicographically precedes r_{j_2}.

```
1 p ← 1;
2 foreach i do
3 |  if B[i] = $ then
4 |  |  (k, j) ← SA[i];
5 |  |  SA[p] ← (0, j);
6 |  |  p ← p + 1;
```

in order to simulate the data structures produced on the rotations of the input strings. For this reason the order of the input strings does not affect in any way our approaches and we will interchangeably refer to suffixes and rotations from now on. Notice that the procedure performs a single sequential scan of B and SA and requires only $O(1)$ additional internal memory.

Overlap graph and string graph We recall that the *overlap graph* is a graph that represents the overlap relationships among the strings in the collection T. Given two strings $r_i, r_j \in T$, we say that r_i *overlaps* r_j iff a nonempty suffix S of r_i is also a prefix of r_j, that is, $r_i = PS$ and $r_j = SX$. In that case we say that S is the *overlap* of r_i and r_j, denoted as $ov_{i,j}$, that X is the *right extension* of r_i with r_j, denoted as $rx_{i,j}$, and P is the *left extension* of r_j with r_i, denoted as $lx_{i,j}$ (see Figure 6.2). The overlap graph is defined as follows.

Definition 6.1 (Overlap graph) *Given a set T of reads, its* overlap graph *[97] is the directed graph $G_O = (T, A)$ whose vertices are the reads in T, and where two reads r_i, r_j form the arc (r_i, r_j) if they have a nonempty overlap. Moreover, each arc (r_i, r_j) of G_O is labeled by the left extension ($lx_{i,j}$) and the right extension ($rx_{i,j}$) of r_j with r_i.*

The main use of an overlap graph is to compute the *assembly* of each path, corresponding to the sequence that can be read by traversing the reads corresponding to vertices of the path. More formally, given a path $r_{i_1}, r_{i_2}, \ldots, r_{i_K}$ of G_O, its assembly can be spelled

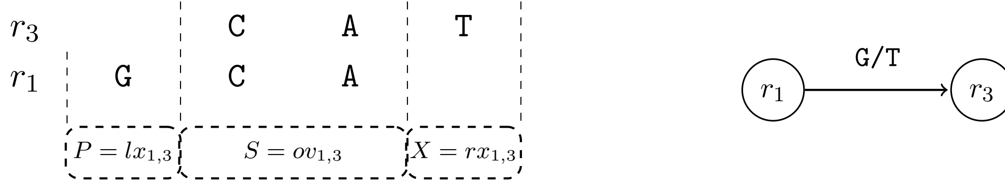


Figure 6.2: Example of an overlap between the reads r_1, r_3 of Figure 6.1 (left) and the associated arc of the overlap graph (right). According to Definition 6.1, the label of the arc is G/T since G is the left extension and T is the right extension of the overlap. The assembly of the path r_1, r_3 is $lx_{1,3}r_3 = r_1rx_{1,3} =$ GCAT.

either by the string $lx_{i_1,i_2}lx_{i_2,i_3}\cdots lx_{i_{K-1},i_K}r_{i_K}$ or the string $r_{i_1}rx_{i_1,i_2}rx_{i_2,i_3}\cdots rx_{i_{K-1},i_K}$ (we will show shortly that these two strings are actually the same).

Figure 6.2 depicts the overlap graph of the set of reads $\{r_1, r_3\}$ with $r_1 =$ GCA and $r_3 =$ CAT. The assembly of the path r_1, r_3 is $lx_{1,3}r_3 =$ GCAT $= r_1rx_{1,3}$.

We will now show that labeling the arcs with both lx and rx is not necessary if we do not consider reverse complemented reads since either one of the two informations suffices.

Lemma 6.4 *Let G_O be the overlap graph for T and let $r_{i_1}, r_{i_2}, \ldots, r_{i_K}$ be a path of G_O. Then, $lx_{i_1,i_2}lx_{i_2,i_3}\cdots lx_{i_{K-1},i_K}r_{i_K} = r_{i_1}rx_{i_1,i_2}rx_{i_2,i_3}\cdots rx_{i_{K-1},i_K}$.*

Proof We will prove the lemma by induction on K. Let (r_h, r_j) be an arc of G_O. Notice that the path r_hr_j represents $lx_{h,j}ov_{h,j}rx_{h,j}$. Since $r_h = lx_{h,j}ov_{h,j}$ and $r_j = ov_{h,j}rx_{h,j}$, the case $K = 2$ is immediate.

Assume now that the lemma holds for paths of length smaller than K and consider the path $r_{i_1}, \ldots, r_{i_K}$. The same argument used for $K = 2$ shows that $lx_{i_1,i_2}lx_{i_2,i_3}\cdots lx_{i_{K-1},i_K}r_{i_K}$ $= lx_{i_1,i_2}lx_{i_2,i_3}\cdots lx_{i_{K-2},i_{K-1}}r_{i_{K-1}}rx_{K-1,K}$. Moreover, by inductive hypothesis we have that $lx_{i_1,i_2}lx_{i_2,i_3}\cdots lx_{i_{K-2},i_{K-1}}r_{i_{K-1}}rx_{K-1,K} = r_{i_1}rx_{i_1,i_2}rx_{i_2,i_3}\cdots rx_{i_{K-2},i_{K-1}}rx_{i_{K-1},i_K}$, completing the proof. □

This definition models the actual use of overlap graphs to reconstruct a genome [97]. If we have perfect data and no relevant repetitions, the overlap graph is a directed acyclic graph (DAG) with a unique topological sort, which in turn reveals a peculiar structure: the graph is made of tournaments [31]. More formally, let $\langle r_1, \ldots, r_n \rangle$ be the topological

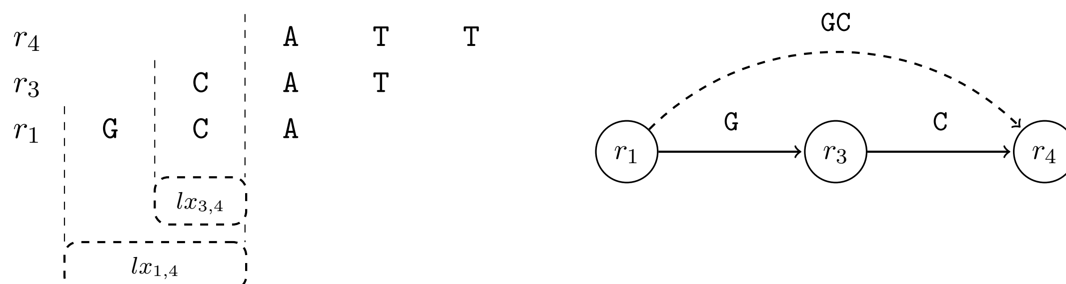


Figure 6.3: Example of a reducible arc of the overlap graph on the subset $\{r_1, r_3, r_4\}$ of the reads of Figure 6.1 labeled using only left extensions. The arc (r_1, r_4) is reducible since the assembly of r_1, r_4 is the same of r_1, r_3, r_4 (GCATT).

order of G_O. If (r_i, r_j) is an arc of G_O then also all (r_h, r_k) with $i < h < k < j$ are arcs of G_O. Notice that in this case, all paths from r_i to r_j have the same assembly. This suggests that it is possible (and desirable) to remove some arcs of the overlap graph without modifying the set of distinct assemblies that can be read from the graph. An arc (r_i, r_j) of G_O is called *reducible* [97] if there exists another path from r_i to r_j with the same assembly (*i.e.*, the string $lx_{i,j}r_j$). After removing all reducible arcs from the overlap graph, we obtain the *string graph* [97]. Figure 6.3 depicts the overlap graph of the subset $\{r_1, r_3, r_4\}$ of the reads of Figure 6.1. Since the assembly of the path r_1, r_4 is the same of that of r_1, r_3, r_4, then the arc (r_1, r_4) is reducible. Notice that the label of the arc (r_3, r_4) is a suffix of the label of the arc (r_1, r_4). The following lemma, introduced but not formally proved in [11], proves that this condition is necessary and sufficient to recognize reducible arcs.

Lemma 6.5 *Let G_O be the labeled overlap graph for a substring-free set T of reads and let (r_i, r_j) be an arc of G_O. Then, (r_i, r_j) is reducible iff there exists another arc (r_h, r_j) of G_O incoming in r_j and such that $lx^{rev}_{h,j}$ is a proper prefix of $lx^{rev}_{i,j}$.*

Proof First notice that $lx^{rev}_{h,j}$ is a proper prefix of $lx^{rev}_{i,j}$ iff and only if $lx_{h,j}$ is a proper suffix of $lx_{i,j}$. By definition, (r_i, r_j) is reducible if and only if there exists a second path $r_i, r_{h_1}, \ldots, r_{h_k}, r_j$ representing the string XYZ, where X, Y and Z are respectively the left extension of r_j with r_i, the overlap of r_i and r_j, and the right extension of r_i with r_j. Assume that such a path $(r_i, r_{h_1}, \ldots, r_{h_k}, r_j)$ exists. Since $r_i, r_{h_1}, \ldots, r_{h_k}, r_j$ represents XYZ and $Z = rx_{i,j}$, $r_{h_k} = X_1YZ_1$ where X_1 is a suffix of X and Z_1 is a proper prefix

of Z. Notice that $X_1 = lx_{h_k,j}$ and R is substring free, hence X_1 is a proper suffix of X, otherwise r_i would be a substring of r_{h_k}, completing this direction of the proof. Assume now that there exists an arc (r_h, r_j) such that $lx_{h,j}$ is a proper suffix of $lx_{i,j}$. Again, $r_h = X_1Y_1Z_1$ where X_1, Y_1 and Z_1 are respectively the left extension of r_j with r_h, the overlap of r_h and r_j, and the right extension of r_h with r_j. By hypothesis, X_1 is a proper suffix of X. Since r_h is not a substring of r_i, the fact that X_1 is a suffix of X implies that Y is a substring of Y_1, therefore r_i and r_h overlap and $|ov_{i,h}| \geq |Y|$, hence (r_i, r_h) is an arc of G_O. The string associated to the path (r_i, r_h, r_j) is $r_i rx_{i,h} rx_{h,j}$. By Lemma 6.4, $r_i rx_{i,h} rx_{h,j} = lx_{i,h} lx_{h,j} r_j$. At the same time the string associated to the path r_i, r_j is $r_i rx_{i,j} = lx_{i,j} r_j$ by Lemma 6.4, hence it suffices to prove that $lx_{i,h} lx_{h,j} = lx_{i,j}$. Since $lx_{h,j}$ is a proper suffix of $lx_{i,j}$, by definition of left extension, $lx_{i,h} lx_{h,j} = lx_{i,j}$, completing the proof. □

We can transform Lemma 6.5 into an easily testable property, by way of Proposition 6.2, that is (r_x, r_z) is reducible iff there exists an arc (r_s, r_z) such that $q^r(lx^r_{s,z})$ includes $q^r(lx^r_{x,z})$ and $|lx_{x,z}| > |lx_{s,z}|$. The following lemma shows that, if (r_x, r_z) can be reduced, then it can be reduced by an arc of the string graph G_S, hence avoiding a comparison between all pairs of arcs of G_O.

Lemma 6.6 *Let G_O be the overlap graph of a set T of reads, let G_S be the corresponding string graph, and let (r_x, r_z) be an arc of G_O that is not an arc of G_S. Then there exists an arc (r_s, r_z) of G_S such that $q^r(lx^r_{s,z})$ includes $q^r(lx^r_{x,z})$ and $|lx_{x,z}| > |lx_{s,z}|$.*

Proof Let (r_s, r_z) be the arc of G_O whose left extension is the shortest among all arcs of G_O such that $q^r(lx^r_{s,z})$ includes $q^r(lx^r_{x,z})$ and $|lx_{x,z}| > |lx_{s,z}|$. By Lemma 6.5, since (r_x, r_z) is not an arc of G_S such an arc must exist. We want to prove that (r_s, r_z) is an arc of G_S.

Assume to the contrary that (r_s, r_z) is not an arc of G_S, that is there exists an arc $e_1 = (r_h, r_z)$ of G_O such that $q^r(lx^r_{s,z})$ includes $q^r(lx^r_{h,z})$ and $|lx_{s,z}| > |lx_{h,z}|$. Then $q^r(lx^r_{x,z})$ includes $q^r(lx^r_{h,z})$ and $|lx_{x,z}| > |lx_{h,z}|$, contradicting the assumption that (r_s, r_z) is the arc of G_O whose left extension is the shortest among all arcs of G_O such that $q^r(lx^r_{s,z})$ includes $q^r(lx^r_{x,z})$ and $|lx_{x,z}| > |lx_{s,z}|$. □

Lemma 6.6 highlights a fundamental notion that we will use in our work. Given a set of reads T, if we want to build its overlap graph and reduce it to its string graph we don't need to store neither the labels of the arcs nor the sequences of the nodes. Indeed, we can perform the graph reduction by labeling the arcs by string-intervals on the BWT of T and the lengths of the labels.

The consequences of Lemma 6.6 are twofold: we need less space to store the information related to the labels and we can test faster if an arc is reducible.

More precisely, the labels can be stored using $|E|(2\lg n + \lg \ell_{\max})$ bits instead of the $|E|\ell_{\max} \lg \sigma$ bits required by the plain representation of the strings[1]. Note that if we consider $\lg n$ to be the word length, the two space requirements become respectively $3|E|$ and $\ell_{\max}|E|(\lg \sigma / \lg n) =\sim \ell_{\max}|E|$. Moreover, we can test if an arc e_i can be reduced by another arc e_j in constant time. Indeed, if $l(e_i)$ is the label of e_i, $[b_i, e_i)$ is its string-interval, $l_i = |l(e_i)|$ is its length, and $[b_j, e_j)$ and l_j are respectively the string-interval and the length of the label of e_j $(l(e_j))$, then we can test if e_j is reduced by e_i by checking in constant time if $b_i \leq b_j < e_j \leq e_i$ and $l_i < l_j$ whereas testing if $l(e_i)$ is a prefix of $l(e_j)$ requires $\mathcal{O}(\ell_{\max})$.

Notice that Lemma 6.6 does not consider reverse complemented reads although they are produced by current sequencing methods. Nevertheless, our practical implementations consider this reads and we will show how we can avoid these problems by doubling the informations for each arc and by implicitly merging nodes in the graph.

In the next two chapters we will show how this model can be used to design methods that pursue different — almost orthogonal — goals, hence proving its usefulness and pliability.

[1]recall that $n = \sum_{i=1}^{i=|T|}(|r_i| + 1)$.

7 Lightweight external memory assembly algorithm

In this section we will present the first implementation of the framework we defined in the previous chapter. The main goal of this tool, called LightStringGraph (*LSG*), is to reduce the main memory usage of the graph construction of string graph based assemblers in order to run such step on resource-limited machines.

The main motivation behind this work is that a single experiment performed with current sequencing technologies produces hundreds of Gigabyte of data and it is usually required to have access to powerful workstations in order to compute assemblies even when the data is compressed. Since nowadays assembling genomes is almost an everyday task for biological analysis and the advent of personalized medicine in sight, it is worth to study possible methods that are able to cope with a limited amount of RAM, *i.e.*, are able to run even on laptops. Moreover, since the cost of solid state hard drives has plummetted, LSG heavily rely on them and aims to maintain the data in external memory and to lower I/O as much as possible.

We developed an open source implementation of our algorithm and we have integrated it in SGA to obtain a complete string graph based assembler. To the best of our knowledge, LSG is the first disk-based exact algorithm to construct a string graph that does not require to have the whole original dataset in main memory. Clearly, an external-memory algorithm usually requires more time than an in-memory approach, even when disk accesses are minimized and made sequential. However, this kind of investigation could greatly help to achieve good compromises between the use of time and space when processing a huge number of reads.

7.1 Background

As for other efficient and widely used assemblers [79, 134, 135], LSG is based on the Burrows-Wheeler Transform and, obviously, requires to build it as a initial step. There are plenty of BWT construction algorithms (see [5, 6, 39, 80, 84] and references therein for a comprehensive review of the field). LSG requires to compute also the LCP array in order to speed up the computation of the overlaps between the reads. Moreover, we strive to produce a string graph building pipeline that is able to compute it in low memory from start to end. We chose BEETL [6] as BWT building tool for LSG since it builds such data structure along the LCP array using a negligible amount of memory.

