Università degli Studi di Udine
Dipartimento di Matematica e Informatica
Dottorato di Ricerca in Informatica

Ph.D. Thesis

Alignment and reconciliation strategies for large-scale *de novo* assembly

Candidate
Riccardo Vicedomini

Supervisor
Prof. Alberto Policriti

March 2016

Institute Contacts
Dipartimento di Matematica e Informatica
Università degli Studi di Udine
Via delle Scienze, 206
33100 Udine — Italia
+39 0432 558400
http://www.dimi.uniud.it/

To my parents

Abstract

The theme of this thesis is sequencing (large) genomes and assembling them: an area at the intersection of algorithmics and technology.

The birth of next-generation sequencing (NGS) and third-generation sequencing (TGS) platforms dropped the costs of genome analysis by orders of magnitude compared to the older (Sanger) method. This event also paved the way to a continuously increasing number of genome sequencing projects and the need of redesigning several algorithms (as well as data structures) in order to cope with the computational challenges introduced by the latest technologies. In this dissertation we will explore two major problems: *de novo* assembly and long-sequence alignment. The former has been tackled, first, with a global approach and then by taking advantage of a hierarchical scheme (more natural considering the type of dataset at our disposal). The second problem, instead, has been studied in order to speed up a computationally critical phase of the first one. We also put a particular effort into the development of solutions able to scale on large datasets with the practical goal of reconstructing/improving the draft sequence of one of the largest genome ever being sequenced to date (*i.e.,* the Norway spruce).

The *de novo* assembly problem consists in the reconstruction of the DNA sequence of an organism starting from *reads* produced by a sequencer. It is tackled by softwares called *assemblers*, which adopt different models and heuristics in order to take advantage of the peculiarities of the input. Currently available methods, however, fail to deal with large and repetitive genomes and results are often unsatisfactory for carrying out many downstream analyses. In view of this, we propose a novel algorithm which follows a quite recent paradigm: *assembly reconciliation* (or merging). The key idea is to compare two or more genome assemblies and *merge* them in order to retain the best features from each one of them (while discarding/correcting possible errors). The algorithm we devised (GAM-NGS) made reconciliation to be feasible on large NGS datasets. In particular, the tool was able to provide an improved assembly of the 20-gigabase Norway spruce, using also a moderate amount of computational resources.

The second addressed problem (long-sequence alignment) consists in finding similarities between sequences. From a practical perspective, the alignment of reads against a known *reference* is crucial for evolutionary studies or to detect variants between sequences of the same species. Moreover, the very same assembly problem requires efficient algorithms and data structures to identify high-similarity overlaps among the reads and, therefore, to guide the assembly process. Our contribution on this matter consists in the development of a method to align and merge pools of long assembled sequences, each one representing a small fraction of the genome and independently assembled from NGS data. This type of datasets has been recently produced for aiding the reconstruction of two large and complex genomes (the Norway spruce and the Pacific oyster) in

order to seek for a trade-off between sequencing cost and assembly accuracy/contiguity. However, to the best of our knowledge, no formal solution has been proposed yet. For this reason, we introduced a framework (HAM) to carry out a Hierarchical Assembly Merging of such pools. Moreover, due to the large number of long sequences we expect to process, we devised a *fingerprint*-based algorithm for detecting overlaps between long and accurate sequences and which also achieves comparable results with state-of-the-art tools, while using considerably less computational resources.

Contents

List of Tables

List of Figures

Introduction

This dissertation focuses on the problem of reconstructing the genomic sequence of an organism – the *de novo* genome assembly problem – from a set of much shorter fragments, named *reads*, which are randomly extracted through the *sequencing* process. The reconstructed sequence is also called *assembly*. The *de novo* assembly problem is probably one of the most significant and studied matters of *bioinformatics* for biological and theoretical reasons. From a biological point of view, it is important for improving our knowledge on the relationships and the interactions within a genome and, hence, for understanding how an organism functions and evolves. Moreover, the assembly process is just the beginning of a quite vast set of downstream analyses that have important and direct applications. Additionally, the technological revolution brought by the recent sequencing technologies has concretely made the study of genomic sequences feasible for a large number of individuals. The characterization of sequences relative to healthy and diseased human cells also opened new possibilities in personalized medicine (*e.g.*, treatment and prevention of diseases). Such studies are usually carried out comparing a reconstructed sequence against an accurate *reference*. A fragmented, mis-assembled or unrepresentative reconstruction might not contribute to any practical use and, therefore, an *accurate* assembly is demanded for a large number of applications, starting from the very same reference definition (if not already available). This last need makes the assembly problem particularly interesting also from a theoretical perspective. The availability of precise models describing genome assembly, in fact, guides us to methods for dealing with the problem in practice.

Unfortunately, as we will often remark throughout the entire thesis, both *theoretical* and *practical* approaches struggle with two major deficiencies. First, theoretical models do *not* accurately formalize the genome assembly problem. Our incomplete knowledge about the non-random substructures of a genome certainly contributes to this limitation. Hence, even a precise modeling of the problem might lead us towards *biologically inaccurate* results. Practical approaches, instead, besides being related to possibly imperfect models, are furthermore based on heuristic methods due to negative theoretical results. The use of heuristics thus provide *approximate* results that can be either satisfactory or not. In general, as the complexity and the size of a genomic sequence increase, approximate methods are overwhelmed by theoretical and technological limitations and new perspectives on the problem should be sought.

A brief history of sequencing

The Sanger method [156] has been the first major breakthrough concerning our better understanding of the genome structure and, due to its simplicity, it became the gold-

standard approach for approximately thirty years. Such a method – often referred to as *first-generation sequencing* – has been continuously improved and automatized through the years and ultimately led to the full characterization of the human genome sequence [181]: a landmark for a number of following applications concerning, for instance, the understanding of diseases but also studies on other species. Unfortunately, the Sanger method is characterized by two major downsides: the relatively low throughput and the extremely high costs. This last shortcoming in particular limited whole-genome-shotgun (WGS) sequencing to large research facilities only and restricted its applications in terms of both the organisms that could be studied (*i.e.*, small-sized genomes or specific genomic regions) and the type of analyses that could be preformed. As a matter of fact, consider that it took approximately three billions of dollars and thirteen years of effort to complete the first human reference sequence.

This impressive result has been followed by several efforts to make sequencing faster and cheaper. This resulted in the advent of next-generation sequencing (NGS) technologies, which marked the second – and probably the most important – milestone concerning genome analysis. These sequencing technologies, in fact, have in common three fundamental hallmarks that distinguish them from the older methods: they are orders of magnitude *cheaper*, very *short fragments* are produced, and a orders-of-magnitude *higher throughput* is provided. Such a sequencing *revolution* paved the way to a large number of earlier unfeasible studies even to small research centers. In order to give an idea of the benefits introduced by NGS technologies, consider that it was then possible to sequence and complete the genome of a human individual in a couple of months at approximately one-hundredth of the cost [191]. Furthermore, recent technological advances now permit the sequencing of a human genome at the cost of 1000 dollars. Even if sequencing data can be directly exploited to perform several kinds of analyses, one of its main applications still remains genome assembly.

Currently, many efforts are being put on improving sequencing in order to increase read length without sacrificing throughput, costs, and accuracy. Recent technological advances, also known as *third-generation sequencing* (TGS), introduced novel methodologies to sequence DNA molecules. Specifically, they are currently able to produce kilobase-long reads that, in principle, would allows us to overcome part of the difficulties which are especially encountered in the assembly of complex genomes. Moreover, Eugene W. Myers recently argued [4] the TGS PacBio technology has also two peculiarities which may get us closer to the achievement of a near perfect assembly: (i) sequencing can be nearly approximated by a Poisson sampling of the genome, and (ii) errors are randomly distributed within reads. Hence, using well known statistical models [91] and provided a high-enough read coverage [41], building an accurate and complete *reference* assembly starts to seem possible. The improvement on read length however has a cost: the error rate is quite high and it can reach 15% (mainly insertions and deletions). Algorithms and data structures have been proposed to overcome this issue, yet the mandatory error correction phase is still computationally demanding. Despite the difficulties, it has been recently shown that a very accurate assembly of a microbial genome could be achieved solely with a TGS dataset [39].

Finally, an alternative to TGS is represented by the *hierarchical assembly* of pools, that seeks for a trade-off among sequencing costs, accuracy and computational complexity. This approach consists in dividing long DNA inserts into several pools which are

independently sequenced and assembled using NGS data. If a pool is sufficiently small (*i.e.*, representative of a little percentage of the underlying genome), then problems generated by the non-random sub-structures of the genome are less likely to appear and, therefore, to hinder the assembly process. Two representative sequencing projects which employed quite successfully this strategy are the Norway spruce [134] and the Pacific oyster [198]. Unfortunately, in such a hierarchical approach, available assemblers do not perform well with *assembled* sequences coming from the aforementioned pools and *ad hoc* methods are required.

Theoretical and practical *de novo* assembly

First models towards the definition of genome assembly were based on the shortest common super-string that explains (*i.e.*, contains) the set of sequenced reads extracted from the genome. Other frameworks instead consist in modeling the problem as finding optimum-weight paths in graphs built from the reads. This translates into solving a generalized Traveling Salesman Problem (TSP) on a String/Overlap graph or a Eulerian Super-path Problem on a de Bruijn graph. Despited not being a precise formulation of the genome assembly problem, such formalizations have all been proven to be NP-hard problems.

As a consequence, these hardness results made *de novo* assembly tools to heavily rely on heuristics and hence to "choose" between correctness, completeness, and computational complexity. Despite the "intractability" of the problem, quite good results have been achieved on Sanger-based datasets, thanks also to the long range information provided by the reads. For this reason, even if the formalization of the problem is difficult in general, such promising results led to think that probably the most difficult cases are restricted to unrealistic scenarios.

The peculiarities of NGS data, moreover, imposed two major issues: the efficient processing of a large amount of data and the feasibility of *de novo* assembly with very short fragments. The contribution of Mathematics and Computer Science methods has been crucial to address these problems and to make assembly and analyses viable with a moderate amount of resources. Specifically, computational efficiency and memory constraints demanded the development of *specific* algorithms and data structures in order to cope with NGS datasets and to perform different type of large-scale analyses (assembly, identification of structural variants, *etc.*). As a matter of fact, previous assemblers had been tailored to Sanger-based datasets and did not adapt to the newer input. As a result, very sophisticated solutions have been proposed in literature for minimizing the time and/or memory requirements for storing and processing the input sequences (*e.g.*, fingerprinting/hashing [108, 145, 182], suffix trees/arrays [179], the FM-index [58]) but also for assisting the assembly process (*e.g.*, de Bruijn graphs and variants). It is worth to mention that, from an asymptotic complexity analysis, some of the employed data structure are not optimal but still they perform much better in practice. The shorter read length instead makes genome assembly even more complex. This is especially due to the non-random structure of the genome: repeats, for instance, are the main cause of poor contiguity and mis-assemblies. This difficulty has been mainly addressed with an extensive use of paired sequences (*i.e.,* pair of reads sequenced from the same genomic *locus* and whose distance can be estimated). Nevertheless, NGS assemblers proved to be inadequate for many *de novo* assembly projects. Therefore, there is still room for

improvements, especially on large-scale *de novo* assembly.

Contributions of the thesis

This dissertation introduces several methods which enhance available approaches in two different settings: whole genome shotgun and hierarchical assembly. Briefly, we will take advantage of the fact that different heuristics/approaches are actually able to reconstruct *specific* regions of a genome accurately (see [26, 52, 154] for several assembly evaluations). Specifically, we will follow a recent paradigm whose purpose is to obtain an enhanced assembly starting from several other assemblies obtained with different heuristics. This strategy is called *assembly reconciliation* and consists in *merging* assembled sequences produced by different tools while detecting possible mis-assemblies and isolating problematic regions. Besides its first application on Sanger-based assemblies [32, 200], such a strategy has found its natural use with NGS-based assemblies, which are more frequently of unsatisfying quality. We will also show how assembly reconciliation could be extended to be used in a hierarchical setting thought for dealing with large-scale assemblies.