In order to evaluate the performance of the LSG algorithm, we have compared our string graph construction tool with the one implemented in SGA [135], a current state-of-the-art string-graph assembler. As stated previously, SGA is a genome assembler that uses the FM-index to output all the irreducible edges of a string graph. First, it builds the FM-index of all the reads and of their reverse. Then, for each read, it searches overlaps of it by querying the index to find suffixes of the read that can be backward extended with the symbol $ and saves the string-intervals on the FM-index of the reverse of the reads of each of these suffixes. Lastly, SGA backward extend all the stored intervals character by character on the second FM-index (actually performing a *forward extension*), outputs the ones that can be extended with $, and filters the ones related to reducible arcs.

The experimental analysis over a large dataset (875 million reads) extracted from the human genome shows that LSG is able to build the string-graph on a standard workstation using 1GB of main memory, while SGA has a peak usage of 43GB. Moreover, since LSG can replace the string graph construction in a string-graph assembler such as SGA, we have implemented a LSG-based assembler by designing a pipeline that uses BEETL for indexing reads, LSG for constructing the graph, and the SGA module that builds the final assembly from the string-graph (`sga-assemble`).

We compared our BEETL+LSG+SGA assembler with SGA and Readjoiner, *i.e.*, two of the most known string-graph assemblers. The experimental results on the aforementioned dataset show that the overlap detection and string graph reduction steps can be performed using a negligible amount of main memory (about 2% of that used by SGA)

at the cost of about 3x increase in running time. This analysis shows that LSG is a competitive approach for the construction of assembly graphs from large NGS dataset, even though its benefits would be more evident if a memory efficient version of more steps of the pipeline were available.

7.1.1 Definitions

We briefly recall some fundamental definitions that will be used in the following. Given two strings (s_i, s_j), we say that s_i *overlaps* s_j iff a nonempty suffix Z of s_i is also a prefix of s_j, that is $s_i = XZ$ and $s_j = ZY$. In that case we say that Z is the *overlap* of s_i and s_j, denoted as $ov_{i,j}$, that Y is the *extension* of s_i with s_j, denoted as $ex_{i,j}$ (called $rx_{i,j}$ in the previous chapter), and X is the *prefix-extension* of s_i with s_j, denoted as $pe_{i,j}$ (called $lx_{i,j}$ in the previous chapter).

In the following of the chapter we will consider a collection $S = \{s_1, \ldots, s_n\}$ of n reads (*i.e.*, strings) over Σ. We append a sentinel symbol $\$ \notin \Sigma$ to the end of each string ($ lexicographically precedes all symbols in Σ) and we denote by $\Sigma^{\$}$ the extended alphabet $\Sigma \cup \{\$\}$. We assume that the sentinel symbol $ is not taken into account when computing overlaps between two strings.

Given a string Q, all suffixes of the GSA whose prefix is Q appear consecutively in GSA, where they induce an interval $[b, e)$ which is called Q*-interval* [6] and denoted by $q(Q)$. We define the *length* and *width* of the Q-interval $[b, e)$ as $|Q|$ and the difference $e - b$, respectively. Notice that the width of the Q-interval is equal to the number of occurrences of Q as a substring of some string $s \in S$. Whenever the string Q is not specified, we use the term *string-interval* to point out that it is the interval on the GSA of all suffixes having a common prefix. Since the BWT and the GSA are closely related, we also say that $[b, e)$ is a string-interval (or Q-interval for some string Q) on the BWT. Let B^{rev} be the BWT of the set $S^{rev} = \{s^{rev} \mid s \in S\}$, let $[b, e)$ be the Q-interval on B for some string Q, and let $[b', e')$ be the Q^{rev}-interval on B^{rev}. Then, $[b, e)$ and $[b', e')$ are called *linked* string-intervals. The linking relation is a 1-to-1 correspondence and two linked intervals have same width and length, hence $e - b = e' - b'$. Finally notice that, given two strings Q_1 and Q_2 s.t. $Q_1 \neq Q_2$, the Q_1-interval $q(Q_1)$ and the Q_2-interval $q(Q_2)$ are either contained one in the other (possibly they are the same) or are disjoint.

If $q(Q_1)$ is contained in $q(Q_2)$, the string Q_2 is a prefix of $Q_1 = Q_2Q'$. Moreover, if $q(Q_1) = q(Q_2)$ then the string Q' follows Q_2 in all reads of S.

Given a Q-interval and a symbol $a \in \Sigma$, the *backward a-extension* of the Q-interval is the aQ-interval (that is, the interval on the GSA of the suffixes sharing the common prefix aQ). We say that a Q-interval has a *nonempty* (*empty*, respectively) backward a-extension if the resulting interval has width greater than 0 (equal to 0, respectively). Conversely, the *forward a-extension* of a Q-interval is the Qa-interval. The backward \$-extension of the Q-interval represents the set of reads whose prefix is Q, while the forward \$-extension of the Q-interval is represents the set of reads whose suffix is Q. The trivial extension of the functions `C` and `Occ` to the set $\Sigma^{\$}$ allows to compute also the backward and forward \$-extensions, provided that the \$-interval refers to the reads in lexicographic order (see Algorithm 1 for a linear time algorithm that reorders the entries of the GSA/BWT/FM-index accordingly). With a slight abuse of language, we will denote with $q(\$Q)$ and $q(Q\$)$ respectively the backward and the forward \$-extension of the Q-interval.

The *overlap graph* [97] of S is the directed graph $G_O = (S, A)$ whose vertices are the reads in S, and two reads s_i, s_j form the arc (s_i, s_j) if they overlap. Moreover, each arc (s_i, s_j) of G_O is labeled by the left extension $ex_{i,j}$ of s_i with s_j. Each path $(s_1, \dots, s_k)$ in G_O represents a string α that is obtained by assembling the reads of the path. If two nodes are connected by two paths, one that connect the nodes directly and one that visits a third node, and if the two paths represent the same string, we say that the arc that directly connects the two nodes is *reducible*. Reducible arcs are not helpful in assembling reads, therefore we can — and want — to remove (or avoid computing) them. After removing all reducible arcs from the overlap graph, we obtain a *string graph* [97]. Moreover, since short overlaps are likely to appear by chance, they are not meaningful for assembling the original sequence. Hence, we will consider only overlaps at least τ long, where τ is a positive constant.

7.2 Methods

As previously stated, we assume that the set S of the reads is *substring-free*, that is, there are no two reads $s_1, s_2 \in S$ such that s_1 is a substring of s_2. Moreover we base our algorithm on a space efficient representation of reads by string-intervals: we use a string-interval $q(Q)$ to represent all the reads having suffixes starting with the prefix Q.

Let us denote by $R^s(\alpha)$ and $R^p(\alpha)$ the sets of reads whose suffix and prefix (respectively) is a given string α. Let s_i and s_j be two reads and let α be a string with $|\alpha| \geq \tau$. Then $s_i \in R^s(\alpha)$, $s_j \in R^p(\alpha)$ if and only if (s_i, s_j) is an arc of $G_O = (S, A)$ whose overlap is α.

Observe that the set $R^p(\alpha)$ is represented by $q(\$\alpha)$, while the set $R^s(\alpha)$ of the reads is represented by $q(\alpha\$)$. Then, the arc set of the overlap graph can be represented by the set of all the pairs $(q(\alpha\$), q(\$\alpha))$, s.t. $|\alpha| \geq \tau$. In particular, a pair $(q(\alpha\$), q(\$\alpha))$ represents the set $A_\alpha \subseteq A$ of the arcs whose overlap is α. In the following, we will refer to an α-interval with nonempty backward and forward \$-extensions as an *overlap-interval*. Observe that the interval $q(\beta\alpha\$)$ represents the reads whose suffix is $\beta\alpha$ (which are a subset of the reads whose suffix is α). Notice that the pair $(q(\beta\alpha\$), q(\$\alpha))$ represents the arcs of A_α outgoing from reads with suffix $\beta\alpha$. When $q(\beta\alpha\$)$ has a nonempty backward \$-extension, then it represents the read $r = \beta\alpha$ (which is unique under the *substring-free* assumption), and the pair $(q(\beta\alpha\$), q(\$\alpha))$ represents the arcs of A_α with prefix-extension (or label) β.

As stated in the main lemma of the previous chapter (Lemma 6.6), as a main consequence of our definition of reducible arc, we need to associate to an arc (s_i, s_j) the reverse of the prefix-extension given in compact form by the $lx^{rev}_{i,j}$-interval on B^{rev}. By using the stated characterization of reducible arcs, the arc (s_i, s_j) is *reducible* if and only if there exists an arc (s_h, s_j) such that the $lx^{rev}_{h,j}$-interval contains the $lx^{rev}_{i,j}$-interval or they are the same but $|lx^{rev}_{h,j}| < |lx^{rev}_{i,j}|$.

Therefore, in order to characterize the arc reducibility in terms of string-intervals, we introduce the following notion of *arc-pair* and *label-pair*, which are fundamental to build arcs and labels of the overlap graph.

Definition 7.1 *Let B be the BWT for the collection S of reads and let B^{rev} be the BWT*

for the set S^{rev} of the reversed reads. Let α and β be two strings s.t. $|\alpha| \geq \tau$. Then, the arc-pair *associated to (β, α) is the pair $(q(\beta\alpha\$), q(\$\alpha))$. The strings α and β are called the* overlap *and the* p-extension *of the arc-pair, respectively. An arc-pair is* terminal *if $q(\beta\alpha\$)$ has a nonempty backward \$-extension (hence β is a complete prefix-extension). An arc-pair is* basic *if β is the empty string ϵ.*

The pair $(q(\beta), q^{rev}(\beta^{rev}))$, where $q(\beta)$ and $q^{rev}(\beta^{rev})$ are the β-interval on B and the β^{rev}-interval on β^{rev} (respectively), is called the label-pair *of the arc-pair associated to (β, α).*

For simplicity, we also use (β, α) to denote the arc-pair associated to such pair of strings. A basic arc-pair $(\epsilon, \alpha) = (q(\alpha\$), q(\$\alpha))$ represents (as described above) the arcs of the overlap graph with overlap α. Two arcs of the overlap graph with the same overlap α may have different prefix-extensions, let us say β and γ, thus they are represented by two different terminal arc-pairs $(q(\beta\alpha\$), q(\$\alpha))$ and $(q(\gamma\alpha\$), q(\$\alpha))$. Observe that a terminal arc-pair cannot have a nonempty backward a-extension where $a \neq \$$.

Our algorithm for building the string graph is composed of three steps: (i) computing the unlabeled overlap graph, (ii) labeling the arcs of the overlap graph, and (iii) reducing the overlap graph to the string graph.

Step (i) consists in computing all basic arc-pairs, while step (ii) labels the overlap graph by computing all terminal arc-pairs and the related label-pairs $(q(\beta), q^{rev}(\beta^{rev}))$ — we recall that the information on $q^{rev}(\beta^{rev})$ is necessary to reduce the overlap graph. The main idea of step (ii) is to start from the basic arc-pairs computed in step (i) to obtain the terminal arc-pairs (and label-pairs) via a sequence of backward extensions. Finally, the step (iii) reduces the overlap graph to the string graph by using the label-pairs computed in step (ii).

Computing the unlabeled overlap graph. In order to compute the set of all basic arc-pairs $(\epsilon, \alpha) = (q(\alpha\$), q(\$\alpha))$ we have designed the procedure **BuildBasicArcPairs** (Algorithm 2). This procedure is based on a single synchronous scan of the BWT B, the GSA SA, and the LCP array L of the set S.

In the following, we call *overlap-candidate* (or simply *candidate*) a string-interval (at least τ long) with a nonempty forward \$-extension. An overlap-candidate with a

Algorithm 2: BuildBasicArcPairs

Input : The BWT B, the LCP array L and the GSA SA of the set R.

Output: The set of the basic arc-pairs.

1 $\#_\$ \leftarrow 0$;
2 **if** $B[1] = \$$ **then** $\#_\$ \leftarrow 1$;
3 $p \leftarrow 2$;
4 $Z \leftarrow$ empty stack;
5 **while** $p \leq |L|$ **do**
6   **if** $L[p] > L[p-1]$ *and* $L[p] \geq \tau$ **then** // p is an opening position
7         $(k, j) \leftarrow SA[p-1]$;
8   **if** $L[p] = k$ **then** // $[p-1, x)$ is an overlap-candidate
9             $q \leftarrow p$;
10             $(k^*, j) \leftarrow SA[q]$;
11   **while** $L[q] = L[p] = k^* = k$ **do**
12                 $q \leftarrow q + 1$;
13                 $(k^*, j) \leftarrow SA[q]$;
14   push $([p-1, q), k, \#_\$)$ to Z;
15             $p \leftarrow q$;
16   **if** $L[p] < L[p-1]$ **then** // p is a closing position
17         $([b, e_1), \ell_S, b_\$) \leftarrow \text{top}(Z)$;
18   **while** Z *is not empty and* $\ell_S > L[p-1]$ **do**
19   **if** $\#_\$ > b_\$$ **then**
20   output $([b, e_1), [b_\$, \#_\$))$;
21   pop(Z);
22             $([b, e_1), \ell_S, b_\$) \leftarrow \text{top}(Z)$;
23   **if** $B[p] = \$$ **then** $\#_\$ \leftarrow \#_\$ + 1$;
24     $p \leftarrow p + 1$;
// Processing remaining records of Z
25 $([b, e_1), \ell_S, b_\$) \leftarrow \text{top}(Z)$;
26 **while** Z *is not empty* **do**
27   **if** $\#_\$ > b_\$$ **then** output $([b, e_1), [b_\$, \#_\$))$;
28   pop(Z);
29     $([b, e_1), \ell_S, b_\$) \leftarrow \text{top}(Z)$;

nonempty backward \$-extension is an overlap-interval. Given an α-interval $[b, e)$, if α is an overlap, then we can compute the $\alpha\$$-interval by finding the largest interval $[b, e_1)$ (starting in b) such that $L[i] = |\alpha|$ for $b < i \leq e_1$ and $SA[i] = (|\alpha|, \cdot)$ for $b \leq i < e_1$. For our purposes, we say that a position p is:

- an *opening position*, if $L[p] > L[p-1]$ and $L[p] \geq \tau$ (*i.e.*, the LCP value increases). The position $p-1$ is the start of a set of string-intervals $[p-1, \cdot)$ of length $> L[p-1]$ and $\leq L[p]$. It is easy to prove that, among those intervals, only the interval of length $L[p]$ may be an *overlap-candidate*. We call that interval as *opening interval* related to p.
- a *closing position*, if $L[p] < L[p-1]$ (*i.e.*, the LCP value decreases). Then, p is the end of a set of string-intervals $[\cdot, p)$ of length strictly larger than $L[p]$ and smaller than (or equal to) $L[p-1]$. In this case we will call such intervals as *closing* in p.

A stack Z (initially empty) is used to store the *overlap-candidates* and, for each position p, the number $\#_\$$ of symbols \$ in the prefix $B[1 : p-1]$ is maintained. At each position p, the behavior of the procedure depends on the type of p.

When p is an *opening position*, the procedure reads B, SA and L, to compute the forward \$-extension $[p-1, e_1)$ of the *opening interval*. If that extension is nonempty, then the *opening interval* is an *overlap-candidate*, and the following data are recorded on top of Z (see line 15): the forward \$-extension $[p-1, e_1)$, the length of the overlap-candidate referred to as *opening length*, and the number $\#_\$$ of symbols \$ in $B[1 : p-1]$ referred to as *opening \$-number*. Observe that, at this step, the end of the overlap-candidate is unknown and will be determined at some next step when its nonempty backward \$-extension is computed.