More precisely, this thesis is based on three main contributions related to the *de novo* assembly problem: an assembly reconciliation algorithm, a hierarchical (pool-based) assembly framework, a fingerprint-based overlap detection algorithm.

Assembly reconciliation. The first main contribution is GAM-NGS an *assembly reconciliation* algorithm which merges assemblies (obtained by different tools) in order to enhance contiguity and correctness of the reconstructed sequence. Differently with respect to other competing tools, GAM-NGS does not rely on global alignment but, instead, it takes exploits the mapping of a set of reads against the two input assemblies in order to detect regions (called *blocks*) representing the same genomic *locus*. The merging phase is carried out with the help of a *weighted graph G* (built from blocks) in order to resolve local problematic regions due to mis-assemblies, repeats, and compression/expansion events. Using state-of-the-art datasets and evaluation methods, we were also able to show that GAM-NGS achieves good results in terms of both computational requirements and accuracy.

During the PhD program, the candidate was also involved in the Spruce Genome Project in order to employ GAM-NGS's strategy to build the first draft of the Norway spruce assembly (one of the longest genomes ever sequenced to date). Specifically, GAM-NGS was used to merge a whole-genome-shotgun assembly with a collection of pool-based assemblies obtained using a hierarchical strategy and NGS technologies. As a result, the tool was able to greatly improve the assembly's contiguity and to decrease the evidences of putative errors (hence enhancing output's accuracy).

Both the aforementioned works/collaborations resulted in the following two publications (as first author and as co-author, respectively):

R. Vicedomini, F. Vezzi, S. Scalabrin, L. Arvestad, and A. Policriti. GAM-NGS: genomic assemblies merger for next generation sequencing. *BMC Bioinformatics*, 14(Suppl 7):S6, 2013

Björn Nystedt, Nathaniel R. Street, Anna Wetterbom, Andrea Zuccolo, Yao-Cheng Lin, Douglas G. Scofield, Francesco Vezzi, Nicolas Delhomme, Stefania Giacomello, Andrey Alexeyenko, Riccardo Vicedomini, Kristoffer Sahlin, Ellen Sherwood, Malin Elfstrand, Lydia Gramzow, Kristina Holmberg, Jimmie Hällman, Olivier Keech, Lisa Klasson, Maxim Koriabine, Melis Kucukoglu, Max Käller, Johannes Luthman, Fredrik Lysholm, Totte Niittylä, Åke Olson, Nemanja Rilakovic, Carol Ritland, Josep A. Rossello, Juliana Sena, Thomas Svensson, Carlos Talavera-López, Gunter Theiszen, Hannele Tuominen, Kevin Vanneste, Zhi-Qiang Wu, Bo Zhang, Philipp Zerbe, Lars Arvestad, Rishikesh Bhalerao, Joerg Bohlmann, Jean Bousquet, Rosario Garcia Gil, Torgeir R. Hvidsten, Pieter de Jong, John MacKay, Michele Morgante, Kermit Ritland, Björn Sundberg, Stacey Lee Thompson, Yves Van de Peer, Björn Andersson, Ove Nilsson, Pär K. Ingvarsson, Joakim Lundeberg, and Stefan Jansson. The Norway spruce genome sequence and conifer genome evolution. *Nature*, 497(7451):579–584, May 2013

Hierarchical pool-based reconciliation. As our second contribution we propose a model for building a draft assembly using the hierarchical pool-based approach (the one considered in the Norway spruce and in the Pacific oyster genome projects). This work has been motivated by the fact that currently *ad hoc* solutions have been considered and no particular framework has been presented to address in general such scenario. This resulted in a prototype tool called *Hierarchical Assemblies Merger* (HAM, for short) which is based on an Overlap-Layout-Consensus (OLC) paradigm and extends GAM-NGS's idea to the reconciliation of multiple pools in a hierarchical manner.

HAM's strategy and preliminary results have been presented in the following paper:

R. Vicedomini, F. Vezzi, S. Scalabrin, L. Arvestad, and A. Policriti. Hierarchical Assembly of Pools. In Francisco Ortuño and Ignacio Rojas, editors, *Bioinformatics and Biomedical Engineering*, volume 9044 of *Lecture Notes in Computer Science*, pages 207–218. Springer International Publishing, 2015

Fingerprint-based overlap detection. Our third and last contribution instead addresses a specific stage of the genome assembly process and is thought to be strictly coupled with HAM's work to address a large-scale assembly reconciliation. In particular, we developed a novel method to speed up and carry out the overlap detection phase required by HAM's framework.

Our method assumes that pools have been assembled in good-quality contigs and hence distinguish itself from competing tools which address the problem of detecting overlaps among long error-rich sequences. Moreover, in order to improve the computational effort required by currently available methods, we devised a "deterministic" fingerprinting technique (called DFP) which greatly reduces the size of the input and that allows to quickly identify – with also good precision – the pairs of sequences which are likely to overlap. The main feature of DFP is to compare sequences solely using their fingerprints and a more precise assessment/computation of the overlap is carried out only in a later stage. Due to the recent development of DFP, the method has not been published yet.

Outline of the thesis

This dissertation is divided into three parts and structured as follows.

Part I outlines some biological concepts and some theoretical and practical aspects concerning the main problems that will be addressed in the remaining parts. Specifically, in Chapter 1 we briefly present some important preliminary biological notions required to fully understand the rest of the thesis. In Chapter 2 we outline some algorithms and data structures which are commonly used to handle large-scale applications in bioinformatics and that are important for the assembly process. Finally, Chapter 3 is dedicated to the theoretical formulations of the *de novo* genome assembly problem.

Part II, instead, will focus on the practical aspects of assembly strategies. More precisely, Chapter 4 is related to the state of the art of methods employed to solve the assembly problem, while in Chapter 5 we describe our first main contribution, *i.e.,* the algorithm of our assembly reconciliation tool (GAM-NGS).

Part III eventually present a novel formalization and a practical implementation for the hierarchical pool-based assembly scenario: Chapter 6 describe the HAM framework and presents some preliminary results; Chapter 7 describe the two fingerprinting techniques we devised in order to speed up the overlap detection in HAM.

I

Genome Sequencing and Assembly

1

Preliminaries

In this first chapter we lay down the foundations of what we think are the most important biological aspects that the reader – who we do not assume to have a background in genomics – should acquire in order to follow the topics discussed in the rest of this thesis. We keep the biochemical details to a minimum and focus mainly on the processes which allows us to "decode" genomes (*i.e.,* sequencing). We believe that a basic understanding of how sequencing works and how it evolved throughout the last dozen of years is necessary to fully grasp many details of the computational challenges further discussed below. Readers already familiar with the subject may safely skip this chapter.

Genomics is the discipline that studies the structure and the encoded information of genomes [142]. More precisely, it deals with the reconstruction of the DNA sequence of organisms and the relationships and interactions within a genome. Every living organism is built up of one (*e.g.,* bacteria) or more (*e.g.,* plants, mammalians) biological units called *cells* wherein it is stored the information needed for the regulation of their specific functions and the evolution of the whole organism. Moreover, cells behave and interact differently depending on their specialized function (*e.g.,* think about the human organs) and the inputs from their surrounding environment.

The DNA, or deoxyribonucleic acid, is usually stored in each cell nucleus as a collection of molecules called *chromosomes*. Such molecules have two fundamental functions: first, replicating themselves and, second, storing, in regions called *genes*, the information which regulates the evolution and the function of each single cell. Genes are, in fact, the starting points of protein synthesis – one of the most important processes for cell's life. We can hence define the *genome* as the collection of DNA molecules (and their encoded genes) within a cell.

One of the most important reasons to study the genome of an organism is treating and preventing diseases. In fact, even though cells have nearly the same DNA "blueprint", this molecule is continuously subject to mutations (*i.e,* modifications of its structure). Genetic mutations could be inherited or acquired during lifetime (*e.g.,* cell division, environmental factors) and while some of them could be beneficial to create diverse and healthy populations, others might damage the DNA and lead to an improper protein synthesis along with different sorts of diseases.

The DNA structure. From a chemical point of view, the DNA can be seen as a large chain-like molecule made up by the concatenation of small repeating units called *nucleotides*. Each nucleotide consists of a sugar (deoxyribose) bound to a phosphate group and to a nitrogenous *base*. The latter can be of four different types: Adenine, Cytosine, Guanine, and Thymine. Hence, DNA molecules can be seen abstractly as strings encoded in the four-letter alphabet $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$, where letters stand for the first letter of the corresponding base. For the sake of simplicity, in the following chapters we might indistinctly use the terms nucleotide and base.

The DNA structure typically consists of two DNA molecules held together in a double-helix shape [188] and running in opposite directions (anti-parallel). A particularity of these coupled chains – the Watson and Crick *strands* – is that they are attached by bonds between *complementary base* pairs: adenine with thymine and cytosine with guanine (see Figure 1.1). Hereafter, we will often use the term *base pair* (bp) to identify one of these two bases. Looking closely to the DNA structure, it is possible to notice that if we pick one strand and we replace each base with its complementary one, we obtain exactly the other strand. Hence, both strands encode exactly the same information and it is precisely this redundancy, along with the stronger attraction between complementary bases, that makes the DNA replication "easy". More specifically, the synthesis of DNA is usually catalyzed by the so-called DNA *polymerase*, that is an enzyme able to bind nucleotides to a single-stranded DNA molecule in order to create a new the complementary strand.

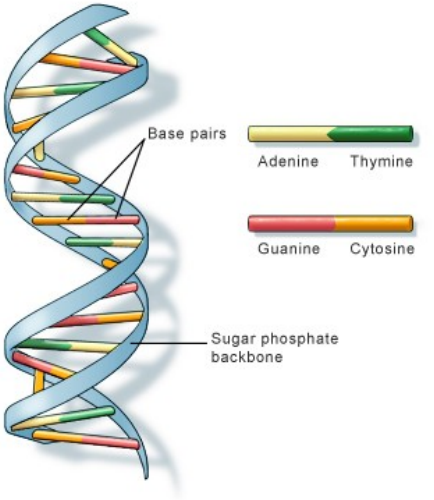


Figure 1.1: The DNA structure. Source: U.S. National Library of Medicine

Living organisms can be also classified by the number of chromosome copies within their cells. More precisely, an organism is said to be *haploid* if cells contain one copy of each chromosome, *diploid* if the copies are two, and, in general, *polyploid* for higher

copy numbers. Humans, for instance, are diploid and, hence, they are characterized by two *homologous* copies of their 23 chromosomes (one inherited from the mother, the other from the father). A particular hallmark of diploids and polyploids is that some regions (*loci*) of homologous chromosomes might be different. A specific region (*locus*) encoding a gene is called *allele*. When two alleles are identical (*i.e.*, the DNA sequence is exactly the same) the *locus* is defined *homozygous*, otherwise it is said *heterozygous*. A specific allelic variant is called *haplotype*.

1.1 Shotgun sequencing

Sequencing is the first mandatory step required to "decipher" genomes. Due to technological limitations it is currently impossible to decode exactly the sequence of very long DNA stretches. However, a strategy named *shotgun sequencing* consists in randomly *cut* DNA molecules into millions of shorter pieces called *inserts*, which can be sequentially "read" by a machine called *sequencer*. The reconstructed sequences are named *reads* and are usually encoded as strings in the alphabet $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. Read length and accuracy strictly depends on the employed technology and, as we will show in the following chapters, this leads to different computational approaches to solve the genome assembly problem.