When a *closing position* p is read, the stack Z contains the records related to overlap-candidates starting before p and ending in a position $e \geq p$. In particular, the records are sorted by increasing value of *opening length*, with the highest value on top. In other words, the records form a nested hierarchy of the related *overlap-candidate* intervals, where the top record is related to the longest overlap-candidate. Let $OI(p)$ be the set of the overlap-intervals *closing* in p (*i.e.*, ending in p). The $|OI(p)|$ top records in Z, together with the closing position p, contain the information that is necessary to reconstruct those overlap-intervals. Observe that for those records, the length of the related overlap-candidates (*i.e.*, the *opening length*) must be between $L[p]+1$ and $L[p-1]$, inclusive. This condition is used to extract from Z the set $OI(p)$ (see the while cycle at lines 18–22).

For each record in $OI(p)$, the backward \$-extension of the related overlap-candidate

is computed. Let $[b, e_1)$ and $b_\$$ be the forward \$-extension and the *opening \$-number*, respectively. Notice that the overlap-candidate is $[b, p)$ and $b_\$$ is the number of occurrences of \$ in $B[1, b]$. Moreover, under the assumption that no input read is a substring of another read, the BWT B cannot contain any \$ in $[b, e_1)$, thus $b_\$$ is equal to the number of occurrences of \$ in $B[1, b-1]$ (*i.e.*, $\texttt{Occ}(\$, b)$). We maintain a variable $\#_\$$ containing the number of occurrences of \$ in $B[1, p-1]$ (*i.e.*, $\texttt{Occ}(\$, p)$). Then, the backward \$-extension of $[b, p)$ is $[b_\$, \#_\$)$: it is clearly nonempty if and only if $\#_\$ > b_\$$. In that case, the basic arc-pair $([b, e_1), [b_\$, \#_\$])$ is output (see lines 20–21).

Observe that, when the last position is not a closing position, after the while cycle at lines 5–24, the stack Z may be nonempty. The while cycle at lines 25–29 empties the stack Z if necessary.

Notice that the stack Z can contain at most l records (more precisely, when the current position is i, the stack Z contains at most $L[i]$ records). Therefore, the memory used by Z can be considered negligible for our purposes.

Labeling the overlap graph. In the previous step all basic arc-pairs representing the unlabeled arcs of the overlap graph are computed. In this step the procedure **LabelOverlapGraph** (Algorithm 3) labels the overlap graph by finding all terminal arc-pairs and their related label-pairs. In the following, we say that $|\beta\alpha\$|$ is the length of the arc-pair $(q(\beta\alpha\$), q(\$\alpha))$ and $|P|$ is the length of the label-pair $(q(\beta), q^{rev}(\beta^{rev}))$. Furthermore, we say that the backward a-extension of an arc-pair is the arc-pair $(a\beta, \alpha) = (q(a\beta\alpha\$), q(\$\alpha))$, and is associated to an extended label-pair $(q(a\beta), q^{rev}(\beta^{rev}a))$. Observe that the backward a-extension of a basic arc-pair (ϵ, α) is the arc-pair (a, α), and that the label-pair associated to a basic arc-pair is $(q(\epsilon), q^{rev}(\epsilon) = q(\epsilon))$. Therefore the backward a-extension of a basic arc-pair will be associated to the label-pair $(q(a), q^{rev}(a) = q(a))$ that can be easily computed by the value $\texttt{C}(a)$.

In order to discover all terminal arc-pairs and their label-pairs, our procedure iteratively backward extends basic and non-basic arc-pairs of increasing length. More precisely, at a given iteration, the procedure backward extends all arc-pairs (β, α), with a given length $\ell = |\beta\alpha\$|$, as well as their related label-pairs $(q(\beta), q^{rev}(\beta^{rev}))$. The aim is to discover which arc-pairs are terminal (that is which labels have been completely

Algorithm 3: LabelOverlapGraph

Input : The set of the basic arc-pairs.
Output: The labeled overlap graph.

```
1  k ← τ + 1;
2  AP_k ← basic arc-pairs of length k;
3  while AP_k is not empty do
4      foreach (β, α) ∈ AP_k do
5          if (β, α) is terminal then
6              output ⟨{βα} × R^p(α), q^rev(β^rev)⟩;
7          else
8              foreach a ∈ Σ do
9                  if (aβ, α) is nonempty then
10                     add (aβ, α) to AP_{k+1};
11                     a-extend the label-pair (q(β), q^rev(β^rev));
12         k ← k + 1;
13         AP_k^b ← basic arc-pairs of length k;
14         AP_k ← AP_k ∪ AP_k^b;
```

computed) and to produce the arc-pairs $(a\beta, \alpha)$ of length $\ell + 1$ as well as their related label-pairs $(q(a\beta), q^{rev}(\beta^{rev}a))$, which will be processed in the following iteration. Notice that an arc-pair (β, α) of length ℓ has an overlap α of $\ell - 1$ symbols (when $\beta = \epsilon$ and the arc-pair is basic) or strictly shorter than $\ell - 1$ (when it is not basic).

Whenever a terminal arc-pair (β, α) is discovered (observe that it cannot be basic), then the set of arcs (outgoing from the unique read $s = \beta\alpha$), with overlap α and prefix-extension β, is output together with the string-interval $q^{rev}(\beta^{rev})$ contained in its label-pair. Recall that $q^{rev}(\beta^{rev})$ will be necessary in the next step to reduce the overlap graph.

Reducing the overlap graph to the string graph. Let A_s be the arc set of the string graph, which is initially empty. Moreover, the previous phases output the arcs (and the labels) of the overlap graph in several lists, where each list contains the arcs with the same length. The lists of the arcs of the overlap graph (computed in the previous step) are read by increasing length ℓ_p (starting from $\ell_p = 1$): an arc $e = (s_i, s_j)$ is put into A_s, iff there does not exist an arc that (i) is already in A_s and (ii) reduces e as stated before.

In fact, let (s_i, s_j) and (s_h, s_j) be two arcs of the overlap graph whose prefix-extensions are respectively p_{ij} and p_{hj}. Testing whether (s_i, s_j) is reduced by (s_h, s_j) is reframed as testing whether p_{hj} is a suffix of p_{ij} and $|p_{hj}| < |p_{ij}|$. Hence, an arc can be only reduced by an arc with smaller prefix-extension. Reading the arcs by increasing values of ℓ_p, allows to detect all transitive arcs, keeping in main memory only arcs of the string graph, since either an arc e is reduced by an arc that is already in A_s, or no arc, that is subsequently read can reduce e. Therefore our use of main memory is parsimonious, since we never remove an arc from the set A_s.

7.2.1 Algorithm Engineering

In this section we are going to describe a number of improvements that we have realized to transform the algorithms described in the previous section into the current implementation. First of all, LSG must manage reads originating from both DNA strands. To such purpose, for each read with index i, we add its reverse-complemented version as a virtual read with index $i + n$ (where n is the number of original reads). Notice that the reverse-complemented read is called virtual because we do not actually append it to the input file, but our implementation simulates the insertion of such a such virtual read $i + n$ when reading the input.

Since our implementation is disk-based, it is paramount to minimize the size of data stored on disk, as well as the number of accesses to the disk and the number of open files (otherwise some disk operations become too slow). This fact requires a careful orchestration of input and output files. More precisely, the basic arc-pairs (ϵ, α) are stored in the class of files $\mathcal{BAI}(a, \ell)$, where $a = \alpha[1]$ and $\ell = |\alpha|$, while the data related to non-basic arc-pairs (β, α) are split in two separate classes of files $\mathcal{AI}(a, \ell)$ and $\mathcal{AL}(a, \ell, \ell_p)$. The first class of files contains the information about $\beta\alpha\$$, while the second class contains the information regarding β. More in detail, the files $\mathcal{AI}(a, \ell)$ contain the records $(q(\beta\alpha\$), q(\$\alpha), \ell_p)$, where $a = \beta[1]$, $\ell = |\beta\alpha\$|$, and $\ell_p = |\beta|$, that is non-basic arc-pairs of length ℓ. The second class consists of the files $\mathcal{AL}(a, \ell, \ell_p)$ containing the records $(q(\beta), q^{rev}(\beta^{rev}))$, where $\ell = |\beta\alpha\$|$ and $\ell_p = |\beta|$, that is the label-pairs corresponding to arc-pairs of length ℓ. Each one of the above files contains essentially a sequence of disjoint string-intervals. Moreover, records in $\mathcal{BAI}(\cdot, \cdot)$, $\mathcal{AI}(\cdot, \cdot)$ and $\mathcal{AL}(\cdot, \cdot, \cdot)$ are sorted

by increasing value of the start of $q(\alpha\$)$, $q(\beta\alpha\$)$ and $q(\beta)$, respectively. The association between an arc-pair and its related label-pair is easily maintained, since they have the same rank inside the set of the arc-pairs with p-extension of length ℓ_p and the set of the label-pairs of length ℓ_p, respectively. In other words, the j-th record of $\mathcal{AI}(a, l)$, containing a given length ℓ_p, is associated to the j-th record of $\mathcal{AL}(a, \ell, \ell_p)$.

At iteration k of the labeling phase, our procedure first backward extends, with a single scan of the BWT file $\mathcal{B}$, each arc-pair (β, α) (possibly $\beta = \epsilon$) of the sorted union of the files $\mathcal{AI}(\cdot, k-1)$ and $\mathcal{BAI}(\cdot, k-1)$. At the same time, the set X of symbols, which have produced a nonempty backward extension, is saved into a temporary file $F(\ell_p)$ as a record (h, X), where $h = |X|$; this information will be used in the following to check if the input arc-pair is terminal (that is, $X = \{\$\}$) or to extend the related label-pair. Each nonempty backward a-extension $(a\beta, \alpha)$, where $a \neq \$$, is put into the file $\mathcal{AI}(a, k)$ and will be used in the next iteration of the labeling phase. Observe that two input records of $\mathcal{AI}(\cdot, k-1)$ and $\mathcal{BAI}(\cdot, k-1)$ may have the same interval $q(\beta\alpha\$)$ (or $q(\alpha\$)$ when the arc-pair is basic): our procedure is able to detect that case and process only the first one of each distinct record, by storing its multiplicity mimicking an RLE-encoding of arc-pairs. Iteration k is completed with a second scan of the file $\mathcal{B}$ in order to backward-extends the label-pairs contained in the files $\mathcal{AL}(\cdot, k-1, \cdot)$: the j-th record in $\mathcal{AL}(\cdot, k-1, \ell_p)$ will be extended by the symbols of the set X of the j-th record in the file $F(\ell_p)$. The results are written in the files $\mathcal{AL}(\cdot, k, \ell_p + 1)$. Observe that, if the input arc-pair is basic, then its label-pair is trivially $(q(a), q^{rev}(a))$ and is written into the file $\mathcal{AL}(a, k, 1)$. Notice that, if $X = \{\$\}$ (that is, the input arc-pair is terminal), then the related label-pair $(q(\beta), q^{rev}(\beta^{rev}))$ is not further extended. In that case, the set of arcs, with overlap α and outgoing from the (unique) read $s = \beta\alpha$, are produced together with the string-interval $q^{rev}(\beta^{rev})$, which is necessary to reduce the overlap graph.

A final improvement regards the reduction step, with the goal of minimizing the main memory usage. In fact, each arc (s_j, s_i) and the corresponding string-interval $q^{rev}(\beta^{rev})$ are output to the file $\mathcal{A}(i \mod z)$, for an opportune z. Since our reducibility test needs to consider only the arcs incoming into the same vertex, each file $\mathcal{A}(j)$ can be analyzed separately to compute the string graph, greatly reducing the RAM usage.

An important improvement regards the GSA. In fact, since only step (i) of the algorithm (computing the unlabeled overlap graph) uses the GSA and requires only the lengths of the suffixes, we only need to store the first field of each GSA entry, omitting the index of the read. The index of the read is needed only when a $\beta\alpha\$$-interval has a nonempty backward \$-extension (that is, we have fully identified the read $s_i = \beta\alpha$ and we require its index i). In fact, we exploit the fact that all suffixes that are equal to \$ (*i.e.*, the sentinel symbol) are at the beginning of the GSA and that there is exactly one such suffix for each input read. We reorder only those suffixes to ensure that $SA[i] = (0, i)$, where s_i is the i-th read in lexicographical order as showed in Algorithm 1 in Section 6.1. Note that Algorithm 1 only requires a single scan of the (complete) and thus can be performed with negligible RAM requirements.

In our algorithm, the backward \$-extensions are computed for a set of $\beta\alpha\$$-intervals that are ordered lexicographically. Hence the nonempty $\$\beta\alpha\$$-intervals (computed via backward \$-extensions) are ordered and can be used to identify the correct source of the edge analyzed.

Since our implementation has been designed to manage a large number of short reads, the suffix lengths are stored as an 8 bit integer (hence it is able to handle reads that are at most 255 characters long).

7.3 Results and discussion

We have implemented our representation and algorithms into an open source software (called LSG and available at `http://lsg.algolab.eu`), and we have performed an experimental comparison on a large portion of the NA12878 dataset.

In fact, since LSG is aimed at reducing memory requirements, a comparison is interesting only on large datasets. For this reason, our experiments have been performed on the human reads of the NA12878 sample of the 1000 Genomes Project matching read group “20FUK”. We preprocessed and filtered these reads according to the first five steps of the workflow used by SGA in the recent Genome Assembly Gold-standard Evaluation (GAGE) project [125] as reported at `http://gage.cbcb.umd.edu/recipes/sga.html`. The resulting filtered dataset contains approximately 875 million reads, all 101bp long.

Table 7.1: Running time (in minutes) and peak memory usage (in GBytes) of LSG and SGA to build the string graph on the NA12878 dataset.

	LSG	SGA (string graph)
Running time (min)	9,388	4,145
Peak memory usage (GB)	0.87	43

Our first analysis compares LSG and the most recent version of SGA [135], examining the running time and the memory usage to compute the string graph, hence not considering the indexing phase that both programs require (since LSG does not include an indexing phase, but relies upon BEETL [6]). The results of this analysis are summarized in Table 7.1, where we can notice that SGA uses almost 50 times the main memory of LSG, while LSG needs a running time that is 2.3x that of SGA. We point out that the string graphs computed by LSG and SGA are essentially the same, except for a very minor divergence regarding pairs of reads with multiple overlaps. More precisely, the string graph computed by LSG has $\approx 3.5\%$ more arcs than the one computed by SGA. This small difference does not (sensibly) impact on the resulting assembly since it is due to the presence of multiple edges with distinct labels between the same pairs of vertices: in that case, LSG keeps all such edges, whereas SGA keeps only the edge with the shortest label. We want to highlight that, although the two graph produced by LSG and SGA are different, both of them are *correct* from a theoretical point of view. Indeed, the standard definition proposed by Myers [97] does not consider such case and choosing only one edge between them is not required (although reasonable). Anyway, we note that we can easily remove multi arcs by a local analysis of the graph in a successive step (not implemented in LSG).

In order to assess the performances of our tool, we have also built a pipeline for genome assembly based upon LSG, using a slightly modified version of BEETL to index the input reads and the assembly phase of SGA. We denote the new pipeline as BEETL+LSG+SGA, shortly BLS. We have compared this complete pipeline with SGA [134] and Readjoiner [54], two state-of-the-art string graph-based assemblers.

Notice that the indexing phases of SGA and LSG are based on the same algorithm

(namely, BCR [6]), but use two different implementations: SGA uses Ropebwt [80], which is an in-memory implementation. On the other hand, we have selected BEETL [6] for the indexing phase, since it is (to the best of our knowledge) the only available disk-based implementation that is able to compute the BWT, and the suffix array of a large set of input reads without concatenating them. Therefore our choice of BEETL is more germane to our investigation of a memory efficient string graph construction. Additionally, BEETL is used also to compute the LCP array. Unfortunately, at the time of this investigation, BEETL does not implement the `LCPext` algorithm [6], that is an external memory computation of the LCP array, hence this part has to resort to an in-memory approach. Our modifications to BEETL 0.10.0 consists of tailoring the output to the need of LSG, so that the disk usage is minimized, without changing the algorithm. More precisely, only suffix lengths of the GSA are stored (besides the BWT and the LCP array).

Furthermore, LSG converts its internal compact representation of the string graph to the ASQG format used by SGA in order to process the resulting string graph by the subsequent assembly and scaffolding steps of the SGA workflow. We point out that the current implementation of the conversion to the ASQG format is not optimized, since the overall memory usage is dominated by the assembly phase.