Applications. The main application of DNA sequencing that we will cover in this dissertation is called *de novo* (genome) assembly and it aims at reconstructing the entire genomic sequence of an organism – the *reference* genome – starting from a set of randomly extracted reads.

Sequencing is also very useful to perform additional downstream analyses. For instance, when a reference is available, sequencing can be exploited to compare genomes of same-species (or closely-related) organisms to discover allelic variants or to perform evolutionary studies such as the annotation and the classification of transposable elements [99, 120]. Finally, a matter of primary interest is also the sequencing and the study of disease-driven variations in the haplotypes of a specific individual. This would allow us to possibly devise new treatments to cure/prevent diseases.

Paired reads. Almost all available sequencers are able to read from both the extremities of a specific DNA fragment, producing the so-called *paired reads*. The ability to estimate the *insert size* (*i.e.*, size of fragments) provides valuable information about the relative positions of reads in the genome. This additional knowledge has been proven to be extremely helpful in dealing with large and complex genomes (*e.g.*, repeat resolution, scaffolding) and also mitigated the drawback of having short reads in input. Paired reads can be further separated into two categories: *paired-end* (PE) and *mate-pair* (MP) reads. While the principle is similar – and they are often confused with each other – they are inherently different. For instance, from a computational point of view, the main differences are insert size (less than 1 Kbp for PE reads, larger for MP reads) and read orientations.

Read quality. Each base of a sequenced read is usually defined by the measurement of a signal (*e.g.*, fluorescent-dye/light intensity, electrical current disruption) which is

detected by the sequencer and it is used to distinguish nucleotides. The process of *base calling* consists of translating these measurements to a sequence of (discrete) nucleotides. Given that the quality of reads varies significantly for many reasons, a simple numerical estimation of base quality is very useful in order to improve genome assembly [56].

The quality value might be defined in several ways and it is related to the technology used. A common method is to relate the probability p of a wrong-base call to a log-transformed value Q using the following formula:

$$Q = -10 \cdot \log_{10} p.$$

This formula defines the so-called Phred quality score [56] and, for instance, a value of 30 means the probability of a specific base call being wrong is 0.001 (*i.e.*, 0.1%). Note that if $p = 1$ (*i.e.*, a base is certainly wrong), $Q = 0$ and it increases as the error probability diminish.

History of sequencing. Sequencing of DNA molecules began in the early 1970s with the development of the Maxam-Gilbert method [63]. Nevertheless, the first concrete sequencing strategy was introduced in 1975 by Frederick Sanger [156] and subsequently refined in order to sequence the first genome ever, *i.e.* the 5374-bp-long bacteriophage ΦX174 [155]. Sanger technology kept being improved and automatized through years, leading to sequencing the first human genome. The first draft was released in 2001 [181] and finished a couple of years later [44], overall requiring a 13-year effort and an estimated cost of 3 billion dollars.

The Sanger method remained the gold standard technique for almost thirty years due to its low complexity and the small amount of toxic/radioactive chemicals needed with respect to the Maxam-Gilbert method. However, the introduction of the Roche-454 technology in 2005 kicked off the so-called *Next Generation Sequencing* (NGS) revolution, as it was a massive improvement compared to previous technologies. As evidence of this, it was then possible to sequence and complete the genome of a human individual in a couple of months at approximately one-hundredth of the cost [191]. Several other different technologies have been commercialized in the following years such as Illumina-Solexa in 2006 and SOLiD in 2007. Specifically, they developed high-throughput and cost-effective techniques which made sequencing of billion-base genomes affordable to many research facilities. Moreover, they allowed to take a huge leap forward to the ambitious goal of sequencing the human genome with less than 1000 dollars. Objective eventually reached in 2014 with the Illumina's HiSeq X Ten sequencer: a machine able to deliver over 18 000 human genomes per year at the price of 1000 dollars each.

While NGS platforms are continuously improving both in terms of cost and throughput, recently, Pacific Biosciences (PacBio) commercialized a different sequencing approach. It is based on the so-called *Single-Molecule Sequencing* (SMS) principle and it is often classified as a *Third Generation Sequencing* technology. More precisely, PacBio proposed an advance towards the thousand-dollar-genome challenge with a fast and cost-effective method to sequence DNA, which is currently able to produce kilobase-long reads. Whereas most technologies require several identical copies of a DNA molecule before its sequence could be decoded, the peculiarity of the method described by Eid *et al.* in [53] is to directly sequence a single molecule of DNA in real-time. Despite not being as competitive as Illumina on accuracy, cost, and throughput, yet, it has been

Sequencer	Phred	Output/run	Time/run	Read length
AB 3730xl	Q20	1.92–0.92 Mbp	24 hours	550–1000 bp
454 GS FLX+	>Q30	450–700 Mbp	10–23 hours	450–700 bp
AB SOLiD v4	>Q30	120 Gbp	7–14 days	2×50 bp
AB SOLiD 5500xl	Q40	up to 300 Gbp	7 days	75-35 bp (PE)
Illumina HiSeq 4000	>Q30	125–1 500 Gbp	$<$ 1–3.5 days	2×150 bp
Illumina HiSeq X Ten	>Q30	1.6–1.8 Tbp	$<$ 3 days	2×150 bp
Ion Torrent PGM	Q20	100–200 Mbp	2 hours	200–400 bp
PacBio RS	<Q10	0.5–1.0 Gbp	2–4 hours	1.5 Kbp
PacBio RS II	<Q10	0.5–1.0 Gbp	0.5–4 hours	$\sim$ 10 Kbp

Table 1.1: Features of modern sequencing technologies [103, 146].

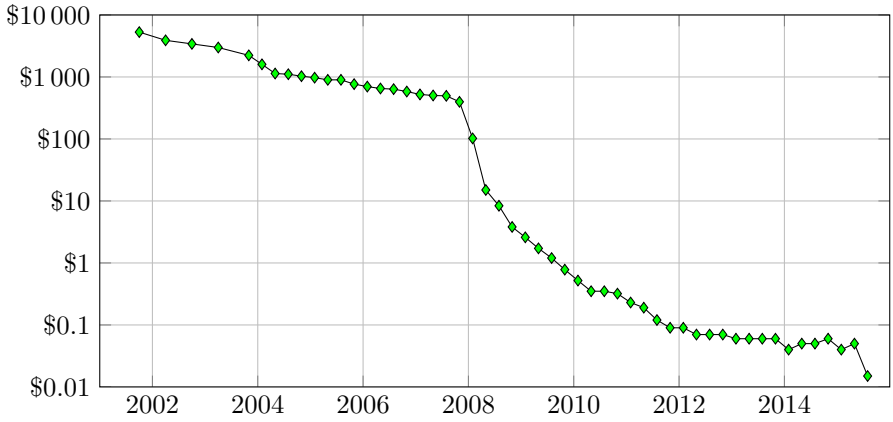


Figure 1.2: Drop of sequencing cost per Mbp throughout last 15 years. Source: National Human Genome Research Institute.

shown that using PacBio sequencing it is however possible to achieve better results on microbial genome assembly [39].

We are now going to delineate those aspects that make each sequencing technology unique, focusing mainly on throughput, cost, read length, and expected accuracy. The latter two are the reasons why many different combinatorial approaches and sophisticated data structures have been devised for the genome assembly problem. Difference among technologies are summarized in Table 1.1.

1.1.1 Sanger (first-generation) sequencing

Also known as the chain termination method, this technique falls into the category of the so-called *First Generation Sequencing* technologies. For three decades, *de novo* genome sequencing projects, such as the Human Genome Project, have been carried out using capillary-based semi-automated implementations of this method [80, 155, 174].

High-throughput Sanger shotgun sequencing [14] protocol typically consists of the

following steps. The DNA molecule to be sequenced – the *target* DNA – is first randomly sheared into small fragments (*e.g.,* using restriction enzymes). These pieces, also called *inserts*, are subsequently ligated with viral/plasmid *cloning vectors*, *i.e.* small DNA fragments able to replicate themselves together with the attached molecule. These spliced sequences are then introduced into host cells – typically harmless strains of the *Escherichia coli* bacterium – wherein they are *amplified* (*i.e.,* replicated). This process yields to a collection of organisms in which the target DNA, along with the host DNA, is present in high-number copies. The vector DNA can be finally extracted to allow the actual sequencing of the target DNA fragments through a series of biochemical reactions [159]. More precisely, modern implementations generate multiple different-size fragments, each one starting from the same location and such that the last base is labeled with a fluorescent dye – such that the four nucleotides could be distinguished. Fragments are finally sorted by their length in order to determine the sequence.

Sanger sequencing was at first a quite complex process: it required the use of radioactive materials and a large amount of infrastructures and human labor. In 1987 the introduction of the first semi-automated machine – thanks to Applied Biosystems – made sequencing faster and less error-prone: it was then possible to output 500 Kbp/day and sequence 600-base-long reads. It was also due to Applied Biosystems, along with the launch of the PRISM 3700 DNA Analyzer, that the Human Genome Project was completed by nearly five years ahead of schedule [43, 181]. Now, after three decades of technological refinements, their latest model (AB 3730xl) can achieve read lengths up to 1 000 bp, a per-base accuracy of 99.999%, and a cost in the order of $0.50 per kilobase [159]. Even though it has been overwhelmed by significantly faster and cost-effective methods, nowadays, this technology is still used for finishing genomes (*e.g.*, gap closure) or to accurately sequence short regions of interest.

1.1.2 High-throughput (second-generation) sequencing

Often referred to as *Next Generation Sequencing* (NGS) technologies, the methods we are going to outline in this section allowed sequencing to be orders of magnitude cheaper and characterized by a much higher throughput with respect to the chain termination method. The main drawback, however, is that read lengths are much shorter and this, along with the huge amount of data produced by sequencers, imposed new algorithmic challenges in genome assembly and downstream analyses. In the last dozen of years many high-throughput systems were commercialized, however we will describe the features of those which are more popular and that have been used in major large-scale sequencing projects: 454, Illumina, and SOLiD. In contrast to Sanger sequencing, these platforms avoid bacterial cloning (potentially being able to sequence DNA that cannot be propagated in bacteria). More precisely, DNA molecules are amplified in spatially-separate locations of highly parallel arrays [19]. Based on the same paradigm – namely *cyclic-array sequencing* – implementation and biochemistry are much different among these systems [160]. This have repercussions on throughput, run time, read length and costs (see Table 1.1).

454. Roche 454 was the first NGS platform commercialized. In order to achieve an approximately 100-fold increase in throughput over Sanger sequencing technology [110],

454 developed a *sequencing-by-synthesis* approach called *pyrosequencing* [151]. In simple terms, it is based on measuring – through a camera – the light emitted when the next complementary nucleotide is sequentially incorporated on a single-stranded DNA template (the target sequence). The intensity of the light is directly correlated to the amount of bases being incorporated and, if the same nucleotide is added n times in a row, the signal is expected to be n times larger. The accuracy of pyrosequencing hence relies on the precise measurement of the emitted light. As a consequence, the major limitation of the 454 technology relates to consecutive instances of the same base (*e.g.*, . . . `AAAAAA` . . .) and, unlike Illumina and SOLiD platforms, 454 data-sets frequently suffer the presence of insertions and deletions [147].

Currently, 454 platform provides a higher per-base cost with respect to its NGS competitors. Its strength however is to output almost a billion base pairs per day of *single-ended* (SE) reads whose lengths are on average 450–700 bp. Despite the higher cost, it still represents a viable and recommended choice in projects where read length matters (*e.g., de novo* assembly).

Illumina. Like 454, this platform is also based on the *sequencing-by-synthesis* principle and, even though reads are much shorter than 454's, this technology allowed to diminish the per-base cost and increase throughput significantly. To achieve this, Illumina's sequencing protocol is based on the so-called *bridge amplification*:

1. DNA molecules are first sheared into fragments that are short-enough to be sequenced. Fragments are then denaturated (*i.e.,* strands are separated) and ligated to specific *adapters* (*i.e.*, short synthetic DNA sequences) which are attached to both the extremities. Afterwards, fragments are bound to a solid surface – namely *flow cell* – where many clusters of reverse complementary copies of the adapters had been previously fixed.