Even though LSG is able to run on commodity hardware, in order to properly compare the performance of the three tools, the experiment has been performed on a server running Ubuntu Linux 12.04 equipped with eight 4-core Intel Xeon E5-4610v2 2.30GHz CPUs, 256GB of RAM, and standard mechanical hard disk drives.

For all tools, we required a minimum overlap τ of 65bp. The index structures used by LSG (*i.e.*, the BWT, the GSA, and the LCP array) has been computed by BEETL, while the index used by SGA has been obtained using `sga-index` with the `-a ropebwt` option.

We used the *running time* and *peak main memory usage* to measure the computational performance of the tools.

First we have compared in Table 7.2 the computational performances of the three phases (indexing, string graph construction and output, and assembly) of BEETL, LSG, and SGA. Since Readjoiner is a single program and its phases are not perfectly corre-

Table 7.2: Running time (in minutes) and peak memory usage (in GBytes) of each phase of BEETL, LSG, SGA on the NA12878 dataset. We report two values for the string graph output step of LSG: the first one preserves the FASTA IDs in the ASQG output, whereas the second one does not and uses uniques integer IDs.

	BEETL	LSG	SGA
Indexing	9,540		1,112
String graph construction and output		9,444	4,145
Assembly			1,637
Indexing	52		26
String graph construction		0.87	*43
String graph output		45 (1)	63

* SGA does not separate the string graph construction and output phases. The peak memory usage for the string graph construction phase has been empirically measured during the execution.

sponding to those of LSG and/or SGA, we could not include it in this comparison.

A cursory examination of Table 7.2 shows that the main bottleneck of BEETL+LSG+SGA is represented by BEETL, which uses 52GB of memory for computing the index structures needed by LSG, while the other two phases – string graph construction and output – require 1GB and 45GB, respectively. As stated previously, the memory requirements of BEETL are largely due to the fact that it does not currently implement the `LCPext` algorithm [6] to compute the LCP. Furthermore, the other phase of BEETL+LSG+SGA that requires a significant amount of memory is the string graph output phase. This is mainly due to the fact that the ASQG format represents edges of the string graph by pairs of read IDs (as indicated in the input FASTA/Q file). As a consequence, during the conversion to the ASQG format, LSG maintains in memory the full list of read IDs, which accounts for most of the memory usage. Indeed, if we replace the original read IDs with their index in the input file (thus avoiding to explicitly store the IDs), then the peak memory usage of this phase becomes less than 1GB. This phase is the most memory expensive phase for SGA as well, which faces the same problem. Nevertheless, SGA still requires a significant amount of memory during string graph construction (43GB), while the memory usage of LSG during this phase is, to any practical extent, negligible

(1GB). Furthermore, since the space complexity of LSG (in terms of main memory) is not dependent on the size of the dataset, this result suggests that, as the size of the dataset increases, then the advantage in memory usage of using LSG for string graph construction would increase even more. We stress that, even if LSG is able to compute the string graph using 43 times less memory than SGA, LSG is only 3 times slower than SGA.

Our final comparison is summarized in Table 7.3 and is dedicated to study the computational performances and the quality of the assemblies produced obtained by the BEETL+LSG+SGA, SGA, and Readjoiner pipelines. In this experiment, Readjoiner has been executed with its default options.

Overall, Readjoiner is 9.7 times faster than SGA and SGA is 3 times faster than BEETL+LSG+SGA. Indeed, the peak memory usage of SGA (63GB) is larger than that of BEETL+LSG+SGA (52GB). Notably, the peak memory usage of Readjoiner (42GB) is smaller than the one of BEETL+LSG+SGA. Readjoiner reaches its memory usage peak (42GB) during its overlap detection phase and, since it does not apparently store read IDs, its memory usage in that phase is essentially devoted to storing the sorted array of SPM-relevant suffixes [54]. Regarding the quality of the assemblies produced by the pipelines analyzed, we have not distinguished BEETL+LSG+SGA and SGA, since the string graphs produced by them are basically identical. As a consequence, we performed a single assembly for both BEETL+LSG+SGA and SGA by invoking the assembly tool of the SGA workflow. All metrics show that the contigs produced by BEETL+LSG+SGA and SGA are clearly better than those produced by Readjoiner (for instance, the N50 value for LSG-SGA assembly is almost 5 times that of Readjoiner), largely thanks to better assembly techniques currently implemented by the assembly tool of SGA. This justifies our attention on the integration with the SGA workflow.

7.4 Conclusions and possible extensions

We have proposed the first external memory algorithm, based on the notions of FM-index and of backward extension [42], for building a string graph from a set T of reads. Overall, we have shown that LSG is a memory-efficient alternative approach to SGA

Table 7.3: Comparison between BEETL+LSG+SGA (BLS), SGA, and Readjoiner pipelines on the NA12878 dataset.

	BLS	SGA	Readjoiner
Total contigs		15,322,517	22,559,637
Max contig size		77,395	20,141
N25		7,258	1,628
N50		3,450	739
N75		981	270
Total running time (min)	20,621	6,894	713
Overall peak memory usage (GB)	52	63	42

and Readjoiner for computing the string graph of large datasets. For this task, LSG achieves a significant reduction in memory usage ($\approx$ 45 times compared to both SGA and Readjoiner), in exchange of a less sharp increase of running times (3 times w.r.t. SGA and 28 times w.r.t. Readjoiner, if we exclude the assembly phase, where Readjoiner implements only basic techniques).

From the algorithmic point of view LSG exploits some characteristics of FM-indexing of reads to compute the overlaps between reads, mainly the fact that it is not required to process single reads one after the other to detect common overlaps among reads, but the FM-index itself retains all the needed information. In fact, once the FM-indexing has been built, differently from SGA, LSG does not need to process the set of reads, but simply by iterating backward extensions over the FM-index it is able to discover common overlaps. We believe that some of the techniques used in LSG could be further improved to achieve an optimal trade-off between time and space in building the string graph. Moreover, it would be useful to improve LSG algorithm implementation by allowing parallel execution of the computation (e.g. by having one process that reads and writes information from and to the disks and one process that performs the computation of the arc-pair and label-pair analyzing the BWT).

We showed in this chapter that LSG can be used in a pipeline for de novo assembly by providing the construction of a string graph in a string-graph based assembler, such as SGA. Following this research direction, we have investigated the use of LSG inside

a novel pipeline: BLS. An experimental analysis shows that BLS has lighter memory requirements than SGA, while retaining the same quality in the assembly phase. Moreover, the experiments have suggested possible improvements to the BLS pipeline, mainly consisting of improving the other two components that have been integrated with LSG: the indexing phase by implementing `LCPext`, and the assembly phase from the string-graph by storing and using the string-graph itself in a more memory efficient way (for example using the external memory again). We believe that some of the ideas of LSG can be further investigated, with the goal of obtaining a complete assembly tool that is entirely disk-based. This would greatly improve the space requirements needs for de novo assembly of huge datasets that are currently ranging in the hundreds of gigabytes of data and cannot be managed with a standard workstation.

Though we have applied LSG to genomic reads, it would be interesting to investigate developments of our approach to build string graphs for de novo assembly of RNA-seq data in absence of a reference. More precisely, FM-indexing of reads exhibits some further interesting combinatorial properties that could be used in this framework. For example, as shown in [27], compressed FM-indexing of a collection of reads (both genomic and transcriptomic reads) allows a memory efficient algorithm for detecting splice sites in RNA-seq data even in absence of a reference. On the other hand, we expect that our approach may be used to face the problem of assembling RNA-seq data and building graph models of gene structures such as for example splicing graphs [8].

Another possible direction for future research regards the design and the analysis of a new version of our algorithm that is able to take advantage of a parallel disk architecture, such as a RAID.

8 Parallel assembly algorithm

In this section we will present a second implementation of the framework we defined in Chapter 6. The main goal of this tool, called FastStringGraph (FSG), is to reduce the time required by the graph construction step of string graph based assemblers.

The main motivation behind this work is that the amount of sequencing data produced is increasing steadily year by year [105] and, hence, fast methods for assembling NGS data are required.

As stated before, this goal is almost orthogonal to the one presented in the previous chapter where we presented LSG, an external memory-based NGS assembler. Indeed, LSG aims to minimize the main memory usage and to run on simple workstation whereas FSG does not emphasize the RAM usage (although it is memory parsimonious).

In the following we will refer to this approach as being parallel. We want to highlight that we do not consider the theoretical definition of parallel algorithm [43, 52, 53] based on a parallel random access memory (pRAM) but that we use such term to refer to a method that uses threads to compute in parallel independent sets of instructions. More precisely, FSG uses Intel®'s Threading Building Blocks Library (Intel® TBB) [26] to split the work between the processors. Intel® TBB is a widely used `C++` library for shared memory parallel programming and heterogeneous computing that provides a wide range of features for parallel programming (concurrent containers, scalable memory allocator, task scheduler, and low-level synchronization primitives) that can be ported to a wide range of CPU architectures (see `https://www.threadingbuildingblocks.org/` and [115] for an in-depth discussion of such library).

FSG aims to minimize the computational time by using a parallel approach and by grouping together redundant operations and performing them only once. Indeed, an important observation is that SGA [135] computes the string graph basically performing, for each read $s = s_1, \ldots, s_m$, a query to the FM-index for each character $s_1, \ldots, s_m$,

to compute the arcs outgoing from the read. While this approach works in $O(nm)$ time (where n is the number of reads and m is the length of them), it can perform several redundant queries, most notably when the reads share common suffixes — a very common case when suffixes are short —. Our algorithm queries the FM-index in a specific order, implicitly grouping together reads by their suffixes and computing only once the shared search operations. We have implemented FSG and integrated it with the SGA assembler, by replacing in SGA the step related to the string graph construction. Our implementation follows the SGA guidelines, *i.e.*, we use the correction step of SGA before computing the overlaps without allowing mismatches (which is also SGA's default). Notice that SGA is a finely tuned implementation that has performed very nicely in the latest Assemblathon competition [16]. We have compared FSG with SGA, where we have used the latter's default parameter (that is, we compute overlaps without errors). Our experimental evaluation on a standard benchmark dataset shows that our approach is 2.3–4.8 times faster than SGA in terms of wall clock time.

Together, FSG and LSG show that the approach proposed in Chapter 6 is extremely flexible.

8.1 Definitions

We briefly recall some standard definitions that will be used in the following. Let Σ be a constant-sized alphabet of size σ and let s be a string over Σ. We denote by $s[i]$ the i-th symbol of s, by $\ell = |s|$ the length of s, and by $s[i:j]$ the substring $s[i]S[i+1]\dots s[j]$ of s. The *suffix* and *prefix* of s of length k are the substrings $s[\ell-k+1:\ell]$ (denoted by $s[\ell-k+1:]$) and $s[1:k]$ (denoted by $s[:k]$) respectively. Given two strings (s_i, s_j), we say that s_i *overlaps* s_j iff a nonempty suffix α of s_i is also a prefix of s_j, that is $s_i = \beta\alpha$ and $s_j = \alpha\beta$. In this chapter we consider a set S of n strings over Σ that are terminated by the sentinel \$, which is the smallest character. To simplify the exposition, we will assume that all input strings have exactly m characters, excluding the \$. The *overlap graph* of a set S of strings is the directed graph $G_O = (S, A)$ whose vertices are the strings in R, and each two overlapping strings $s_i = \beta\alpha$ and $s_j = \alpha\gamma$ form the arc $(s_i, s_j) \in A$ labeled by β. In this case α is called the *overlap* of the arc and β is called

the (left) *extension* of the arc. Observe that the notion of overlap graph originally given by [97] is defined by labeling with γ the arc $(s_i, s_j) \in A$.

We briefly recall that, the *Generalized Suffix Array (GSA)* [133] of S is the array SA where each element $SA[i]$ is equal to (k, j) iff the k-long suffix $s_j[|s_j| - k + 1 :]$ of the string s_j is the i-th smallest element in the lexicographic ordered set of all suffixes of the strings in S. The *Burrows-Wheeler Transform (BWT)* of S is the sequence B such that $B[i] = s_j[|s_j| - k]$, if $SA[i] = (k, j)$ and $k > 1$, or $B[i] = \$$, otherwise. Informally, $B[i]$ is the symbol that precedes the k-long suffix of a string s_j where such suffix is the i-th smallest suffix in the ordering given by SA. For any string ω, all suffixes of (the lexicographically sorted) SA whose prefix is ω appear consecutively in SA. Consequently, we define the ω-interval [6], denoted by $q(\omega)$, as the maximal interval $[b, e]$ such that $b \leq e$, $SA[b]$ and $SA[e]$ both have prefix ω. Notice that the width $e-b+1$ of the ω-interval is equal to the number of occurrences of ω in some read of S. Since the BWT B and SA are closely related, we also say that $[b, e]$ is a ω-interval on B. Given a ω-interval and a character c, the *backward c-extension* of the ω-interval is the $c\omega$-interval.

8.2 Methods

FSG's algorithm is based on two steps: in the first we compute the overlap graph whereas in the second we remove all transitive arcs. Given a string α and a set of reads R, let $R^S(\alpha)$ and $R^P(\alpha)$ be respectively the subset of R with suffix (resp. prefix) α. As usual in string graph construction algorithms, we will assume that the set R is *substring free*, *i.e.*, no string is a substring of another. A fundamental observation is that the list of all nonempty overlaps α is a compact representation of the overlap graph, since all pairs in $R^S(\alpha) \times R^P(\alpha)$ are arcs of the overlap graph. Our approach to compute all overlaps between pairs of strings is based on the notion of *potential overlap*, which is a nonempty string $\alpha^* \in \Sigma^+$, s.t. there exists at least one input string $s_i = \beta\alpha^*$ $(\beta \neq \epsilon)$ with suffix α^*, and there exists at least one input string $s_j = \gamma\alpha^*\delta$ $(\delta \neq \epsilon)$ with α^* as a substring (possibly a prefix). The first part of Algorithm 4 (lines 3–11) is devoted to the computation of potential overlaps, starting from those of length 1 and extending the potential overlaps by adding a new leading character to each one. Lemma 8.1 is a direct

consequence of the definition of potential overlap that motivates the design of the first part of our algorithm.

Lemma 8.1 *Let α be an overlap. Then all suffixes of α are potential overlaps.*

Therefore, this part of the algorithm computes potential overlaps for increasing length and, once they cannot be extended anymore, it outputs the arcs of the overlap graph.

The second part of our algorithm is devoted to the detection of all the transitive arcs. We start considering the set of all arcs sharing the same overlap and a suffix of their extensions, as stated in the following definition.

Definition 8.1 *Assume that $\beta, \alpha \in \Sigma^*, \alpha \neq \epsilon$ and $X \subseteq R^P(\alpha)$. The* arc-set $ARC(\beta, \beta\alpha, X)$ *is the set $\{(s_1, s_2) : \beta\alpha$ is a suffix of $s_1,\ s_1 \in R,$ and $s_2 \in X\}$. The strings β and α are called the* extension *and the* overlap *of the arc-set. The set X is called the* destination *set of the arc-set.*

In other words, an arc-set contains the arcs with overlap α and extension β. An arc-set is *terminal* if there exists $s \in R$ s.t. $r = \alpha\beta$, while an arc-set is *basic* if $\beta = \epsilon$ (the empty string). Since the arc-set $ARC(\beta, \beta\alpha, X)$ is uniquely determined by strings β, $\beta\alpha$, and X, the triple $(\beta, \beta\alpha, X)$ encodes the arc-set $ARC(\beta, \beta\alpha, X)$. Moreover, the arc-set $ARC(\beta, \beta\alpha, X)$ is *correct* if X includes all reads whose irreducible arcs have overlap α and extension with suffix β, that is $X \supseteq \{s_2 \in R^P(\alpha) : s_1 \in R^S(\beta\alpha)$ and (s_1, s_2) is irreducible$\}$. Observe that our algorithm computes only correct arc-sets. Moreover, terminal arc-sets only contain irreducible arcs (Lemma 8.4).

Lemma 8.2 shows the use of arc-sets to detect transitive arcs.