2. A local amplification process is repeated several times: hanging adapters ligate to nearby complementary adapters (thus forming a *bridge*, see Figure 1.3); unlabeled nucleotides and polymerase enzymes are added to initiate the synthesis of fragments' complementary strands; double stranded molecules are finally denaturated to allow successive iterations. In this way the flow cell will be covered with clusters of thousand copies of the formerly attached fragments.

3. The system now sequences all clusters simultaneously. More precisely, nucleotides are modified such that their incorporation – and hence the replication process – is controlled step by step. A fluorescent dye is also employed in order to decode which bases have been added at each cycle through highly sensitive optics. Finally, before the next reading cycle starts, a "washing" step is performed in order to remove both the "termination feature" from the inserted nucleotides and the fluorescent dye.

As a consequence of the employed *wash-and-scan* paradigm, base-reading accuracy drops quickly and this is the reason why much shorter reads are produced with respect to 454 sequencing. On the contrary, Illumina sequencing is not hindered by consecutive instances of the same base in sequenced fragments.

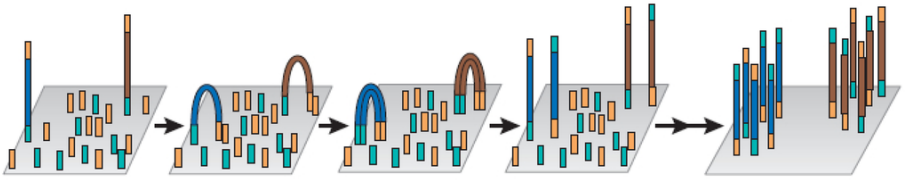


Figure 1.3: Illumina's *bridge amplification*.

Even though read length is significantly smaller than 454, the throughput is unmatched. As a matter of fact, Illumina sequencers are currently able to produce billions of 150 bp paired reads within a couple of days (see Table 1.1). Moreover, due to the massive amount of information generated and the higher 454 cost, Illumina's platform is nowadays the first choice in most large sequencing projects.

SOLiD. Currently produced by Applied Biosystems, it was the latest-commercialized sequencer of the first three major NGS platforms. Even tough this technology provides shorter reads than Illumina, a particular effort was put on accuracy and throughput. When it was presented, in fact, it allowed to achieve a higher throughput at a slightly lower per-base cost [109].

The first DNA fragmentation and amplification phase recalls the process employed by the 454 platform. However, amplified sequences are inserted on a glass surface where sequencing is based on the different *sequencing-by-ligation* principle.

More precisely, SOLiD sequencing involves sequential rounds of hybridization and ligation processes using dinucleotides, *i.e.* two-nucleotide sequences, labeled by four different fluorescent dyes. Being only four available colors and sixteen possible dinucleotides, each color encodes four different nucleotide *transitions* (see Figure 1.4). Moreover, each base of the target sequence is effectively covered in two different reactions by two 1-base shifted dinucleotides. This allows to decode a specific base by looking at the colors of two successive ligation reactions. As the last base of the adapter is already known, the color sequence can be successfully translated to the corresponding nucleotide sequence.

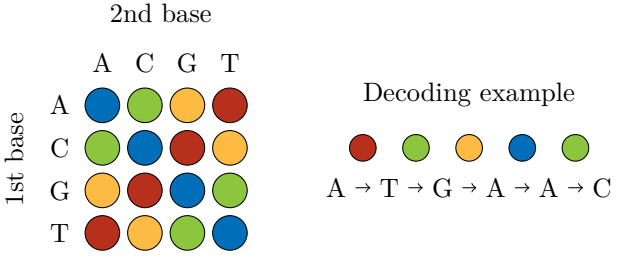


Figure 1.4: SOLiD color-space encoding.

Being the number of *legal* two-color transitions limited, the main advantage of this sequencing/encoding technique is to effectively allow the detection of sequencing errors or single-nucleotide variants (when a reference is available). As a matter of fact, SOLiD

platforms proved to be particularly useful in structural variation analyses and, generally, resequencing projects [111, 163]. SOLiD instruments are nowadays able to output 300 Gbp of 75 bp fragments or 75-35 bp PE-reads in a single run of 7 days. However, due to the very small read length and the need of dedicated software to effectively handle the color-space encoding, this technology is often ruled out in favor of Illumina – especially when large-scale *de novo* assembly is concerned.

Ion Torrent. This technology is actually considered to lie between second and third generation sequencing technologies [157]. It implements a *sequencing-by-synthesis* principle carried out by a high-density array of semiconductors capable to perceive the protons released while nucleotides are incorporated during the synthesis [152]. This particular technology allows to diminish both time and sequencing costs. Being a *wash-and-scan* system, however, read length is limited. Latest Ion Torrent's Personal Genome Machine (PGM) is able to produce 200-bp reads in less than 2 hours.

1.1.3 Single molecule (third-generation) sequencing

Over the last ten years, second-generation sequencing platforms have been optimized improving cost, throughput, time, and accuracy, hence allowing a more complete (and beyond expectation) understanding of the information encoded within whole genome sequences [119]. During these years Illumina has undoubtedly been the leader in cost-effective high-throughput massively parallel sequencing [20].

Several companies, however, explored the possibility to directly sequence single molecules, hence avoiding the canonical amplification phase [19]. These strategies are often referred to as *Third Generation Sequencing* (TGS) technologies and they have been introduced from 2009. Currently, three major TGS systems have been (or soon will be) commercialized: Helicos Genetic Analysis, Pacific Biosciences (PacBio), and Oxford Nanopore.

Helicos Genetic Analysis. This is the first *true* single-molecule sequencing system being commercialized [25, 70]. In this method, (short) fragments of DNA molecules are first attached on glass flow cells. Subsequently, fluorescent nucleotides are added one at a time with a terminating nucleotide which halts the process so that a "snapshot" of the sequence can be taken [178]. The main drawback of this method is the high time required to sequence a single nucleotide and the small read length (approximately 32 bp). Moreover, like any other SMS system, the error rate is typically grater than 5% and mostly involves insertions and deletions. The highly parallel nature of this technology however allows to produce corrected reads with high-accuracy ($> 99\%$).

Unfortunately, this method was not very successful as it was not much different from the leading second-generation technologies: read lengths, throughput, and run times are in fact comparable to top-notch NGS platforms. As a matter of fact, this hallmark, combined with the higher error rates, usually translates into a more expensive sequencing.

PacBio. The single-molecule real-time (SMRT) sequencing approach developed by Pacific Biosciences is the first TGS approach to directly observe the single nucleotides

being incorporated by DNA polymerases as they synthesize the complementary strand of a target DNA molecule [53].

The first problem encountered during the development of this method was observing DNA polymerase's work in real time – as it binds nucleotides – and it was solved exploiting the zero-mode waveguide technology [93]. A second problem was due to the typical use of fluorescent dyes attached to the bases as they can inhibit polymerase's activity (this is the reason why most sequencing approaches synthesize DNA molecules one nucleotide at a time). In order to overcome this issue, the dye is attached to the phosphate group of the nucleotide. In this way, during the synthesis, it is automatically cleaved when each nucleotide is incorporated.

PacBio sequencing is currently able to output reads which are one or two orders of magnitude longer than any other first- or second-generation technology (see Table 1.1). More precisely, this platform is currently able to produce reads which are typically 10 Kbp long (or even longer).

Despite many potential improvements are expected by the use of such long reads, this technology poses also new challenges due to error rates reaching 15% (as reported in [34]). Fortunately, errors are uniformly distributed and uncorrelated between different sequences. Hence, provided a high-enough coverage or exploiting NGS data-sets [90], it has been shown that PacBio reads can be effectively corrected (yet with a significant computational effort) in order to obtain highly accurate sequences.

Oxford Nanopore. Nanopore-based technologies are able to detect nucleotides while a DNA molecule passes through a *nanopore* (*i.e.*, a nano-scale hole) by measuring their response to an electrical signal. Like PacBio's SMRT, nanopore technologies have the potential to be extremely fast while using a small amount of input material (remember that amplification is not needed in SMS) [157].

Oxford Nanopore is currently developing and commercializing different devices able to analyze single molecules such as DNA, RNA, and proteins, using nanopore sensing technologies. More precisely, an ionic current flows through a nanopore (by setting a voltage across it) and the characteristic current's disruption – due to nucleotides passing through/near the pore – is exploited to identify a target molecule [42, 76, 171].

Like other SMS technologies, the expected advantages of nanopore-based sequencing are long reads, high scalability, low costs, and small instrument size (*e.g.*, Oxford Nanopore's MinION sequencer, which is being released through an early-access program, is a pocket-sized portable device). This technique however is not exempt from the high error rates (between 5 and 40%) [66] typical of SMS implementations and, hence, from a (computationally expensive) mandatory error correction pre-processing.

1.2 Coverage, read length and assembly contiguity

In a sequencing process reads are randomly sampled (extracted) from a genome. Due to their very small length (w.r.t. the DNA molecules), it is clear that extracting a high-enough number of them is a necessary condition in order to reconstruct most of the genomic sequence (even in the absence of repeats). As a matter of fact, in the best case scenario, in case some regions of the genome were not sufficiently covered, the

assembly would consist of a *set* of sequences, named *contigs* [169]. Moreover, a high-enough coverage does not only increase the contiguity of an assembly but also improves accuracy: a single read might have an error rate of 1%, while an 8-fold coverage has an error rate as low as 10^{-16} when eight high-quality reads agree with each other [158].

The *coverage* c of the sequencing process can be defined as the average number of reads covering each single nucleotide of the genome (*i.e.*, extracted from that *locus*).

Definition 1.1 (Coverage)**.** Let n be the number of input reads, l be their length, and $|g|$ be the genome size. The (sequencing) coverage c is defined as

$$c = \frac{nl}{|g|}.$$

The notation $c\times$ is often used to indicate that g has been sequenced with coverage c. Since the genome size might be unknown, in practice, a rough estimate of $|g|$ can be computed analyzing, for instance, the k-mer spectrum (*i.e.*, the distribution of the substrings of length k contained in the input dataset) or using flow cytometry.

Sequencing coverage, however, is not the only parameter influencing the assembly pipeline. Sequencing distribution and read length are also extremely important. In 1988 Lander and Waterman proposed a model [91] applicable to shotgun sequencing. More precisely, the two fundamental parameters involved in this study are

- the read coverage c,
- the length t required to detect an overlap between two reads extracted from the same genomic *locus*.

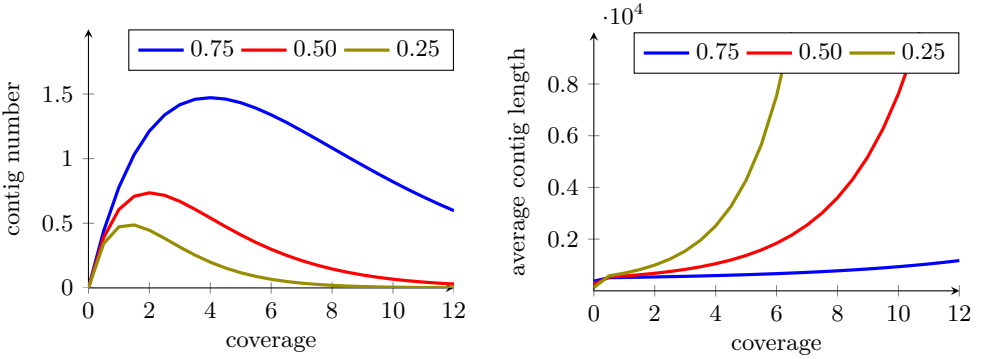
Assuming that the probability one of the n reads starts at a specific position is very low, Lander and Waterman formally proved the following results:

i. the expected number of contigs is $\frac{|g|}{l} c \cdot e^{-c\sigma}$,

ii. the expected contig length is $l(\frac{e^{c\sigma}-1}{c} + (1-\sigma))$,

where e is the Napier's constant and σ is the maximum read fraction not involved in an overlap, that is $\sigma = 1 - \frac{t}{l}$. Figure 1.5 depicts the expected number and length of contigs as a function of coverage.