Lemma 8.2 *Let (s_1, s_2) be an arc with overlap α. Then (s_1, s_2) is transitive iff (i) there exist $\beta, \gamma, \delta, \eta \in \Sigma^*$, $\gamma, \eta \neq \epsilon$ such that $s_1 = \gamma\beta\alpha$, $s_2 = \alpha\delta\eta$, (ii) there exists an input read $s_3 = \beta\alpha\delta$ such that (s_3, s_2) is an irreducible arc of a nonempty arc-set $ARC(\beta, \beta\alpha\delta, X)$.*

Proof Let $s_3 = \beta\alpha\delta$ be the input string maximizing $|\delta|$ so that $s_1 = \gamma\beta\alpha$, $s_2 = \alpha\delta\eta$, for some $\beta, \gamma, \delta, \eta \in \Sigma^*$. Notice that such string s_3 exists iff (s_1, s_2) is transitive.

If no such input string s_3 exists, then the arc-set $ARC(\beta, \beta\alpha\delta, X)$, for each set X, are empty.

Algorithm 4: Compute the string graph

```
   Input   : The set R of input strings
   Output: The string graph of R, given as a list of arcs
1  Cluster ← empty list;
   // Compute all overlaps of length 1
2  Last ← {c ∈ Σ | suff(c) > 0 and substr(c) > suff(c)};
3  while Last is not empty do
4  |  New ← ∅;
5  |  foreach α* ∈ Last do
6  |  |  if pref(α*) > 0 then
7  |  |  |  Append (ϵ, α*, listpref(α*)) to Cluster;
8  |  |  for c ∈ Σ do
9  |  |  |  if suff(cα*) > 0 and substr(cα*) > suff(cα*) then
10 |  |  |  |  Add cα* to New;
11 |  Last ← New;
12 Clusters ← the stack with Cluster as its only element;
13 while Clusters is not empty do
14 |  CurrentCluster ← Pop(Clusters);
15 |  Rdc ← ∅;
16 |  Let ExtendedClusters be an array of |Σ| empty clusters;
17 |  foreach (β, βα, X) ∈ CurrentCluster do
18 |  |  if substr(βα) = pref(βα) = suff(βα) > 0 then
19 |  |  |  Output the arcs (βα, x) with label β for each x ∈ X;
20 |  |  |  Rdc ← Rdc ∪ X;
21 |  foreach (β, βα, X) ∈ CurrentCluster do
22 |  |  if X ⊈ Rdc then
23 |  |  |  for c ∈ Σ do
24 |  |  |  |  if suff(cβα) > 0 then
25 |  |  |  |  |  Append (cβ, cβα, X \ Rdc) to ExtendedClusters[c];
26 |  Push each non-empty cluster of ExtendedClusters to Clusters;
```

Assume now that such an input string s_3 exists, then the arc (s_3, s_2) reduces (s_1, s_2). First we prove that (s_3, s_2) is irreducible. Assume to the contrary that (s_3, s_2) is transitive, hence there exists an arc (s_4, s_2) whose extension is a suffix of β. Since s_4 is not a substring of s_3, this fact contradicts the assumption that s_3 maximizes $|\delta|$. Consequently (s_3, s_2) is irreducible.

Moreover, let $ARC(\beta, \beta\alpha\delta, X)$ be a correct arc-set (at least one such correct arc-set exists, when $X = R^P(\alpha\delta)$). Since (s_3, s_2) is correct, then $s_2 \in X$ hence $ARC(\beta, \beta\alpha\delta, X)$

is nonempty. □

Note that Lemma 8.2 is a reformulation of Lemma 6.5 in terms of arc-sets. A direct consequence of Lemma 8.2 is that a nonempty correct terminal arc-set $ARC(\beta, \beta\alpha\delta, X)$ implies that all arcs of the form $(\gamma\beta\alpha, \alpha\delta\eta)$, with $\gamma, \eta \neq \epsilon$ are transitive. Another consequence of Lemma 8.2 is that an irreducible arc $(\beta\alpha\delta, \alpha\delta\eta)$ with extension β and overlap $\alpha\delta$ reduces all arcs with overlap α and extension $\gamma\beta$, with $\gamma \neq \epsilon$. Lemma 8.2 is the main ingredient used in our algorithm. More precisely, it computes terminal correct arc-sets of the form $ARC(\beta, \beta\alpha\delta, X)$ for extensions β of increasing length. By Lemma 8.2, $ARC(\beta, \beta\alpha\delta, X)$ contains arcs that reduce all the arcs contained in $ARC(\beta, \beta\alpha, X')$ which have a destination in X. Since the transitivity of an arc is related to the extension β of the arc that is used to reduce it, and since our algorithm considers extensions of increasing length, a main consequence of Lemma 8.2 is that it computes terminal arc-sets that are correct, that is they contain only irreducible arcs. We will further speed up the computation by clustering together the arc-sets sharing the same extension.

Definition 8.2 *Let T be a set of arc-sets, and let β be a string. The* cluster *of β, denoted by $C(\beta)$, is the union of all arc-sets of T whose extension is β.*

We sketch Algorithm 4 which consists of two phases: the first phase to compute the overlap graph, and the second phase to remove all transitive arcs. In our description, we assume that, given a string ω, we can compute in constant time (i) the number $\mathsf{suff}(\omega)$ of input strings whose suffix is ω, (ii) the number $\mathsf{pref}(\omega)$ of input strings whose prefix is ω, (iii) the number $\mathsf{substr}(\omega)$ of occurrences of ω in the input strings. Moreover, we assume to be able to list the set $\mathsf{listpref}(\omega)$ of input strings with prefix ω in $O(|\mathsf{listpref}(\omega)|)$ time. In Sect. 8.2.1 we will describe such a data structure.

The first phase (lines 3– 11) exploits Lemma 8.1 to compute all overlaps. Potential overlaps are defined inductively. The empty string ϵ is a potential overlap of length 0; given an i-long potential overlap α^*, the $(i+1)$-long string $c\alpha^*$, for $c \in \Sigma$, is a potential overlap iff $\mathsf{suff}(c\alpha^*) > 0$ and $\mathsf{substr}(c\alpha^*) > \mathsf{suff}(c\alpha^*)$. Our algorithm uses this definition to build potential overlaps of increasing length, starting from those with length 1, *i.e.*, symbols of Σ (line 2). The lists *Last* and *New* store the potential overlaps computed at the previous and current iteration respectively. Observe that a potential overlap α^* is an

overlap iff $\mathsf{pref}(\alpha^*) > 0$. The first phase produces (line 7) the set of disjoint *basic* arc-sets $ARC(\epsilon, \alpha, R^p(\alpha))$ for each overlap α, whose union is the set of arcs of the overlap graph. Recall that $\mathsf{listpref}(\alpha)$ gives the set of reads with prefix α, which has been denoted by $R^p(\alpha)$.

The second phase (lines 13– 25) classifies the arcs of the overlap graph into reducible or irreducible by computing arc-sets of increasing extension length, starting from the basic arc-sets $ARC(\epsilon, \epsilon\alpha, R^p(\alpha))$ obtained in the previous phase.

By Lemma 8.2, we compute all correct terminal arc-sets $ARC(\beta, \beta\alpha, X)$ and remove all arcs that are reduced by $ARC(\beta, \beta\alpha, X)$. The set *Rdc* is used to store the destination set X of the computed terminal arc-sets. Notice that if $ARC(\beta, \beta\alpha, X)$ is terminal, then all of its arcs have the same origin $r = \beta\alpha$, *i.e.*, $ARC(\beta, \beta\alpha, X) = \{(s, x) : x \in X\}$. By Lemma 8.2 all arcs in the cluster $C(\beta)$ with a destination in X and with an origin different from s are transitive and can be removed, simply by removing X from all destination sets in the arc-sets of $C(\beta)$. Another application of Lemma 8.2 is that when we find a terminal arc-set all of its arcs are irreducible, *i.e.*, it is also correct. In fact, Lemma 8.2 classifies an arc as transitive according to the existence of a read $s = \beta\alpha$ with extension β. Since the algorithm considers extensions β of increasing length, all arcs whose extensions is shorter than β have been reduced in a previous step, thus all terminal arc-set of previous iterations are irreducible. More precisely, the test at line 18 is true iff the current arc-set is terminal. In that case, at line 19 all arcs of the arc-set are output as arcs of the string graph, and at line 20 the destination set X is added to the set *Rdc* that contains the destinations of $C(\beta)$ that must be removed. For each cluster $C(\beta)$, we read twice all arc-sets that are included in $C(\beta)$. The first time to determine which arc-sets are terminal and, in that case, to determine the set *Rdc* of reads that must be removed from all destinations of the arc-sets included in $C(\beta)$. The second time to compute the clusters $C(c\beta)$ that contain the nonempty arc-sets with extension $c\beta$ consisting of the arcs that we still have to check if they are transitive or not (that is the arcs with destination set $X \setminus Rdc$).

In Algorithm 4, the cluster $C(\beta)$ that is currently analyzed is stored in *CurrentCluster*, that is a list of the arc-sets included in the cluster. Moreover, the clusters that still have to be analyzed are stored in the stack *Clusters*. We use a stack to guarantee

that the clusters are analyzed in the correct order, that is the cluster $C(\beta)$ is analyzed after all clusters $C(\beta[i:])$ — $\beta[i:]$ is a generic suffix of β. We can prove that an irreducible arc (s_1, s_2) with extension β and overlap α belongs exactly to the clusters $C(\epsilon), \ldots, C(\beta[2:]), C(\beta)$. Moreover, s_2 does not belong to the set Rdc when considering $C(\epsilon), \ldots, C(\beta[2:])$, hence the arc (s_1, s_2) is correctly output when considering the cluster $C(\beta)$.

The lemmas leading to the correctness of the algorithm follow.

Lemma 8.3 *Let e_1 be an irreducible arc (s_1, s_2) with extension β and overlap α. Then e_1 belongs exactly to the $|\beta|+1$ clusters $C(\beta), C(\beta[2:]), C(\beta[3:]), \ldots, C(\epsilon)$, while s_2 does not belong to the set Rdc when currentCluster is any of $C(\beta[2:]), C(\beta[3:]), \ldots, C(\epsilon)$. Moreover, e_1 is output by the algorithm when currentCluster is $C(\beta)$.*

Proof By construction, e_1 can belong only to the clusters $C(\beta), C(\beta[2:]), C(\beta[3:]), \ldots, C(\epsilon)$.

Now we will prove that e_1 belongs to all clusters $C(\beta), C(\beta[2:]), C(\beta[3:]), \ldots, C(\epsilon)$, while s_2 does not belong to the set Rdc when currentCluster is any of $C(\beta[2:]), C(\beta[3:]), \ldots, C(\epsilon)$. Notice that $e_1 \in C(\epsilon)$. Assume to the contrary that there exists $i \geq 2$ such that $e_1 \in C(\beta[i:])$ and $s_2 \in Rdc$ when considering a cluster $C(\beta[i:])$. Since $s_2 \in Rdc$, by Lemma 8.2 there exists a nonempty terminal arc-set $ARC(\beta[i:], \beta[i:]\alpha\gamma, X)$ s.t. $s_2 = \alpha\gamma\delta$ and $s_2 \in X$. Since it is terminal and nonempty, such arc-set contains the arc $(\beta[i:]\alpha\gamma, s_2)$ with extension $\beta[i:]$. Since $\beta[i:]$ is a suffix of β the arc e_1 is transitive, which is a contradiction.

In particular, when the algorithm examines $C(\beta[2:])$, $e_1 \in C(\beta[2:])$ and $s_2 \in X \setminus Rdc$. Moreover, e_1 belongs to the arc-set $ARC(\beta, \beta\alpha, X \setminus Rdc)$ added to ExtendedClusters[$\beta[1]$] at line 25. Clearly, such arc-set is included in $C(\beta)$. When the algorithm examines the cluster $C(\beta)$, the arc-set containing e_1 satisfies the condition at line 18, hence such arc-set is output. □

Lemma 8.4 *Let $ARC(\beta, \beta\alpha, X)$ be an arc-set inserted into a cluster by Algorithm 4. Then such arc-set is correct.*

Proof Let e_1 be an irreducible arc (s_1, s_2) of $ARC(\beta, \beta\alpha, X)$, and let β_1 be respectively the extension and the overlap α of e_1. Since $e_1 \in ARC(\beta, \beta\alpha, X)$, β is a suffix of β_1,

therefore we can apply Lemma 8.3 which implies that $e_1 \in C(\beta)$. Since the only arc triple contained in $C(\beta)$ to which e_1 can belong is $ARC(\beta, \beta\alpha, X)$, then $s_2 \in X$ which completes the proof. □

Lemma 8.5 *Let e_1 be a transitive arc (s_1, s_2) with overlap α. Then the algorithm does not output e_1.*

Proof Since e_1 is transitive, by Lemma 8.2 $s_1 = \gamma\beta\alpha$, $s_2 = \alpha\delta\eta$, and there exists an input string $s_3 = \beta\alpha\delta$ such that the arc $e_2 = (s_3, s_2)$ with overlap $\alpha\delta$ is irreducible, and all correct arc-sets of the form $ARC(\beta, \beta\alpha\delta, X)$ are nonempty and terminal.

Assume to the contrary that e_1 is output by Algorithm 4, and notice that such arc can be output only when the current cluster is $C(\beta)$ and the current arc-set is $ARC(\gamma\beta, \gamma\beta\alpha, X)$ with $s_2 \in X$.

By the construction of our algorithm, since the cluster $C(\gamma\beta)$ is nonempty, also $C(\beta)$ is nonempty: let us consider the iteration when the current cluster is $C(\beta)$. By Lemma 8.4 the arc-set $ARC(\beta, \beta\alpha\delta, X_1)$ is correct, hence it contains the arc e_2. But such arc-set satisfies the condition at line 18, hence $s_2 \in Rdc$ at that iteration. Consequently, $C(\beta)$ cannot contain an arc-set with destination set including s_2. □

Theorem 8.6 is a direct consequence of Lemmas 8.3 and 8.5.

Theorem 8.6 *Given as input a set of strings R, Algorithm 4 computes exactly the arcs of the string graph.*

Computational analysis We now sketch the time complexity of Algorithm 4. The first step (lines 3 – 11) requires $\mathcal{O}(nm)$, where m and n are the length and the number of input strings, respectively. Indeed, since a potential overlap is a suffix of some input string, there are at most nm distinct suffixes. Moreover, each query $\mathsf{suff}(\cdot)$, $\mathsf{pref}(\cdot)$, $\mathsf{substr}(\cdot)$ requires $\mathcal{O}(1)$ time, thus the time complexity related to the total number of such queries is $\mathcal{O}(nm)$. Given two strings α_1 and α_2, when $|\alpha_1| = |\alpha_2|$ no input string can be in both $\mathsf{listpref}(\alpha_1)$ and $\mathsf{listpref}(\alpha_2)$. Since each overlap is at most m long, the overall time spent in the $\mathsf{listpref}(\cdot)$ queries is $\mathcal{O}(nm)$.

The second step (lines 13 – 25) requires $\mathcal{O}(|E|d(n))$ where E is the set of the edges of the string graph and $d(n)$ is the time required to compute a union-difference between

two sets of size n. Note that, since each string $\beta\alpha$ considered in the second phase is a suffix of an input string, and there are at most nm such suffixes, at most nm arc-sets are considered in the second phase. Moreover, for each cluster a set Rdc is computed. If Rdc is empty, then each arc-set of the cluster can be examined in constant time, since all unions at line 20 are trivially empty and at line 25 the set $X \setminus Rdc$ is equal to X, therefore no operation must be computed. The interesting case is when $X \neq \varnothing$ for some arc-set. In that case the union at line 20 and the difference $X \setminus Rdc$ at line 25 are computed. Let $d(n)$ be the time complexity of those two operations on n-element sets (the actual time complexity depends on the data structure used). Notice that X is not empty only if we have found an irreducible arc, that is an arc of the string graph. Overall, there can be at most $|E|$ nonempty such sets X, where E is the set of arcs of the string graph. Summing-up, the time complexity of the entire algorithm is $\mathcal{O}(nm + |E|d(n))$.

8.2.1 Data representation

Our algorithm entirely operates on the (potentially compressed) FM-index of the collection of input reads. Indeed, each processed string ω (both in the first and in the second phase) can be represented in constant space by the ω-interval $[b_\omega, e_\omega]$ on the BWT (*i.e.*, $\mathsf{q}(\omega)$), instead of using the naïve representation with $O(|\omega|)$ space. Notice that in the first phase, the i-long potential overlaps, for a given iteration, are obtained by prepending a symbol $c \in \Sigma$ to the $(i-1)$-long potential overlaps of the previous iteration (lines 8–10). In the same way the arc-sets of increasing extension length are computed in the second phase. In other words, our algorithm needs in general to obtain string $c\omega$ from string ω, and, since we represent strings as intervals on the BWT, this operation can be performed in $\mathcal{O}(1)$ time via backward c-extension of the interval $\mathsf{q}(\omega)$ [42].

Moreover, both queries $\mathsf{pref}(\omega)$ and $\mathsf{substr}(\omega)$ can be answered in $\mathcal{O}(1)$ time. In fact, given $\mathsf{q}(\omega) = [b_\omega, e_\omega]$, then $\mathsf{substr}(\omega) = e_\omega - b_\omega + 1$ and $\mathsf{pref}(\omega) = e_{\$\omega} - b_{\$\omega} + 1$ where $\mathsf{q}(\$\omega) = [b_{\$\omega}, e_{\$\omega}]$ is the result of the backward \$-extension of $\mathsf{q}(\omega)$. Similarly, it is easy to compute $\mathsf{listpref}(\omega)$ as it corresponds to the set of reads that have a suffix in the interval $\mathsf{q}(\$\omega)$ of the GSA.

The interval $\mathsf{q}(\omega\$) = [b_{\omega\$}, e_{\omega\$}]$ allows to answer to the query $\mathsf{suff}(\omega)$ which is computed as $e_{\omega\$} - b_{\omega\$} + 1$.] The interval $\mathsf{q}(\omega\$)$ is maintained along with $\mathsf{q}(\omega)$. Moreover, since

$\mathsf{q}(\omega\$)$ and $\mathsf{q}(\omega)$ share the lower extreme $b_\omega = b_{\omega\$}$ (recall that \$ is the smallest symbol), each string ω can be compactly represented by the three integers $b_\omega, e_{\omega\$}, e_\omega$. While in our algorithm a substring ω of some input read can be represented by those three integers, we exploited the following representation for greater efficiency.

In the first phase of the algorithm we mainly have to represent the set of potential overlaps. At each iteration, the potential overlaps in *Last* (*New*, resp.) have the same length, hence their corresponding intervals on the BWT are disjoint. Hence, we can store those intervals using a pair of $n(m+1)$-long bitvectors. For each potential overlap $\alpha \in \mathit{Last}$ (*New*, resp.) represented by the α-interval $[b_\alpha, e_\alpha]$, the first bitvector has 1 in position b_α and the second bitvector has 1 in positions $e_{\alpha\$}$ and e_α. Recall that we want also to maintain the interval $q(\alpha\$) = [b_\alpha, e_{\alpha\$}]$. Since $\mathsf{substr}(\alpha) > \mathsf{suff}(\alpha)$, then $e_{\alpha\$} \neq e_\alpha$ and can be stored in the same bitvector. In the second phase of the algorithm, we mainly represent clusters. A cluster groups together arc-sets whose overlaps are pairwise different or one is the prefix of the other. Thus, the corresponding intervals on the BWT are disjoint or nested. Moreover, the destination set of the *basic* arc-sets can be represented by a set of pairwise disjoint or nested intervals on the BWT (since $\mathsf{listpref}(\alpha)$ of line 7 correspond to the interval $\mathsf{q}(\$\alpha)$). Moreover, the loop at lines 13–25 preserves the following invariant: let $ARC(\beta, \beta\alpha_1, X_1)$ and $ARC(\beta, \beta\alpha_2, X_2)$ be two arc-sets of the same cluster $C(\beta)$ with α_1 prefix of α_2, then $X_2 \subseteq X_1$. Hence, each subset of arc-sets whose extensions plus overlaps share a common nonempty prefix γ is represented by means of the following three vectors: two integers vectors V_b, V_e of length $e_\gamma - b_\gamma + 1$ and a bitvector B_x of length $e_{\$\gamma} - b_{\$\gamma} + 1$, where $[b_\gamma, e_\gamma]$ is the γ-interval and $[b_{\$\gamma}, e_{\$\gamma}]$ is the $\$\gamma$-interval. More precisely, $V_b[i]$ ($V_e[i]$, resp.) is the number of arc-sets whose representation (BWT interval) of the overlap starts (ends, resp.) at $b_\gamma + i$, while $B_x[i]$ is 1 iff the read at position $b_{\$\gamma} + i$, in the lexicographic order of the GSA, belongs to the destination set of all the arc-sets.

8.3 Results and discussion

A C++ implementation of our approach, called FSG (short for Fast String Graph), has been integrated in the SGA suite and is available at `http://fsg.algolab.eu` under the

GPLv3 license. We have evaluated the performance of FSG on a standard benchmark of 875 million 101bp-long reads sequenced from the NA12878 individual of the International HapMap and 1000 genomes project and compared the running time of FSG with SGA. We have run SGA with its default parameters, that is SGA has compute exact overlaps after having corrected the input reads. Since the string graphs computed by FSG and SGA are almost the same the same (as highlighted in Chapter 7), we have not compared the entire pipeline, but only the string graph construction phase. We could not compare FSG with Fermi, since Fermi does not split its steps in a way that allows to isolate the running time of the string graph construction — most notably, it includes reads correction and scaffolding.

Especially on the DNA alphabet, short overlaps between reads may happen by chance and, hence, for genome assembly purposes, only overlaps whose length is larger than a user-defined threshold are considered. The value of the minimum overlap length threshold that empirically showed the best results in terms of genome assembly quality is around the 75% of the read length [135]. To assess how graph size affects performance, different values of minimum overlap length (called τ) between reads have been used (clearly, the lower this value, the larger the graph). The minimum overlap lengths used in this experimental assessment are 55, 65, 75, and 85, hence the chosen values test the approaches also on larger-than-normal ($\tau = 55$) and smaller-than-normal ($\tau = 85$) string graphs.

Another aspect that we have wanted to measure is the scalability of FSG. We have run the programs with 1, 4, 8, 16, and 32 threads. In all cases, we have measured the elapsed (wall-clock) time and the total CPU time (the time a CPU has been working). All experiments have been performed on an Ubuntu 14.04 server with four 8-core Intel® Xeon E5-4610v2 2.30GHz CPUs. The server has a NUMA architecture with 64GiB of RAM for each node (256GiB in total).

Table 8.1 summarizes the running times of both approaches on the different configurations of the parameters. Notice that FSG approach is from 2.3 to 4.8 times faster than SGA in terms of wall-clock time and from 1.9 to 3 times in terms of CPU time. On the other hand, FSG uses approximately 2.2 times the memory used by SGA — on the executions with at most 8 threads. We want to highlight that on a larger number

Table 8.1: Comparison of FSG and SGA, for different minimum overlap lengths and numbers of threads. The wall-clock time is the time used to compute the string graph. The CPU time is the overall execution time over all CPUs actually used.

Min. overlap	no. of threads	Wall time [min] FSG	Wall time [min] SGA	$\frac{FSG}{SGA}$	Work time [min] FSG	Work time [min] SGA	$\frac{FSG}{SGA}$
55	1	1,485	4,486	0.331	1,483	4,480	0.331
	4	474	1,961	0.242	1,828	4,673	0.391
	8	318	1,527	0.209	2,203	4,936	0.446
	16	278	1,295	0.215	3,430	5,915	0.580
	32	328	1,007	0.326	7,094	5,881	1.206
65	1	1,174	3,238	0.363	1,171	3,234	0.363
	4	416	1,165	0.358	1,606	3,392	0.473
	8	271	863	0.315	1,842	3,596	0.512
	16	255	729	0.351	3,091	4,469	0.692
	32	316	579	0.546	6,690	4,444	1.505
75	1	1,065	2,877	0.37	1,063	2,868	0.371
	4	379	915	0.415	1,473	2,903	0.507
	8	251	748	0.336	1,708	3,232	0.528
	16	246	561	0.439	2,890	3,975	0.727
	32	306	455	0.674	6,368	4,062	1.568
85	1	1,000	2,592	0.386	999	2,588	0.386
	4	360	833	0.432	1,392	2,715	0.513
	8	238	623	0.383	1,595	3,053	0.523
	16	229	502	0.457	2,686	3,653	0.735
	32	298	407	0.733	6,117	3,735	1.638

of threads, the performances of FSG slightly degrades (in particular the wall time of FSG on 32 threads is larger than that on 16 threads). We want to point out that the current implementation of FSG is almost a proof of concept and future improvements to its codebase and a better analysis of the race conditions of our tool will likely lead to better performances with a large number of threads. Furthermore, notice that also the SGA algorithm, which is (almost) embarrassingly parallel (since, in principle, each input read can be processed independently) and has a stable implementation, does not achieve a speed-up better than 6.4 with 32 threads. As such, a factor that likely contributes to a poor scaling behaviour of both FSG and SGA could be also the NUMA architecture of the server used for the experimental analysis, which makes different-unit memory accesses more expensive (in our case, the processors in each unit can manage at most 16 logical threads, and only 8 on physical cores).

FSG uses more memory than SGA since genome assemblers must correctly manage reads extracted from both strands of the genome. In our case, this fact has been addressed by adding each reverse-and-complement read to the set of strings on which the FM-index has been built, hence immediately doubling the size of the FM-index. Moreover, FSG needs some additional data structures to correctly maintain potential overlaps and arc-sets: two pairs of $n(m+1)$-long bitvectors and the combination of two (usually) small integer vectors and a bitvector of the same size. Our experimental evaluation shows that the memory required by the latter is usually negligible, hence a better implementation of the four bitvectors — for instance compressing the static ones — could decrease the memory use. Nevertheless, we want to stress that the main goal of FSG is to show that the framework proposed in Chapter 6 can improve the running time, not the memory usage (as shown in Chapter 7).

The combined analysis of the CPU time and the wall-clock time on at most 8 threads (which is the number of physical cores of each CPU on our server) suggests that FSG is more CPU efficient than SGA and is able to better distribute the workload across the threads. In our opinion, our greater efficiency is achieved by operating only on the FM-index of the input reads and by the order on which extension operations (*i.e.*, considering a new string $c\beta$ after β has been processed) are performed. These two characteristics of our algorithm allow to eliminate the redundant queries to the index which, instead, are performed by SGA. In fact, FSG considers each string that is longer than the threshold at most once, while SGA potentially reconsiders the same string once for each read in which the string occurs. Indeed, FSG uses 2.3–3 times less user time than SGA when $\tau = 55$ (hence, when such sufficiently-long substrings occur more frequently) and "only" 2–2.6 times less user time when $\tau = 85$ (hence, when such sufficiently-long substrings are more rare).

8.4 Conclusions and possible extensions

We have proposed FSG, a tool implementing a new parallel algorithm for constructing a string graph that works directly querying a FM-index representing a collection of reads, instead of processing the input reads. Our main goal is to provide a simpler and fast

algorithm to construct string graphs, so that its implementation can be easily integrated into an assembly pipeline that analyzes the paths of the string graph to produce the final assembly. Indeed, FSG could be used for related purposes, such as transcriptome assembly [8, 74], and haplotype assembly [10]. These topics are some of the research directions that we plan to investigate.

A Appendix

A.1 Additional tables

Table A1: Sequencing technologies list. Data retrieved and adapted from [105].

Platform	Instrument	Year	Reads per run	Read length (mode or average)	Bases per run (gigabases)
ABI Sanger	3730xl	2002	96	800	0.0000768
454	GS20	2005	200000	100	0.02
454	GS FLX	2007	400000	250	0.1
454	GS FLX Titanium	2009	1000000	500	0.45
454	GS FLX+	2011	1000000	700	0.7
454	GS Junior	2010	100000	400	0.04
454	GS Junior+	2014	100000	700	0.07
Illumina (Solexa)	GA	2006	28000000	25	0.7
Illumina	GA	2008	28000000	35	1
Illumina	GA II	ND	100000000	50	5
Illumina	GAIIx	2009	440000000	75	33
Illumina	GAIIx	2011	640000000	75	48
Illumina	GAIIx	2012	640000000	150	95
Illumina	HiSeq 2000	2010	2000000000	100	200
Illumina	HiSeq 2000	2011	3000000000	100	600
Illumina	HiSeq 2000/2500	2014	4000000000	125	1000
Illumina	HiSeq 2500 RR	2012	600000000	150	180
Illumina	HiSeq 2500 RR	2014	600000000	250	300
Illumina	HiSeq 4000	2015	5000000000	150	1500
Illumina	HiSeq X	2014	6000000000	150	1800
Illumina	NextSeq 500	2014	400000000	150	120
Illumina	MiSeq	2011	30000000	150	4.5
Illumina	MiSeq	2012	30000000	250	8.5
Illumina	MiSeq	2013	30000000	300	15
Illumina	MiniSeq	2016	25000000	150	7.5
SOLiD	1	2007	40000000	25	1
SOLiD	2	2008	115000000	35	4
SOLiD	3	2009	320000000	50	16
SOLiD	4	2010	2000000000	50	100
SOLiD	5500xl	2011	3000000000	60	180
SOLiD	5500xl W	2013	3000000000	75	320
IonTorrent	PGM 314 chip	2011	100000	100	0.01
IonTorrent	PGM 316 chip	2011	1000000	100	0.1
IonTorrent	PGM 318 chip	2011	5000000	100	0.5
IonTorrent	PGM 318 chip	2012	5000000	200	1
IonTorrent	PGM 318 chip V2	2013	5000000	400	2
IonTorrent	Proton PI	2012	50000000	200	10
IonTorrent	Ion S5/S5XL 530 chip	2015	20000000	400	8
IonTorrent	Ion S5/S5XL 540 chip	2015	75000000	200	15
PacBio	RS C1	2011	432000	1300	0.540
PacBio	RS C2	2012	432000	2500	1.080
PacBio	RS C2 XL	2012	432000	4300	1.858
PacBio	RS II C2 XL	2013	564000	4600	2.594
PacBio	RS II P5 C3	2014	528000	8500	4.500
PacBio	RS II P6 C4	2014	660000	13500	12.000
Oxford Nanopore	MinION Mk1	2015	2200000	9545	21
Oxford Nanopore	MinION Mk1 fast	2015	4400000	9545	42

List of Figures

2.1 Example of a Suffix Tree 19
2.2 Example of Suffix Array and BWT 20
2.3 Developments in high throughput sequencing 25
2.4 An example of a node centric de Bruijn graph 29
2.5 An example of an overlap graph 31

3.1 An example of Bloom filter. 39

4.1 Example of a de Bruijn graph update procedure 46

5.1 ABOSS representation of a `dBG` 54
5.2 A graphical example of `bi-shorter`. 58
5.3 A graphical example of `bi-longer`. 60
5.4 A graphical example of `FwdBwd`. 62

6.1 Example of RT, BWT, GSA, and LCP 72
6.2 Example of overlap between two reads 77
6.3 Example of a reducible arc of the overlap graph 78

List of Tables

7.1 Running time (in minutes) and peak memory usage (in GBytes) of LSG and SGA to build the string graph on the NA12878 dataset. 94
7.2 Running time (in minutes) and peak memory usage (in GBytes) of each phase of BEETL, LSG, SGA on the NA12878 dataset. We report two values for the string graph output step of LSG: the first one preserves the FASTA IDs in the ASQG output, whereas the second one does not and uses uniques integer IDs. 96
7.3 Comparison between BEETL+LSG+SGA (BLS), SGA, and Readjoiner pipelines on the NA12878 dataset. 98

8.1 Comparison of FSG and SGA, for different minimum overlap lengths and numbers of threads. The wall-clock time is the time used to compute the string graph. The CPU time is the overall execution time over all CPUs actually used. 113

A1 Sequencing technologies list 117

Bibliography

[1] M. Abouelhoda, S. Kurtz, and E. Ohlebusch, "Replacing suffix trees with enhanced suffix arrays," *J. of Discrete Algorithms*, vol. 2, no. 1, pp. 53–86, 2004.