Using Lander-Waterman model in the context of high-throughput short-read sequencing, however, the coverage needed in order to expect long contigs might be underestimated. Previous results, in fact, assume that sequencing is a Poisson process. While this assumption makes sense with a sequencer able to produce a low coverage of long and accurate reads (*e.g.*, Sanger), this hypothesis is no longer valid for NGS and TGS sequencing projects. In NGS data-sets, for instance, errors, repeats and other complicating factors often require a coverage as high as $100\times$ in order to deal with the assembly of large and complex genomes [65] (much greater than the one computed with the aforementioned model). Nagarajan and Pop [126] tried to explore the connections between repeat complexity, (short) reads, overlap lengths and coverage. Specifically, they showed how a short read length increase the difficulty of the problem and also proved several



(a) Expected contig number as function of coverage and expressed as a multiple of $\frac{|g|}{l}$.

(b) Expected contig length as function of coverage for $l = 500$.

Figure 1.5: Lander-Waterman statistics for different values of $\frac{t}{l}$, that is the minimum portion of a read needed to detect overlaps with other reads.

hardness results for the assembly problem. The inaccuracy of the long TGS reads, instead, demands a higher coverage for the preliminary error-correction phase. However, once such reads are corrected, the hypotheses might be used in order to consider a lower coverage for the assembly.

2

Sequence alignment

Before diving into the main matter of this dissertation, we briefly outline the algorithms and data structures commonly employed within the assembly process. As we will report later (starting from Chapter 4), the development of clever methods allowed, first, to obtain practical solutions to the "easier" assembly of Sanger reads and, later, to efficiently handle the newer challenges imposed by the successive generations of sequencing technologies. Specifically, two major tasks became of critical importance: the efficient processing of *large* datasets of very *short* reads, and the use of *error-rich* sequences to improve the assembly's quality. In the last decade, this translated into the development of very effective theoretical and practical methods which were able to cope with such *large* amounts of *error-prone* data. As far as genome assembly is concerned, the aforementioned tasks are usually tackled by employing advanced data structures (*e.g.,* succinct indexes, fast-lookup hash tables) and algorithms that often take advantage of the capabilities of the underlying hardware.

One critical stage of the assembly process, that is affected from the large amount of data provided by next-generation sequencing technologies, is sequence alignment. The task consists in finding regions of close similarity between a *query* sequence and a set of *target* sequences. When the number of searched sequences is huge and/or the target is too large, the alignment process becomes computationally challenging. For this reason, in practice, the problem is usually addressed by building an index from the data in order to perform searches in a (space and time) efficient manner.

In this chapter we will outline some of the most important methods and data structures which are used to solve the alignment problem. Specifically, in Section 2.1 we provide some preliminary definitions, in Section 2.2 we describe the most important suffix-based data structures, and in Section 2.3 we describe some alternative methods particularly suited for the inexact alignment problems.

2.1 Fundamentals of the alignment problem

Let Σ be a finite alphabet and Σ^* be the set of all strings over it. Specifically, we will consider Σ to be defined as the DNA alphabet (*i.e.,* $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$), although the algorithms

and data structures we are going to outline simply require $|\Sigma| \geq 2$.

Given a generic string $s \in \Sigma^*$, we denote its length by $|s|$ and its ith character with s_i (or $s[i]$), where $i \in \{1, \ldots, |s|\}$. Given two indices i, j such that $1 \leq i \leq j \leq |s|$, we define the *sub-string* of s from the ith to the jth character (included) as $s_{i,j} = s_i s_{i+1} \cdots s_j$ (or $s[i, j]$).

Definition 2.1 (The alignment problem)**.** Let $p, t \in \Sigma^*$ be two strings and consider a *distance* function $\delta : \Sigma^* \times \Sigma^* \to \mathbb{R}$. The alignment problem is defined as finding the following set:

$$\mathcal{I} = \{\ (i, j) \mid \delta(p, t_{i,j}) \leq k\ \wedge\ 1 \leq i \leq j \leq |t|\ \},$$

where $k \in \mathbb{R}$ is the maximal allowed distance between two aligned regions. In other words, we want to find all the positions (or *occurrences*) of those sub-strings of t which are *similar* (yet not necessarily equal) to p according to δ.

In general, for $k > 0$ the problem is usually referred to as finding an *approximate* alignment and, in our scenario, it is the most interesting one due to the fact that the strings we want to process are affected by errors. Moreover, in DNA sequence alignment, δ is commonly defined according to two metrics: the *Hamming* distance and the *Levenshtein* (or *edit*) distance.

The first one simply represents the number of mismatches (*i.e.*, differences) between two strings a and b. Formally, the Hamming distance $\delta_H : \Sigma^n \times \Sigma^n \to \mathbb{N}$ is defined as $\delta_H(a, b) = \Sigma_{i=1}^{n} h(a_i, b_i)$, where $|a| = |b| = n$, $h(x, y) = 1$ if $x \neq y$, and $h(x, y) = 0$ otherwise. The main downside of such a metric is that it can be used on equal-length strings only. However, it has been particularly adopted to tackle the alignment of short reads (*e.g.,* Illumina) [182], as they are much less affected by insertion/deletion errors.

The Levenshtein distance, instead, is tightly related to the representation of an alignment as a list (string) of operations (*i.e.*, matches, insertions, deletions, and substitutions) that can be "executed" as a program to convert a sequence into another one. In this way, the edit distance $\delta_E : \Sigma^* \times \Sigma^* \to \mathbb{N}$ can be simply defined as the *minimum* number of edit operations needed to perform such a transformation (matches excluded). In practice, to each edit operations are also associated specific costs and the goal of the alignment problem usually becomes finding alignments which optimize the cumulative cost of the edit string.

Alignment methods can be further divided into two main categories: *global* and *local* algorithms. The former consists in finding an optimal edit string which transforms an *entire* sequence a into an *entire* sequence b. The latter, instead, involves finding optimal alignments between an *entire* sequence a and *portions* of a sequence b (or maximal alignments between portions of both sequences). Furthermore, a special case is represented by *semi-global* alignments which are constrained to the start and/or to the end of one of the sequences. When we seek an alignment which is bonded to both the start and the end of either one of a pair of sequences we refer to an *overlap detection* problem, that is, the search of optimal prefix-suffix alignments above a certain length threshold within a collection of input strings.

The Needleman-Wunsch [129] and the Smith-Waterman [164] algorithm are well-known dynamic programming methods that allows to compute global and local *optimal* alignments, respectively. A semi-global alignment can be computed simply with a variant of the Smith-Waterman algorithm.

2.2 Suffix-based data structures

2.2.1 Suffix tries and trees

Suffix tries. A *suffix trie* (or, simply, *trie*) is a data structure that stores all the suffixes of a string. Formally, a suffix trie of a string $t = t_1 t_2 \cdots t_n$ is a rooted labeled tree with the following properties:

- each edge is labeled with a letter from Σ;
- any two edges leaving the same vertex have distinct labels;
- every suffix of t (*i.e.*, $t_{i,n}$ for $i = 1, \ldots, n$) corresponds to a unique path from the root to a leaf.

In the above definition there is a problem when a suffix of t occurs as a prefix of a different suffix. In order to avoid that, a special character $\$ \notin \Sigma$ is added at the end of t.

Suffix tries allows us to determine whether a string p occurs (exactly) within t in $\mathcal{O}(|p|)$ time by visiting the trie from the root and checking whether there is a path which spells p or not. The main drawback of tries, however, is the space complexity which is proportional to n^2 (and hence not viable for many biological applications)

Suffix trees. An improvement over tries has been brought by *suffix trees*. As a matter of fact, this data structure not only allows to search for a pattern in time proportional to $|p|$ but also occupies a linear amount of memory. Another remarkable peculiarity of suffix trees is the possibility to build them in linear time. Peter Weiner was the first to provide a solution [189]. Nevertheless, Esko Ukkonen later devised a conceptually simpler algorithm [179] which paved the way to a number of successful applications in bioinformatics. Ukkonen's algorithm has been also preferred in practice due to its lower memory requirements and for the on-line construction of the suffix tree. More in detail, a *suffix tree* is nothing but a compact representation of a trie and, in order to achieve the linear-space complexity, edges of linear paths are compressed into a single one which additionally stores two indexes on t (to avoid storing the actual sub-string).

Despite the optimal theoretical complexity, the major downside of such an index is that it requires more memory than the actual string. As a matter of fact, most space-efficient implementations in bioinformatics use 12–17 bytes per nucleotide (impractical for large-scale applications).

2.2.2 Suffix arrays

In 1990 Manber and Myers introduced *suffix arrays* – a space-efficient alternative to suffix trees – and described the first algorithms for their construction and use [107].

The suffix array SA_t of a string t can be simply seen as a sorted list of all suffixes of t. More formally, it is an integer array of size $|t| = n$ and such that $SA_t[i] = j$ if and only if $t_{j,n}$ is the ith suffix of t in ascending lexicographic order.

A basic implementation allows to build a SA_t in $\mathcal{O}(|t|)$ time and requires only 4 bytes per character (a significant improvement over suffix trees). Unfortunately, the search

for exact occurrences of a pattern p is based on a simple binary search and, hence, proportional to $|p| \cdot \log |t|$. Nevertheless, it is possible to get rid of the $\log |t|$ factor by enhancing suffix array with an additional data structure which stores the information about the *longest common prefixes* (LCP). In this way, finding pattern occurrences can be carried out in $\mathcal{O}(|t| + \log |t|)$ [87], without compromising space efficiency. Moreover, it has been also showed that suffix arrays can substitute each suffix tree algorithm and solve the same problems with the same computational complexity [10]. This last result has been achieved with a sophisticated implementation of the suffix array (plus the LCP structure) characterized by a slightly larger memory footprint.

Even though the canonical implementation of suffix arrays is not asymptotically optimal due to the logarithmic factor, it is worth mentioning that operations performed on this data structure are usually faster compared to suffix trees. This is mainly due to the fact that the logarithmic factor is small in practice and to the improved cache locality of the array.

2.2.3 The Burrows-Wheeler transform

Given a string t, the Burrows-Wheeler transform (BWT) [29] can be defined as a permutation BWT_t of t's characters. The BWT has two fundamental properties making it interesting for text-processing purposes:

1. it is *lossless*: from BWT_t it is possible to easily reconstruct t,

2. it can be highly *compressible*.

In order to build BWT_t it is assumed that a character $\$ \notin \Sigma$ had been appended at the end of t. Consider now a square $|t| \times |t|$ matrix M where rows are characterized by all the cyclic permutations of t sorted by (ascending) lexicographic order. The last column of M defines BWT_t (*i.e.*, $BWT_t[i] = M[i, |t|]$, for $i \in \{1, \ldots, |t|\}$).

It is worth to observe that, as the BWT sorts repeated sub-strings into contiguous intervals, the transformed string is likely to contain runs of repeated symbols. This allows the efficient compression of BWT_t using, for instance, a run-length encoding algorithm. This strategy proves to be particularly effective for compressing high-coverage DNA sequencing data as the length of the character (nucleotide) runs depend on coverage [161].

Another interesting property of BWT_t is that it can be defined in terms of SA_t. Formally, given an element $SA_t[i] = j$ of the suffix array, then

$$BWT_t[i] = \begin{cases} t_{j-1} & \text{if } j > 1, \\ \$ & \text{otherwise.} \end{cases}$$

From a practical point of view, recently, Hon *et al.* [74] showed that the BWT of a human genome can be computed with just $\mathcal{O}(n)$ bits of working space and using up to 1 GB of RAM. A slightly better space occupancy could be achieved with the recent result from Policriti *et al.* [143] which uses about 2.6 bits per input symbol.

The FM-index.

Ferragina and Manzini obtained a major improvement over suffix-based indexes with the definition of the FM-index [58]: a succinct data structure to be coupled with the Burrows-Wheeler transform and suffix arrays, characterized by a $3n$-bit footprint per character.

Specifically, the FM-index allows to efficiently compute the number of occurrences of a pattern p in a string t in $\mathcal{O}(|p|)$. This result is achieved with the so-called *backward search algorithm* which is based on the key observation that occurrences of p in t induce an interval in SA_t which can be defined using two pre-computed structures:

- $C(a)$: the number of occurrences of lexicographically smaller characters than $a \in \Sigma$ within t.
- $Occ(a, i)$: the number of occurrences of a in the sub-string $BWT_t[1, i]$.

Exploiting these two structures the positions where p occurs can be found in $\mathcal{O}(|p| + k)$, where k is the number of p's occurrences.

From a practical point of view, sophisticated implementations of this compressed index allows to store the entire human genome in 2 GB.

2.2.4 Suffix-based alignment in practice

All the data structures outlined in this section are certainly well suited for the exact alignment problem. Unfortunately, they are less able to cope with the inexact variant. As a consequence, short-read alignment (as well as overlap detection) is usually carried out either searching for exact occurrences of small sub-sequences, or by generalizing known algorithms, using strategies such as *backtracking* and other hybrid techniques, in order to deal with mismatches (but also risking to compromise the run time in practical scenarios).

Well known BWT-based tools are BWA [95], BWA-SW [96], BOWTIE [92], SOAP2 [100], BLASR [34], BWA-MEM [94], and BW-ERNE [145]. It is worth to notice that BWA-MEM, BWA-SW, and BLASR – while making use of a FM-index – are mainly based on a (hybrid) seed and extend approach.

2.3 Seeds and fingerprints

Exact algorithms are usually not well suited to find approximate alignments. For this reason, most tools commonly perform the search using small *exact seeds* (*i.e.* fixed-length sub-strings extracted from the sequences) or *fingerprints*. Hereafter we present some of the techniques based on this approach and that are related to one of our main results which will be presented in Chapter 7.

2.3.1 Seed-based alignment

The first stage of seed-based algorithm is to detect seeds shared by two or more sequences: this allows to quickly identify a *limited* subset of putative approximate alignments. Seeds are extracted from the input sequences and, typically, indexed in *hash*

tables or *properly-filtered sorted tables*. Matching seeds are subsequently extended using sophisticated (yet slower) implementation of dynamic-programming algorithms, such as Smith-Waterman [57, 175]. This approach is also referred to as *seed and extend* [60]. Here we will outline some of the hash-based strategies/tools commonly employed in sequence alignment.

Very popular seed-based tools are BLAST [13], SSAHA [131], BLAT [89], Novoalign [72], and DALIGNER [124].

Spaced seeds. A common heuristic for the approximate alignment problem is to look for high-scoring matches starting from exact seed hits. Such hits are the first evidence of a putative larger similarity between sequences. However, it has been shown that allowing few mismatches in seed hits can increase sensitivity without any loss of specificity [31, 105].

In general, a seed can be defined from a binary string p, called *seed pattern*, which defines what positions of the seed require a match (*i.e.*, $p_i = 1$) and those for which we *do not care* if they match or not (*i.e.*, $p_i = 0$). A seed pattern of length m that contains at least a *do-not-care* position between two matching positions is called *spaced seed* and its weight is defined as $w(p) = \Sigma_{i=1}^{m} p_i$.

To get the idea of the improved sensitivity of these seeds, consider two sequences both 20 bp long and with 70% similarity (*i.e.,* the probability of a match is 70%). Using 5-bp exact seeds, the chances of getting at least one hit is approximately 73%. Instead, using the pattern `110111`, which has the same weight but allows for a mismatch, increases the probability of having at least a hit to 80%.

However, the performance of a spaced seed strictly depends on how its pattern is defined: two distinct equal-weight spaced seed which differs in just one *do-not-care* position might have different sensitivities on sequences with different degrees of similarity. A study on how seed sensitivity can be computed has been done in [28].

Known alignment tools based on spaced seeds are LASTZ [69], ZOOM [101], PerM [37], SToRM [132], and SpEED [84].

2.3.2 Fingerprint-based comparison

Fingerprinting can be defined as a process which uses cleverly-designed hash function to map elements of an arbitrarily large domain $\mathcal{U}$ to much smaller objects (*i.e.*, the fingerprints). This general idea typically introduces two major benefits in large-scale applications: efficient *comparison* between (possibly large-size) elements and data *compression*. However, it is also affected by the problematics of hash functions: *collisions*, among them. For this reason it is of utmost importance to design a fingerprint algorithm which diminishes the chances of collisions (*i.e.*, false positives).

In this section we are now going to outline some interesting fingerprint-based techniques which also found applications in bioinformatics and, particularly, in sequence comparison.

Rabin-Karp fingerprinting. The idea of fingerprints has been adopted to solve the pattern-matching problems since the Rabin-Karp algorithm [86]: the rationale of the method is to replace computations on fixed-length strings with (much more efficient)

computations of *much shorter* strings (*i.e.,* the *fingerprints*). The key idea of the algorithm is to see, without loss of generality, a binary *string* $x = x_1 x_2 \dots x_n$ as the binary representation of the *number*

$$h(x) = \sum_{i=1}^{n} \left(x_i \cdot 2^{n-i}\right).$$

In this way, it is possible to define a collection of fingerprint (or hash) functions $h_m(x) = h(x) \mod m$. Searching the occurrences of a pattern p of length n within a string s of size $|s| \geq n$ can be done as follows: iteratively compare $h_m(p)$ with $h_m(s[i, i+n-1])$ $(i = 1, \dots, |s|-n+1)$ and, if such values are equal, perform a string comparison between that sub-string and p to check whether it is an actual occurrence or a false positive. The performance of such an algorithm depends on two main aspects: the choice of m and the efficient computation of h_m for successive sub-strings of s. For instance, setting m to a properly-chosen prime (*e.g.* a Mersenne number) diminishes the chances of hitting false positives. Moreover, choosing a value of m which is small enough to fit in a computer word allows to perform fingerprint comparisons in $\mathcal{O}(1)$. Finally, using also a rolling-hash scheme to compute $h_m(s[i, j])$ from $h_m(s[i-1, j-1])$ in constant time yields to an overall $\mathcal{O}(|s|)$ complexity.

An extension of such a fingerprint technique, which also account for inexact searches, has been introduced in rNA [182] (a software package which is currently known as ERNE). More precisely, it adopts a hybrid strategy which splits the pattern in t blocks while computing, for each of them, a hash value. The peculiarity of this method is the employment of *Hamming-aware* functions designed to restrict the search of matches on a Hamming sphere of radius $\mathcal{O}(k)$ "centered" at a hash value $h(p)$ of a pattern p, instead of the entire Hamming sphere of radius k. Formally such functions, as presented in [144], are defined as follows:

Definition 2.2 (Hamming-aware function)**.** Let w be the word size of the computer architecture. A hash function h is said to be Hamming-aware if there exist

- a set $\mathcal{Z}(k) \subseteq \Sigma^w$ such that $|\mathcal{Z}(k)| \in \mathcal{O}(c^k w^k)$ for some constant c, and
- a binary operation $\varphi : \Sigma^w \times \Sigma^w \to \Sigma^w$ computable in $\mathcal{O}(w)$ time,

such that if $p \in \Sigma^n$ then the following holds:

$$\{h(p') \mid p' \in \Sigma^n \ \wedge \ \delta_H(p, p') \leq k\} \quad \subseteq \quad \{\varphi(h(p), z) \mid z \in \mathcal{Z}(k)\}$$

The use of Hamming-aware hash functions hence allows to search the entire Hamming sphere of p efficiently. Specifically, ERNE computes *fingerprints* of blocks with at most k/t mismatches with respect to the original block. Finally, fingerprints are looked up within the hash index in order to seek for putative occurrences of the pattern.

MinHash. A different fingerprinting idea, employed recently in DNA sequence comparison, exploits a dimensionality reduction technique called MinHash [27] to create a more compact representation of sequences. Specifically, given a set $S = \{s_1, \dots, s_n\}$, m distinct hash functions $h^i : \Sigma^* \to \mathbb{N}$ (for $i = 1, \dots, m$), the fingerprint $\mathcal{H}_S$ of S is

a m-length vector such that $\mathcal{H}_S[i] = h^i_{min}(S)$, where $h^i_{min}(S)$ represents the minimal value returned by h^i applied to all the elements of S. Moreover, this technique allows to efficiently estimate the Jaccard's similarity between two sets.

In practice, a tool named MHAP [21] successfully applied this idea to detect overlaps among PacBio reads. More precisely, MHAP first considers, for each input sequence r, the set R consisting of all the k-length sub-strings of r. The fingerprints $\mathcal{H}_R$ are then computed and, using a sort-and-merge approach, MHAP then finds pairs of fingerprints sharing a minimum number of elements. Such identified pairs are then processed to check whether the two corresponding sequences overlap.

Bloom filters. A Bloom filter [23] is a space-efficient probabilistic data structure which can also be seen as a fingerprint. It is particularly useful to test whether an object of a domain $\mathcal{U}$ belongs to a set or, more generally, to efficiently compare sets. Formally, a bloom filter is a tuple $(B, m, h_1, \ldots, h_k)$, where B is a bit vector of size m and $h_i : \mathcal{U} \to \{1, \ldots, m\}$ are k distinct hash functions that independently and uniformly map each element of $\mathcal{U}$ to a random integer over the range $[1, m]$. Given a Bloom filter $\mathcal{B}$, the following two operations are defined:

- *Insertion.* an element $a \in \mathcal{U}$ is added to $\mathcal{B}$ by setting to 1 all the bits of B corresponding to the positions $h_i(a)$ for $i = 1, \ldots, k$.
- *Membership query.* an element $a \in \mathcal{U}$ belongs to $\mathcal{B}$ if and only if $B[h_i(a)] = 1$ for each $i \in \{1, \ldots, k\}$.

It is straightforward to see that checking whether an element belongs to the set does not produce false negatives (the removal of elements is not allowed). False positives, instead, are expected to occur with a probability p related to m, k and the number of inserted elements n. It is possible to prove that, given m and n, the value of k that minimizes such probability is

$$k \approx \frac{m}{n} \ln 2 \ .$$

Moreover, assuming k is set to this optimal value, it is possible to show that

$$m \approx -\frac{n \ln p}{(\ln 2)^2} \ .$$

Thus, given a false positive probability p and an estimation of the (to be) inserted elements n, suitable values of k and m can be computed.

Due to the non-optimal space occupancy, a non-constant time complexity [135], and the absence of a delete operation, several variants have been proposed in literature. However, to the best of our knowledge, the use of Bloom filters in assembly-related methods has been mainly restricted to its classical implementation. Applications in bioinformatics involve distributed read alignment [117], sequence classification [172], k-mer counting [114], error correction [71,166], and sequence-graph compression [38,136].

3

Theoretical *de novo* assembly

The *de novo* genome assembly problem (DGAP) is the task of reconstructing the sequence of a genome, starting from a (usually large) set of randomly-extracted short sequences called *reads*. Many efforts have been done in order to address the problem of reconstructing a genome from a theoretical perspective [112, 126, 137].