[2] A. Amir, T. M. Chan, M. Lewenstein, and N. Lewenstein, "On hardness of jumbled indexing," in *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part I*, ser. Lecture Notes in Computer Science, J. Esparza, P. Fraigniaud, T. Husfeldt, and E. Koutsoupias, Eds., vol. 8572. Springer, 2014, pp. 114–125.

[3] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi, *Complexity and approximation: Combinatorial optimization problems and their approximability properties.* Springer Science & Business Media, 2012.

[4] A. Bankevich, S. Nurk, D. Antipov, A. A. Gurevich, M. Dvorkin, A. S. Kulikov, V. M. Lesin, S. I. Nikolenko, S. Pham, A. D. Prjibelski *et al.*, "SPAdes: A new genome assembly algorithm and its applications to single-cell sequencing," *Journal of Computational Biology*, vol. 19, no. 5, pp. 455–477, 2012.

[5] M. Bauer, A. Cox, and G. Rosone, "Lightweight BWT construction for very large string collections," in *Combinatorial Pattern Matching*, ser. LNCS, vol. 6661. Springer, 2011, pp. 219–231.

[6] ——, "Lightweight algorithms for constructing and inverting the BWT of string collections," *Theoretical Computer Science*, vol. 483, pp. 134–148, 2013.

[7] I. Ben-Bassat and B. Chor, "String graph construction using incremental hashing," *Bioinformatics*, vol. 30, no. 24, pp. 3515–3523, 2014.

[8] S. Beretta, P. Bonizzoni, G. Della Vedova, Y. Pirola, and R. Rizzi, "Modeling alternative splicing variants from RNA-Seq data with isoform graphs," *J. of Computational Biology*, vol. 16, no. 1, pp. 16–40, 2014.

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

[10] P. Bonizzoni, G. Della Vedova, R. Dondi, and J. Li, "The haplotyping problem: An overview of computational models and solutions," *Journal of Computer Science and Technology*, vol. 18, no. 6, pp. 675–688, 2003.

[11] P. Bonizzoni, G. Della Vedova, Y. Pirola, M. Previtali, and R. Rizzi, "Constructing string graphs in external memory," in *Algorithms in Bioinformatics - 14th International Workshop, WABI 2014, Wroclaw, Poland, September 8-10, 2014. Proceedings*, ser. Lecture Notes in Computer Science, D. G. Brown and B. Morgenstern, Eds., vol. 8701. Springer, 2014, pp. 311–325.

[12] ——, "FSG: fast string graph construction for de novo assembly of reads data," in *Bioinformatics Research and Applications - 12th International Symposium, ISBRA 2016, Minsk, Belarus, June 5-8, 2016, Proceedings*, ser. Lecture Notes in Computer Science, A. G. Bourgeois, P. Skums, X. Wan, and A. Zelikovsky, Eds., vol. 9683. Springer, 2016, pp. 27–39.

[13] C. Boucher, A. Bowe, T. Gagie, S. J. Puglisi, and K. Sadakane, "Variable-order de bruijn graphs," in *2015 Data Compression Conference, DCC 2015, Snowbird, UT, USA, April 7-9, 2015.* IEEE, 2015, pp. 383–392.

[14] A. Bowe, T. Onodera, K. Sadakane, and T. Shibuya, "Succinct de bruijn graphs," in *Algorithms in Bioinformatics - 12th International Workshop, WABI 2012, Ljubljana, Slovenia, September 10-12, 2012. Proceedings*, ser. Lecture Notes in Computer Science, B. J. Raphael and J. Tang, Eds., vol. 7534. Springer, 2012, pp. 225–235.

[15] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Commun. ACM*, vol. 20, no. 10, pp. 762–772, 1977.

[16] K. R. Bradnam, J. N. Fass, A. Alexandrov, P. Baranay, M. Bechner, I. Birol, S. Boisvert, J. A. Chapman, G. Chapuis, R. Chikhi *et al.*, "Assemblathon 2: evaluating de novo methods of genome assembly in three vertebrate species," *GigaScience*, vol. 2, no. 1, pp. 1–31, 2013.

[17] D. G. Brown and B. Morgenstern, Eds., *Algorithms in Bioinformatics - 14th International Workshop, WABI 2014, Wroclaw, Poland, September 8-10, 2014. Proceedings*, ser. Lecture Notes in Computer Science, vol. 8701. Springer, 2014.

[18] P. Burcsi, F. Cicalese, G. Fici, and Z. Lipták, "Algorithms for jumbled pattern matching in strings," *Int. J. Found. Comput. Sci.*, vol. 23, no. 2, pp. 357–374, 2012.

[19] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," Digital Systems Research Center, Tech. Rep., 1994.

[20] B. Cazaux, G. Sacomoto, and E. Rivals, "Superstring graph: A new approach for genome assembly," in *Algorithmic Aspects in Information and Management - 11th International Conference, AAIM 2016, Bergamo, Italy, July 18-20, 2016,*

Proceedings, ser. Lecture Notes in Computer Science, R. Dondi, G. Fertin, and G. Mauri, Eds., vol. 9778. Springer, 2016, pp. 39–52.

[21] R. Chikhi and P. Medvedev, "Informed and automated k-mer size selection for genome assembly," *Bioinformatics*, vol. 30, no. 1, pp. 31–37, 2014.

[22] R. Chikhi and G. Rizk, "Space-efficient and exact de bruijn graph representation based on a bloom filter," in *Algorithms in Bioinformatics - 12th International Workshop, WABI 2012, Ljubljana, Slovenia, September 10-12, 2012. Proceedings*, ser. Lecture Notes in Computer Science, B. J. Raphael and J. Tang, Eds., vol. 7534. Springer, 2012, pp. 236–248.

[23] ——, "Space-efficient and exact de bruijn graph representation based on a bloom filter," *Algorithms for Molecular Biology*, vol. 8, p. 22, 2013.

[24] T. C. Conway and A. J. Bromage, "Succinct data structures for assembling large genomes," *Bioinformatics*, vol. 27, no. 4, pp. 479–486, 2011.

[25] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms (3. ed.).* MIT Press, 2009.

[26] I. Corporation, "Intel® threading building blocks," https://www.threadingbuildingblocks.org/, accessed: 2017-01-07.

[27] A. Cox, T. Jakobi, G. Rosone, and O. Schulz-Trieglaff, "Comparing DNA sequence collections by direct comparison of compressed text indexes," in *Algorithms in Bioinformatics*, ser. LNCS. Berlin, Germany: Springer, 2012, vol. 7534, pp. 214–224.

[28] R. D'Amore, U. Z. Ijaz, M. Schirmer, J. G. Kenny, R. Gregory, A. C. Darby, M. Shakya, M. Podar, C. Quince, and N. Hall, "A comprehensive benchmarking study of protocols and sequencing platforms for 16s rrna community profiling," *BMC Genomics*, vol. 17, no. 1, p. 55, 2016.

[29] S. Dasgupta, C. H. Papadimitriou, and U. V. Vazirani, *Algorithms.* McGraw-Hill, 2008.

[30] N. G. De Bruijn, "A combinatorial problem," *Proceedings of the Section of Sciences of the Koninklijke Nederlandse Akademie van Wetenschappen te Amsterdam*, vol. 49, no. 7, pp. 758–764, 1946.

[31] R. Diestel, *Graph Theory*, 3rd ed., ser. Graduate Texts in Mathematics. Springer-Verlag, Heidelberg, 2005.

[32] D. A. Durai and M. H. Schulz, "Informed $k$mer selection for *de novo* transcriptome assembly," *Bioinformatics*, vol. 32, no. 11, pp. 1670–1677, 2016.

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

[34] J. Ebert, "Computing eulerian trails," *Inf. Process. Lett.*, vol. 28, no. 2, pp. 93–97, 1988.

[35] M. Eisenstein, "Oxford nanopore announcement sets sequencing sector abuzz," *Nature biotechnology*, vol. 30, no. 4, pp. 295–296, 2012.

[36] P. Elias, "Efficient storage and retrieval by content and address of static files," *J. ACM*, vol. 21, no. 2, pp. 246–260, 1974.

[37] S. Even and G. Even, *Graph Algorithms, Second Edition.* Cambridge University Press, 2012.

[38] M. Farach, "Optimal suffix tree construction with large alphabets," in *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997.* IEEE Computer Society, 1997, pp. 137–143.

[39] P. Ferragina, T. Gagie, and G. Manzini, "Lightweight data indexing and compression in external memory," *Algorithmica*, vol. 63, no. 3, pp. 707–730, 2012.

[40] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan, "Compressing and indexing labeled trees, with applications," *J. ACM*, vol. 57, no. 1, 2009.

[41] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in *41st Annual Symposium on Foundations of Computer Science, FOCS 2000, 12-14 November 2000, Redondo Beach, California, USA*. IEEE Computer Society, 2000, pp. 390–398.

[42] ——, "Indexing compressed text," *J. of the ACM*, vol. 52, no. 4, pp. 552–581, 2005.

[43] S. Fortune and J. Wyllie, "Parallelism in random access machines," in *Proceedings of the 10th Annual ACM Symposium on Theory of Computing, May 1-3, 1978, San Diego, California, USA*, R. J. Lipton, W. A. Burkhard, W. J. Savitch, E. P. Friedman, and A. V. Aho, Eds. ACM, 1978, pp. 114–118.

[44] E. Fredkin, "Trie memory," *Communications of the ACM*, vol. 3, no. 9, pp. 490–499, 1960.

[45] T. Gagie, D. Hermelin, G. M. Landau, and O. Weimann, "Binary jumbled pattern matching on trees and tree-like structures," *Algorithmica*, vol. 73, no. 3, pp. 571–588, 2015.

[46] T. Gagie, G. Manzini, and D. Valenzuela, "Compressed spaced suffix arrays," in *Proceedings of the 2nd International Conference on Algorithms for Big Data , Palermo, Italy, April 07-09, 2014.*, ser. CEUR Workshop Proceedings, C. S. Iliopoulos and A. Langiu, Eds., vol. 1146. CEUR-WS.org, 2014, pp. 37–45.

[47] J. E. Gallo, J. F. Muñoz, E. Misas, J. G. McEwen, and O. K. Clay, "The complex task of choosing a de novo assembly: Lessons from fungal genomes," *Computational Biology and Chemistry*, vol. 53, pp. 97–107, 2014.

[48] V. Genetics, "Veritas genetics breaks $1,000 whole genome barrier – press release," https://www.veritasgenetics.com/documents/VG-PGP-Announcement-Final.pdf, accessed: 2017-01-02.

[49] R. Giegerich, S. Kurtz, and J. Stoye, "Efficient implementation of lazy suffix trees," in *Algorithm Engineering, 3rd International Workshop, WAE '99, London, UK, July 19-21, 1999, Proceedings*, ser. Lecture Notes in Computer Science, J. S. Vitter and C. D. Zaroliagis, Eds., vol. 1668. Springer, 1999, pp. 30–42.

[50] ——, "Efficient implementation of lazy suffix trees," *Softw., Pract. Exper.*, vol. 33, no. 11, pp. 1035–1049, 2003.

[51] T. C. Glenn, "Field guide to next-generation dna sequencers," *Molecular Ecology Resources*, vol. 11, no. 5, pp. 759–769, 2011.

[52] L. M. Goldschlager, "A unified approach to models of synchronous parallel machines," in *Proceedings of the 10th Annual ACM Symposium on Theory of Computing, May 1-3, 1978, San Diego, California, USA*, R. J. Lipton, W. A. Burkhard, W. J. Savitch, E. P. Friedman, and A. V. Aho, Eds. ACM, 1978, pp. 89–94.

[53] ——, "A universal interconnection pattern for parallel computers," *J. ACM*, vol. 29, no. 4, pp. 1073–1086, 1982.

[54] G. Gonnella and S. Kurtz, "Readjoiner: a fast and memory efficient string graph-based sequence assembler," *BMC Bioinformatics*, vol. 13, no. 1, p. 82, 2012.

[55] I. J. Good, "Normal recurring decimals," *Journal of the London Mathematical Society*, vol. 1, no. 3, pp. 167–169, 1946.

[56] S. Goodwin, J. D. McPherson, and W. R. McCombie, "Coming of age: ten years of next-generation sequencing technologies," *Nature Reviews Genetics*, vol. 17, no. 6, pp. 333–351, 2016.

[57] M. G. Grabherr, B. J. Haas, M. Yassour, J. Z. Levin, D. A. Thompson, I. Amit, X. Adiconis, L. Fan, R. Raychowdhury, Q. Zeng *et al.*, "Full-length transcriptome assembly from rna-seq data without a reference genome," *Nature biotechnology*, vol. 29, no. 7, pp. 644–652, 2011.

[58] R. Grossi, A. Gupta, and J. S. Vitter, "High-order entropy-compressed text indexes," in *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA.* ACM/SIAM, 2003, pp. 841–850.

[59] R. Grossi and J. S. Vitter, "Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract)," in *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, May 21-23, 2000, Portland, OR, USA*, F. F. Yao and E. M. Luks, Eds. ACM, 2000, pp. 397–406.

[60] ——, "Compressed suffix arrays and suffix trees with applications to text indexing and string matching," *SIAM J. Comput.*, vol. 35, no. 2, pp. 378–407, 2005.

[61] D. Hernandez, P. François, L. Farinelli, M. Østerås, and J. Schrenzel, "De novo bacterial genome sequencing: millions of very short reads assembled on a desktop computer," *Genome research*, vol. 18, no. 5, pp. 802–809, 2008.

[62] Y. Huang and C. Liao, "Integration of string and de bruijn graphs for genome assembly," *Bioinformatics*, vol. 32, no. 9, pp. 1301–1307, 2016.

[63] M. Huerta, G. Downing, F. Haseltine, B. Seto, and Y. Liu, "Nih working definition of bioinformatics and computational biology," *US National Institute of Health*, 2000.

[64] D. A. Huffman *et al.*, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[65] R. M. Idury and M. S. Waterman, "A new algorithm for DNA sequence assembly," *Journal of Computational Biology*, vol. 2, no. 2, pp. 291–306, 1995.

[66] N. H. G. R. Institute, "The cos of sequencing a human genome," https://www.genome.gov/sequencingcosts/, accessed: 2017-01-02.

[67] Z. Iqbal, M. Caccamo, I. Turner, P. Flicek, and G. McVean, "De novo assembly and genotyping of variants using colored de bruijn graphs," *Nature genetics*, vol. 44, no. 2, pp. 226–232, 2012.

[68] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM Journal of Research and Development*, vol. 31, no. 2, pp. 249–260, 1987.

[69] J. D. Kececioglu and E. W. Myers, "Combinatiorial algorithms for DNA sequence assembly," *Algorithmica*, vol. 13, no. 1/2, pp. 7–51, 1995.

[70] D. K. Kim, J. S. Sim, H. Park, and K. Park, "Linear-time construction of suffix arrays," in *Combinatorial Pattern Matching, 14th Annual Symposium, CPM 2003, Morelia, Michocán, Mexico, June 25-27, 2003, Proceedings*, ser. Lecture Notes in Computer Science, R. A. Baeza-Yates, E. Chávez, and M. Crochemore, Eds., vol. 2676. Springer, 2003, pp. 186–199.

[71] M. Kircher and J. Kelso, "High-throughput dna sequencing–concepts and limitations," *Bioessays*, vol. 32, no. 6, pp. 524–536, 2010.

[72] D. E. Knuth, J. H. M. Jr., and V. R. Pratt, "Fast pattern matching in strings," *SIAM J. Comput.*, vol. 6, no. 2, pp. 323–350, 1977.

[73] T. Kociumaka, J. Radoszewski, and W. Rytter, "Efficient indexes for jumbled pattern matching with constant-sized alphabet," in *Algorithms - ESA 2013 - 21st Annual European Symposium, Sophia Antipolis, France, September 2-4, 2013. Proceedings*, ser. Lecture Notes in Computer Science, H. L. Bodlaender and G. F. Italiano, Eds., vol. 8125. Springer, 2013, pp. 625–636.

[74] V. Lacroix, M. Sammeth, R. Guigo, and A. Bergeron, "Exact transcriptome reconstruction from short sequence reads," in *Algorithms in Bioinformatics*, ser. LNCS, vol. 5251. Springer Berlin Heidelberg, 2008, pp. 50–63.

[75] B. Lai, R. Ding, Y. Li, L. Duan, and H. Zhu, "A *de novo* metagenomic assembly program for shotgun DNA reads," *Bioinformatics*, vol. 28, no. 11, pp. 1455–1462, 2012.

[76] H. Y. Lam, M. J. Clark, R. Chen, R. Chen, G. Natsoulis, M. O'Huallachain, F. E. Dewey, L. Habegger, E. A. Ashley, M. B. Gerstein *et al.*, "Performance comparison of whole-genome sequencing platforms," *Nature biotechnology*, vol. 30, no. 1, pp. 78–82, 2012.

[77] T. Lam, R. Li, A. Tam, S. Wong, E. Wu, and S. Yiu, "High throughput short read alignment via bi-directional BWT," in *Bioinformatics and Biomedicine (BIBM '09)*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 31–36.

[78] E. S. Lander, L. M. Linton, B. Birren, C. Nusbaum, M. C. Zody, J. Baldwin, K. Devon, K. Dewar, M. Doyle, W. FitzHugh *et al.*, "Initial sequencing and analysis of the human genome," *Nature*, vol. 409, no. 6822, pp. 860–921, 2001.

[79] H. Li, "Exploring single-sample SNP and INDEL calling with whole-genome de novo assembly," *Bioinformatics*, vol. 28, no. 14, pp. 1838–1844, Jul. 2012.

[80] ——, "Fast construction of FM-index for long sequence reads," *Bioinformatics*, vol. 30, no. 22, pp. 3274–3275, 2014.

[81] Y. Lin and P. A. Pevzner, "Manifold de bruijn graphs," in *Algorithms in Bioinformatics - 14th International Workshop, WABI 2014, Wroclaw, Poland, September 8-10, 2014. Proceedings*, ser. Lecture Notes in Computer Science, D. G. Brown and B. Morgenstern, Eds., vol. 8701. Springer, 2014, pp. 296–310.

[82] R. J. Lipton, W. A. Burkhard, W. J. Savitch, E. P. Friedman, and A. V. Aho, Eds., *Proceedings of the 10th Annual ACM Symposium on Theory of Computing, May 1-3, 1978, San Diego, California, USA*. ACM, 1978.

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

[84] Y. Liu, T. Hankeln, and B. Schmidt, "Parallel and space-efficient construction of burrows-wheeler transform and suffix array for big genome data," *IEEE/ACM Trans. Comput. Biology Bioinform.*, vol. 13, no. 3, pp. 592–598, 2016.

[85] N. J. Loman, C. Constantinidou, J. Z. Chan, M. Halachev, M. Sergeant, C. W. Penn, E. R. Robinson, and M. J. Pallen, "High-throughput bacterial genome sequencing: an embarrassment of choice, a world of opportunity," *Nature Reviews Microbiology*, vol. 10, no. 9, pp. 599–606, 2012.

[86] R. Luo, B. Liu, Y. Xie, Z. Li, W. Huang, J. Yuan, G. He, Y. Chen, Q. Pan, Y. Liu *et al.*, "Soapdenovo2: an empirically improved memory-efficient short-read de novo assembler," *GigaScience*, vol. 1, no. 1, p. 1, 2012.

[87] M.-A. Madoui, S. Engelen, C. Cruaud, C. Belser, L. Bertrand, A. Alberti, A. Lemainque, P. Wincker, and J.-M. Aury, "Genome assembly using nanopore-guided long and error-free dna reads," *BMC genomics*, vol. 16, no. 1, p. 1, 2015.

[88] U. Manber and E. W. Myers, "Suffix arrays: A new method for on-line string searches," in *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '90. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1990, pp. 319–327.

[89] G. Manzini, "XBWT tricks," in *String Processing and Information Retrieval - 23rd International Symposium, SPIRE 2016, Beppu, Japan, October 18-20, 2016, Proceedings*, ser. Lecture Notes in Computer Science, S. Inenaga, K. Sadakane, and T. Sakai, Eds., vol. 9954, 2016, pp. 80–92.

[90] L. Mariot, A. Leporati, A. Dennunzio, and E. Formenti, "Computing the periods of preimages in surjective cellular automata," *Natural Computing*, pp. 1–15, 2016.

[91] E. M. McCreight, "A space-economical suffix tree construction algorithm," *J. ACM*, vol. 23, no. 2, pp. 262–272, 1976.

[92] P. Melsted and B. V. Halldórsson, “Kmerstream: streaming algorithms for k-mer abundance estimation,” *Bioinformatics*, vol. 30, no. 24, pp. 3541–3547, 2014.

[93] A. S. Mikheyev and M. M. Tin, “A first look at the oxford nanopore minion sequencer,” *Molecular ecology resources*, vol. 14, no. 6, pp. 1097–1102, 2014.

[94] G. Moore, “Cramming more components onto integrated circuits,” 1965.

[95] E. W. Myers, “Whole-genome DNA sequencing,” *Computing in Science and Engineering*, vol. 1, no. 3, pp. 33–43, 1999.

[96] ——, “The whole genome assembly of drosophila,” in *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms, January 9-11, 2000, San Francisco, CA, USA.*, D. B. Shmoys, Ed. ACM/SIAM, 2000, p. 753.

[97] ——, “The fragment assembly string graph,” *Bioinformatics*, vol. 21, no. suppl. 2, pp. ii79–ii85, 2005.

[98] ——, “A history of DNA sequence assembly,” *it - Information Technology*, vol. 58, no. 3, pp. 126–132, 2016.

[99] E. W. Myers, G. G. Sutton, A. L. Delcher, I. M. Dew, D. P. Fasulo, M. J. Flanigan, S. A. Kravitz, C. M. Mobarry, K. H. Reinert, K. A. Remington *et al.*, “A whole-genome assembly of drosophila,” *Science*, vol. 287, no. 5461, pp. 2196–2204, 2000.

[100] T. Namiki, T. Hachiya, H. Tanaka, and Y. Sakakibara, “Metavelvet: an extension of velvet assembler to de novo metagenome assembly from short sequence reads,” *Nucleic acids research*, vol. 40, no. 20, pp. e155–e155, 2012.

[101] N. H. G. R. I. National Institutes of Health, “Talking glossary of genetic terms,” https://www.genome.gov/glossary/, accessed: 2017-01-02.

[102] G. Navarro, “Wavelet trees for all,” *J. Discrete Algorithms*, vol. 25, pp. 2–20, 2014.

[103] G. Navarro and V. Mäkinen, “Compressed full-text indexes,” *ACM Comput. Surv.*, vol. 39, no. 1, Apr. 2007.

[104] G. Navarro and E. Providel, “Fast, small, simple rank/select on bitmaps,” in *Experimental Algorithms - 11th International Symposium, SEA 2012, Bordeaux, France, June 7-9, 2012. Proceedings*, ser. Lecture Notes in Computer Science, R. Klasing, Ed., vol. 7276. Springer, 2012, pp. 295–306.

[105] L. Nedberbragt, “Developments in high throughput sequencing – july 2016 edition,” https://flxlexblog.wordpress.com/2016/07/08/developments-in-high-throughput-sequencing-july-2016-edition/, accessed: 2016-12-21.

[106] U. D. of Energy, "Human genome project information archive," http://www.ornl.gov/hgmis, accessed: 2017-01-08.

[107] D. Okanohara and K. Sadakane, "Practical entropy-compressed rank/select dictionary," in *Proceedings of the Nine Workshop on Algorithm Engineering and Experiments, ALENEX 2007, New Orleans, Louisiana, USA, January 6, 2007*. SIAM, 2007.

[108] J. Pell, A. Hintze, R. Canino-Koning, A. Howe, J. M. Tiedje, and C. T. Brown, "Scaling metagenome sequence assembly with probabilistic de bruijn graphs," *Proceedings of the National Academy of Sciences*, vol. 109, no. 33, pp. 13 272–13 277, 2012.

[109] Y. Peng, H. C. M. Leung, S. Yiu, and F. Y. L. Chin, "IDBA - A practical iterative de bruijn graph de novo assembler," in *Research in Computational Molecular Biology, 14th Annual International Conference, RECOMB 2010, Lisbon, Portugal, April 25-28, 2010. Proceedings*, ser. Lecture Notes in Computer Science, B. Berger, Ed., vol. 6044. Springer, 2010, pp. 426–440.

[110] Y. Peng, H. C. Leung, S.-M. Yiu, and F. Chin, "IDBA-UD: a de novo assembler for single-cell and metagenomic sequencing data with highly uneven depth," *Bioinformatics*, vol. 28, no. 11, pp. 1420–1428, 2012.

[111] P. A. Pevzner, H. Tang, and M. S. Waterman, "An eulerian path approach to dna fragment assembly," *Proceedings of the National Academy of Sciences*, vol. 98, no. 17, pp. 9748–9753, 2001.

[112] A. D. Prjibelski, I. Vasilinetc, A. Bankevich, A. Gurevich, T. Krivosheeva, S. Nurk, S. Pham, A. Korobeynikov, A. Lapidus, and P. A. Pevzner, "Exspander: a universal repeat resolver for dna fragment assembly," *Bioinformatics*, vol. 30, no. 12, pp. i293–i301, 2014.

[113] J. Qin, R. Li, J. Raes, M. Arumugam, K. S. Burgdorf, C. Manichanh, T. Nielsen, N. Pons, F. Levenez, T. Yamada *et al.*, "A human gut microbial gene catalogue established by metagenomic sequencing," *Nature*, vol. 464, no. 7285, pp. 59–65, 2010.

[114] B. J. Raphael and J. Tang, Eds., *Algorithms in Bioinformatics - 12th International Workshop, WABI 2012, Ljubljana, Slovenia, September 10-12, 2012. Proceedings*, ser. Lecture Notes in Computer Science, vol. 7534. Springer, 2012.

[115] J. Reinders, *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007.

[116] A. Rhoads and K. Au, "Pacbio sequencing and its applications," *Genomics, Proteomics & Bioinformatics*, vol. 13, no. 5, pp. 278–289, 2015.

[117] M. Roberts, B. R. Hunt, J. A. Yorke, R. A. Bolanos, and A. L. Delcher, "A preprocessor for shotgun assembly of large genomes," *Journal of Computational Biology*, vol. 11, no. 4, pp. 734–752, 2004.

[118] G. Robertson, J. Schein, R. Chiu, R. Corbett, M. Field, S. D. Jackman, K. Mungall, S. Lee, H. M. Okada, J. Q. Qian *et al.*, "De novo assembly and analysis of rna-seq data," *Nature methods*, vol. 7, no. 11, pp. 909–912, 2010.

[119] B. Y. Ryabko, "Data compression by means of a "book stack"," *Problemy Peredachi Informatsii*, vol. 16, no. 4, pp. 16–21, 1980.

[120] K.-J. Räihä and E. Ukkonen, "The shortest common supersequence problem over binary alphabet is np-complete," *Theoretical Computer Science*, vol. 16, no. 2, pp. 187 – 198, 1981.

[121] K. Sadakane, "Compressed text databases with efficient query algorithms based on the compressed suffix array," in *Algorithms and Computation, 11th International Conference, ISAAC 2000, Taipei, Taiwan, December 18-20, 2000, Proceedings*, ser. Lecture Notes in Computer Science, D. T. Lee and S. Teng, Eds., vol. 1969. Springer, 2000, pp. 410–421.

[122] K. Salikhov, G. Sacomoto, and G. Kucherov, "Using cascading bloom filters to improve the memory usage for de brujin graphs," in *WABI*, 2013, pp. 364–376.

[123] L. Salmela and E. Rivals, "Lordec: accurate and efficient long read error correction," *Bioinformatics*, vol. 30, no. 24, pp. 3506–3514, 2014.

[124] M. Salson, T. Lecroq, M. Léonard, and L. Mouchard, "Dynamic extended suffix arrays," *J. Discrete Algorithms*, vol. 8, no. 2, pp. 241–257, 2010.

[125] S. L. Salzberg *et al.*, "GAGE: A critical evaluation of genome assemblies and assembly algorithms," *Genome research*, vol. 22, no. 3, pp. 557–567, 2012.

[126] F. Sanger, S. Nicklen, and A. R. Coulson, "Dna sequencing with chain-terminating inhibitors," *Proceedings of the National Academy of Sciences*, vol. 74, no. 12, pp. 5463–5467, 1977.

[127] K. Sato, Y. Sakakibara *et al.*, "Metavelvet-sl: an extension of the velvet assembler to a de novo metagenomic assembler utilizing supervised learning," *DNA research*, vol. 22, no. 1, pp. 69–77, 2015.

[128] M. H. Schulz, D. Weese, M. Holtgrewe, V. Dimitrova, S. Niu, K. Reinert, and H. Richard, “Fiona: a parallel and automatic strategy for read error correction,” *Bioinformatics*, vol. 30, no. 17, pp. 356–363, 2014.

[129] M. H. Schulz, D. R. Zerbino, M. Vingron, and E. Birney, “*Oases:* robust *de novo* rna-seq assembly across the dynamic range of expression levels,” *Bioinformatics*, vol. 28, no. 8, pp. 1086–1092, 2012.

[130] J. Seward, “bzip2 and libbzip2, version 1.0.5 – a program and library for data compression,” http://bzip.org/1.0.5/bzip2-manual-1.0.5.html, accessed: 2017-01-06.

[131] C. E. Shannon, “A mathematical theory of communication,” *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, July 1948.

[132] ——, “A mathematical theory of communication,” *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 5, no. 1, pp. 3–55, 2001.

[133] F. Shi, “Suffix arrays for multiple strings: A method for on-line multiple string searches,” in *Concurrency and Parallelism, Programming, Networking, and Security*, ser. LNCS, vol. 1179. Springer Berlin Heidelberg, 1996, pp. 11–22.

[134] J. Simpson and R. Durbin, “Efficient construction of an assembly string graph using the FM-index,” *Bioinformatics*, vol. 26, no. 12, pp. i367–i373, 2010.

[135] ——, “Efficient de novo assembly of large genomes using compressed data structures,” *Genome Research*, vol. 22, pp. 549–556, 2012.

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

[137] S. Skiena, *The Algorithm Design Manual (2. ed.).* Springer, 2008.

[138] K. Sutner, “De bruijn graphs and linear cellular automata,” *Complex Systems*, vol. 5, no. 1, pp. 19–30, 1991.

[139] C. Tech Correspondence, “Technical correspondence,” *Commun. ACM*, vol. 30, no. 9, pp. 792–796, Sep. 1987.

[140] E. Ukkonen, “On-line construction of suffix trees,” *Algorithmica*, vol. 14, no. 3, pp. 249–260, 1995.

[141] J. C. Venter, M. D. Adams, E. W. Myers, P. W. Li, R. J. Mural, G. G. Sutton, H. O. Smith, M. Yandell, C. A. Evans, R. A. Holt *et al.*, “The sequence of the human genome,” *science*, vol. 291, no. 5507, pp. 1304–1351, 2001.

[142] P. Weiner, "Linear pattern matching algorithms," in *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973.* IEEE Computer Society, 1973, pp. 1–11.

[143] Y. Xie, G. Wu, J. Tang, R. Luo, J. Patterson, S. Liu, W. Huang, G. He, S. Gu, S. Li *et al.*, "Soapdenovo-trans: de novo transcriptome assembly with short rna-seq reads," *Bioinformatics*, vol. 30, no. 12, pp. 1660–1666, 2014.

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

[145] A. V. Zimin, G. Marçais, D. Puiu, M. Roberts, S. L. Salzberg, and J. A. Yorke, "The masurca genome assembler," *Bioinformatics*, vol. 29, no. 21, pp. 2669–2677, 2013.