In 1984, Peltola *et al.* presented the first application of the NP-hard *shortest common super-string problem* (SCSP) to solve DGAP [137], suggesting a solution based on the Overlap-Layout-Consensus (OLC) paradigm. The NP-hardness of the SCSP formulation led also to develop several approximation algorithms.

However, reducing DGAP to SCSP is biologically inaccurate and, for this reason, the assembly problem has been subsequently modeled as finding paths in graphs (built from the reads) which better explain the input data. The main benefit of this approach is the ability to reduce DGAP to known problems and, therefore, properly study the computational complexity along with the use of efficient exact/approximate algorithms. In 1995, Myers addressed SCSP problems by proposing a model based on the *overlap graph* in which vertices correspond to reads and (bi-directed) edges properly encode overlaps. DGAP was then reduced to the NP-complete problem of finding a Hamiltonian path. In the same year, the use of *de Bruijn graphs* (dBG) for genome assembly was also introduced – and later expanded by Pevzner [140] to account for the double-stranded nature of DNA. This structure can be seen as a lossy representation of the overlap graph and, from a theoretical perspective, it is the other face of the same coin. As a matter of fact, in 2007, Medvedev proved the NP-hardness of finding a coherent (and optimal) assembly path both in overlap/string and de Bruijn graphs [112]. Due to the intractability of both models and the presence of sequencing errors in the input reads – which further complicate the problem – assemblers have been mostly designed to exploit different heuristics in order to directly provide an approximate solution while also using moderate amounts of computational resources.

Finally, in 2008, Narzisi and Mishra developed an approach based on combinatorial optimization techniques in order to overcome the inaccuracy of heuristic methods [127]. Their idea however is limited to small-sized genomes and cannot compete with other algorithms on large-scale data-sets.

3.1 Definitions

Let $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ be the DNA alphabet and $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$ be the set of all non-empty strings over it. The *complement* of each character is defined as follows: $\overline{\mathtt{A}} = \mathtt{T}$, $\overline{\mathtt{T}} = \mathtt{A}$, $\overline{\mathtt{C}} = \mathtt{G}$, and $\overline{\mathtt{G}} = \mathtt{C}$. Given a generic string $s \in \Sigma^+$ we denote its length by $|s|$ and the ith character as $s[i]$. Given two indices i, j such that $1 \leq i \leq j \leq |s|$, we define the *substring* of S from the ith to the jth character (included) as $s[i, j] = s[i]s[i + 1] \cdots s[j]$. The *reverse complement* $\overleftarrow{\overline{s}}$ is defined by reversing s and substituting each character with its complement. Formally, $\overleftarrow{\overline{s}}[i] = \overline{s[|s| - i + 1]}$, for $i \in \{1, \ldots, |s|\}$.

In the following sections we will often refer to particular strings of fixed length. More precisely, a string consisting of k characters is also called k-mer. Moreover, given a sequence s, we define its ith k-mer by $k_i^s = s[i, i+k-1]$, $i \in \{1, \ldots, |s|-k+1\}$. We may just write k_i when s is clear from the context. For a sequence collection $\mathcal{S} = \{s_1, \ldots, s_n\}$, we also define the set of all its k-mers (the *spectrum*) as follows:

$$\mathcal{K}^k(\mathcal{S}) = \{\ k_i^s \mid s \in \mathcal{S} \ \wedge\ |s| \geq k \ \wedge\ i \in [1, |s| - k + 1]\ \}\,.$$

Definition 3.1 (Overlap)**.** Let $a, b \in \Sigma^+$ be two strings such that $|a|, |b| > 1$. We say that *a overlaps b* if and only if there exist a string $z \in \Sigma^+$ such that:

- $1 \leq |z| < \min(|a|, |b|)$;
- $a[z_a, |a|] = z = b[1, z_b]$, where $z_a = |a| - |z| + 1$ and $z_b = |z|$. In other words, z is suffix of a and a prefix of b.

We also define $ol(a, b) = |z'|$, where z' is the longest string for which the aforementioned constraints are fulfilled ($ol(a, b) = 0$ if such a string does not exist).

Notice that in the above definition we require a match between a *proper* suffix/prefix and none of the two sequences can be fully contained into the other one. In case a is a substring of b we say that *a is contained* in b or, equivalently, *b contains a*.

Definition 3.2 (Layout)**.** Let $\mathcal{R} = \{r_1, \ldots, r_n\}$ be a set of reads such that, for each $i \neq j$, r_i is not contained in r_j. A *layout* is defined as a permutation $\pi : [1, n] \to [1, n]$ of the elements of $\mathcal{R}$ and its weight $w(\pi)$ is defined as

$$w(\pi) = \sum_{i=1}^{n-1} ol\left(r_{\pi(i)}, r_{\pi(i+1)}\right).$$

Intuitively, a read layout depicts a way to combine (*i.e.*, assemble) reads in order to possibly reconstruct the genome they were extracted from. More precisely, the reconstruction is obtained from a layout π by the concatenation of the reads – following π's order – and keeping only one copy of the "overlap string". We now define the *de novo* genome assembly problem as follows:

Problem 3.3 (*De novo* Genome Assembly Problem (DGAP))**.** Let $\mathcal{R} = \{r_1, \ldots, r_n\}$ be a set of input sequences (*i.e.*, the reads) such that, for each $i \in \{1, \ldots, n\}$, $r_i \in \Sigma^+$ and r_i is a substring of an *unknown* string $g \in \Sigma^+$ (*i.e.*, the genome). The problem is finding a layout π_g of $\mathcal{R}$ that better explains the observed set of reads $\mathcal{R}$ (*i.e.*, that better approximate the unknown genomic sequence).

Notice that the absence of containment relations in a layout of $\mathcal{R}$ can be assumed without any loss of generality: contained reads can be initially omitted and "mapped" back when the layout has been built.

3.2 The Shortest Common Super-string Problem

Based on a parsimony assumption, at the beginning, genome assembly has been approximated as finding the *shortest common super-string* (SCS) containing each input sequence and it can be formally defined as follows.

Problem 3.4 (Shortest Common Super-string Problem (SCSP))**.** Let $\mathcal{R} = \{r_1, \ldots, r_n\}$ be a set of input reads, find the *shortest* string s such that, for each $i \in \{1, \ldots, n\}$, r_i is a substring of s.

This problem can be seen as finding a *Hamiltonian path* of maximum weight in a directed graph where edges represent overlap lengths between read pairs [176] or, equivalently, finding a maximum-weight layout. The NP-hardness proof of SCSP [61] led to study several polynomial-time approximation algorithms. Among those, a particular attention has been put on a simple greedy approach: iteratively combine two strings in $\mathcal{R}$ which have the longest overlap amongst all pairs until either there is only one string or all pairs have an empty overlap. Tarhio and Ukkonen studied its performance in terms of the "compression" achieved by a super-string s and defined as $\Sigma|r_i| - s$. In particular they showed the compression achievable by the greedy algorithm is at least half of the one of an optimal solution and conjectured a 2 approximation factor [176].

Even though minimizing output length (using different constraints) is a very popular choice, from a biological perspective, an accurate approximation of the genome is not guaranteed. In particular, for SCSP we can observe that:

i. the double stranded nature of DNA is not taken into account;

ii. input sequences are assumed to be exempt from errors;

iii. repeated sub-sequences are compressed in a single one.

The last point is probably the most significant one, as there is no biological reason for collapsing repeats. In fact, doing so would likely yield wrong assemblies, as it is not uncommon that repeated sub-sequences are found in multiple copies at different points of a genome.

3.3 Overlap Graphs

An overlap graph is typically defined as a graph where vertices correspond to reads and edges depict overlap relations. In order to take care of the two DNA strands, Kececioglu and Myers introduced a very intuitive representation of overlaps in a graph [88]. First, notice that there are only four admissible overlaps between two sequences a and b:

i. a suffix of a matches a prefix of b (a overlaps b);

ii. a prefix of a matches a suffix of b (b overlaps a);

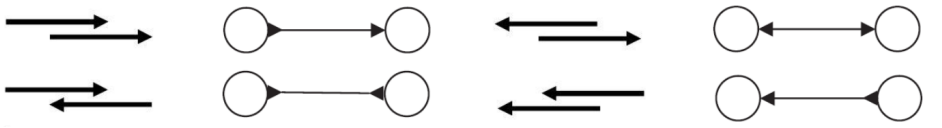


Figure 3.1: Bi-directed edges defining overlaps.

iii. a suffix of a matches a prefix of $\overleftarrow{b}$ (a overlaps $\overleftarrow{b}$);

iv. a prefix of a matches a suffix of $\overleftarrow{b}$ ($\overleftarrow{b}$ overlaps a);

We denote any of the previous cases with $a \rightleftharpoons b$. Then, an overlap can be modeled as a single *bi-directed* edge by adding an oriented arrow at both the endpoints which also tells us whether to use the actual read or the reverse complement. More specifically, consider two reads r_i, r_j such that $r_i \rightleftharpoons r_j$ and let e be an edge between the corresponding vertices. The direction of the arrow on r_i's side of e is depicted in Figure 3.1 and is defined as follows:

- if the overlap involves r_i's suffix (*i.e.*, cases i. and iii.), it points away from r_i;
- if the overlap involves r_i's prefix (*i.e.*, cases ii. and iv.), it points towards r_i .

Entering a vertex corresponding to r_i with an arrow pointing towards it tells us to consider the read r_i. If the arrow points away, instead, $\overleftarrow{r_i}$ must be used. The same reasoning also applies to r_j.

Hereafter, we might also use $r_i \overset{e}{\rightleftharpoons} r_j$ to denote a bi-directed edge e between two vertices r_i and r_j.

Definition 3.5 (Overlap graph)**.** Let $\mathcal{R} = \{r_1, \ldots, r_n\}$ be a set of reads such that, for each $i \neq j$, r_i is not contained in r_j and let $k \in \mathbb{N}_{>0}$ be an overlap threshold. The *overlap graph* is a bi-directed weighted graph $OG^k(\mathcal{R}) = (V, E, w)$ such that:

- $V = \mathcal{R}$;
- $E = \{\ e \mid e = (r_i, r_j) \ \wedge\ r_i \rightleftharpoons r_j \ \wedge\ ol(e) \geq k\ \}$, where $ol(e)$ denotes the length of the specific overlap modeled by e;
- $w(e) = |r_j| - ol(e)$, where $r_i \overset{e}{\rightleftharpoons} r_j$.

It is worth noticing that the weight of and edge e is defined with respect to the direction e is visited and that more than one type of overlaps – and hence more than one edge – might arise between two reads.

Due to the non-canonical representation of edges, we still have to define how such a bi-directed graph needs to be visited. Formally, we introduce the notion of *read-coherent* path (or walk).

Definition 3.6 (Read-coherent path)**.** Let $\mathcal{R} = \{r_1, \ldots, r_n\}$ and $OG^k(\mathcal{R}) = (V, E, w)$ be an overlap graph. A *read-coherent* path is defined as a sequence σ of m vertices and $m - 1$ edges such that:

- $\sigma = r_{i_1} \overset{e_1}{\rightleftharpoons} r_{i_2} \overset{e_2}{\rightleftharpoons} r_{i_3} \overset{e_3}{\rightleftharpoons} \cdots \overset{e_{m-2}}{\rightleftharpoons} r_{i_{m-1}} \overset{e_{m-1}}{\rightleftharpoons} r_{i_m}$;
- for each $z \in \{2, \ldots, m\}$, the arrows of edges e_z and e_{z-1} on r_{i_z}'s endpoint have opposite directions.

The weight of σ is defined as $w(\sigma) = \sum_{i=1}^{m-1} w(e_i)$.

The problem of seeking a layout from $OG^k(\mathcal{R})$ can be modeled as a special case of the *traveling salesman problem* (TSP).

Problem 3.7 (Traveling Salesman Problem (TSP))**.** Let $G = (V, E, w)$ be a weighted graph (not necessarily complete). Find a minimum-weight Hamiltonian path, that is, a path σ which visits every vertex $v \in V$ *exactly once* and whose weight $w(\sigma)$ is minimum.

In the overlap graph framework, indeed, we need to seek for a minimum-weight *read-coherent* Hamiltonian path.

3.3.1 String Graphs

As Myers pointed out in [122], the size of $OG^k(\mathcal{R})$ can be dramatically reduced by removing redundant nodes and edges, while preserving all the information encoded in the input data-set.

More precisely, in addition to contained reads, the idea is to remove transitively inferred edges. As a matter of fact, included reads just provide redundant information for the assembly process and they could easily be excluded from the graph construction (they are however still used to estimate the copy numbers of repeated substrings). Given three reads/vertices $x, y, z \in V$, an edge $e \in E$ such that $x \overset{e}{\rightleftharpoons} z$ can be transitively inferred if there exist a read-coherent path $x \overset{e_1}{\rightleftharpoons} y \overset{e_2}{\rightleftharpoons} z$ with the same arrow orientations of e_1 and e_2 on x and z endpoints, respectively. In simple terms, $x \rightleftharpoons z$ can be deduced from the overlaps $x \rightleftharpoons y$ and $y \rightleftharpoons z$. For this reason e is superfluous and with its removal we do not lose any information.

Another common operation (yet not essential) – performed after the transitive reduction – is a sort of *compression* and consists in replacing unambiguous paths with singleton vertices/edges which account for the corresponding (again, unambiguous) path strings. It is worth noticing that this compression greatly simplifies a string graph and also retains its former topology. These merged sequences, in fact, possibly represent (maximal) DNA fragments unequivocally resolved by the reads. This operation is also called *unitigging* and merged strings are also referred to as *unitigs* [123].

Definition 3.8 (String graph)**.** Let $\mathcal{R} = \{r_1, \ldots, r_n\}$ be a set of reads such that, for each $i \neq j$, r_i is not contained in r_j and let $k \in \mathbb{N}_{>0}$ be an overlap threshold. The *string graph* is a bi-directed weighted graph $SG^k(\mathcal{R})$ obtained from $OG^k(\mathcal{R})$ by removing transitive edges.

A *string graph* can be computed in polynomial time [122]. However, the mere presence of non-redundant edges force us to consider a different formulation of DGAP in $SG^k(\mathcal{R})$. The assembly problem can then be formulated as finding a minimum-weight generalized Hamiltonian path in $SG^k(\mathcal{R})$. Nagarajan and Pop provided a reduction from SCSP, hence demonstrating the NP-hardness [126].

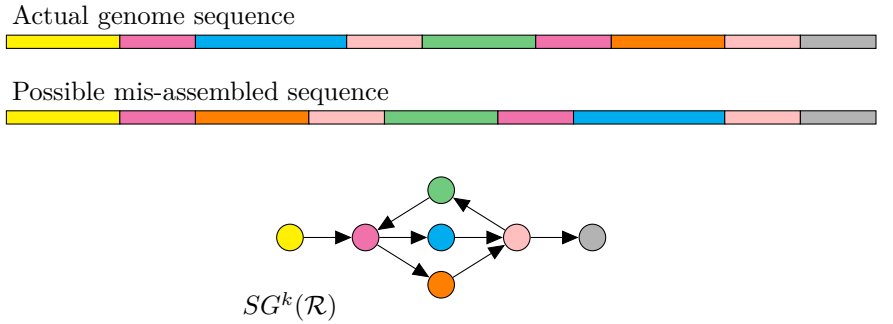


Figure 3.2: Example of possible mis-assemblies in the string graph framework. Nodes in $SG^k(\mathcal{R})$ represent unitigs. A Hamiltonian path does not necessarily correspond to a biologically accurate assembly.

Problem 3.9 (Generalized Hamiltonian Path)**.** Let $G = (V, E)$ be a graph (not necessarily complete). Find a path σ which visits every vertex $v \in V$ *at least once*.

Finding a generalized Hamiltonian path on $SG^k(\mathcal{R})$ (or, equivalently, a Hamiltonian path on $OG^k(\mathcal{R})$) is indeed a better approximation of DGAP with respect to solving SCSP. Repeats, however, might still get collapsed or lead to wrong genome reconstructions (see Figure 3.2).

Exploiting included reads, Myers also proposed a statistical method to estimate the number of times an edge should be accounted. More precisely, edge traversals are first partitioned in three categories: exactly once, at least once, and optional (*i.e.*, any number of times). Solving a (polynomial) minimum-cost network flow problem it is then possible to estimate a putative traversal count. This lead to model DGAP as finding a minimum-weight path such that edge constraints are fulfilled (again, an NP-hard problem [112]).

A detailed overview of the algorithms based on this framework will be presented in Section 4.1.2.

3.4 de Bruijn Graphs

Since its introduction, the Overlap-Layout-Consensus paradigm has been the classical approach for fragment assembly [64, 88, 169]. The main drawback of such a method is that finding pairwise overlaps among sequences is computationally expensive and both the SCSP and the overlap graph formulation have been proven to be NP-hard problems. This led to the introduction of many (error-prone) heuristics in order to seek for computational efficiency.

An alternative to the OLC paradigm is given by the de Bruijn graph [47] (dBG) framework as it models the genome assembly problem into finding an Eulerian path in a graph. de Bruijn graphs were first exploited in this context in 1995 by Idury and Waterman [83] which defined the so-called *spectrum* graph (*i.e.,* a graph built from k-mers). More precisely, they modeled fragment assembly as a *sequencing-by-hybridization*

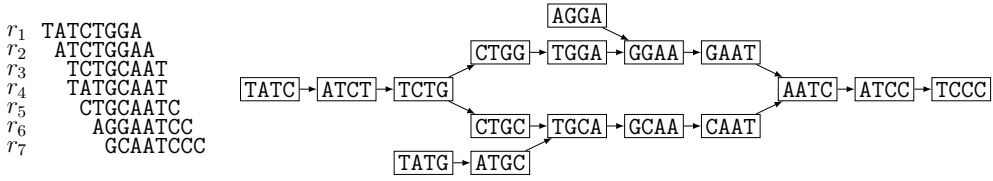


Figure 3.3: de Bruijn graph $dBG^k(\mathcal{R})$ example, where $\mathcal{R} = \{r_1, \ldots, r_7\}$ and $k = 5$.

problem by replacing every read with the set of all its k-mers. Formally a de Bruijn graph can be defined as follows:

Definition 3.10 (de Bruijn graph)**.** Let $\mathcal{R} = \{r_1, \ldots, r_n\}$ be a set of reads and $k \geq 2$ an integer constant. The *de Bruijn graph* is a directed graph $dBG^k(\mathcal{R}) = (V, E)$ such that:

- $V = \mathcal{K}^{k-1}(\mathcal{R})$;
- $E = \{\ (v_i, v_j) \mid v_i, v_j \in V\ \wedge\ v_i[2, k-1] = v_j[1, k-2]\ \}$.

In other words, $dBG^k(\mathcal{R})$ is build such that nodes are distinct $(k-1)$-mers and edges correspond to distinct k-mers. More precisely, an edge is put between a pair of $(k-1)$-mers which have the same first and last $k-1$ bases, respectively (see Figure 3.3). For instance, an edge $(v_i, v_j) \in E$ represents the k-mer $v_i \cdot v_j[k-1]$.

Breaking reads into shorter fragments might seem counterproductive, because the long range information provided is lost. However, choosing a high enough value of k limits this issue and brings to the table the computational advantages that allowed dBGs to cope with the large amount of reads provided by NGS technologies, as the computationally expensive overlap detection among reads can be avoided. If we do not take into account errors, the genome can hence be reconstructed by properly choosing a path which visits every edge (k-mer) exactly once (*i.e.*, an Eulerian path).

The main drawback of this framework is that paths describing an assembly might be unsupported by any read, in contrast to the overlap/string graph in which a path consistently represent an assembly. As a matter of fact, a de Bruijn graph could contain several Eulerian paths which do not correspond to a correct genome reconstruction. In order to cope with this issue, Pevzner *et al.* [140] proposed to model the assembly problem as finding an Eulerian *super-walk* that actually contains each *read-path* (*i.e.*, a path that "spells" an actual read). Unfortunately, this more precise formalization has been proven to be NP-hard [112].

3.5 Constraint Optimization Problem

Despite being both the OLC and the dBG frameworks well-defined theoretical formulations, they do not model properly the *de novo* genome assembly problem, as a solution might still not be biologically correct. The NP-hardness also led to the development of many heuristic approaches which usually yield approximate results.

A different theoretical framework has been proposed by Narzisi *et al.* [127] and it aims at solving DGAP while being as much biologically consistent as possible. More

precisely, they acknowledge the difficulty of the problem and directly model genome assembly as a constrained optimization problem (COP). The idea is to employ a search which could potentially consider exhaustively all possible read layouts, while limiting the (exponential) solution space with a branch and bound schema which does not explore further unlikely layouts.

Problem 3.11 (Constrained DGAP)**.** Let $\mathcal{R} = \{r_1, \ldots, r_n\}$ be a set of reads and let ε be an error-rate threshold. Find an assembly g' based on a read layout π such that:

- $r_{\pi(i)}$ can be mapped on g' with at most $\varepsilon \cdot |r_{\pi(i)}|$ differences for each $i \in \{1, \ldots, n\}$;
- the overall overlap score given by all reads overlapping in π is optimized;
- the insert size of paired reads is consistent;
- the observed distribution of restriction enzyme sites is consistent with the distribution of a provided experimental *optical map*.

The method is then based on the optimization of well-defined score functions, each one defining a feature that we expect from a good assembly.

II

Real World Genome Assembly

4

Practical Assembly Strategies

The shotgun process allows to obtain reads uniformly distributed along a target molecule. Whole-genome shotgun (WGS) sequencing, in particular, refers to sampling reads from genome's chromosomes. Hence, WGS assembly consists in the reconstruction of sequences up to the chromosome length. Given the possibility to sequence a large amount of reads in parallel, this approach has been the most used strategy in many sequencing projects and, due to the computational complexity results outlined in Chapter 3, assembly tools employed different heuristics in order to approximately solve the difficult task of genome assembly. The NP-hardness of each read-coherent formalization, however, should not frighten us. As a matter of fact, approximate methods achieved satisfactory results in many projects and this might indicate that the theoretical difficulty could be effectively and efficiently overcome in practice.

In an ideal world, assemblers would output a number of sequences fully representing chromosomes. In the "real world", though, *de novo* assembly methods need to take into account several factors which make the problem even more difficult than the one formalized in Chapter 3. First, reads might contain sequencing errors (or parts of clone/vector sequences) and this often requires an error-correction phase. Second, genomes may contain repeated sub-sequences as well as several haplotypes. These biological traits, along with uncorrected errors, yield peculiar – and more complex – graph structures and demand the employment of *ad hoc* solutions in order to take them into account. For these reasons, an assembly typically consists of a (large) collection of contiguous sequences named *contigs*. The core part of the assembly process is usually carried out with greedy or graph-based approaches (such as those presented in Sections 3.3 and 3.4). Specifically, tools are often classified into four main categories: Greedy, Overlap-Layout-Consensus (OLC), de Bruijn Graph (dBG), and Branch-and-Bound (B&B). In the final stages of the assembly process, these fragments might be further ordered, oriented, and joined into longer (*gapped*) sequences called *scaffolds*, exploiting, for instance, paired-read libraries. Many tools are available for this last phase also known as *scaffolding*. However, they will not be covered in this dissertation and we will only focus on the core step of the assembly process. The interested reader can find more details on this subject in [24, 46, 51, 62, 153].

4.1 Assembly strategies

Through the evolution of sequencing technologies, algorithms have been specifically designed to accommodate particular read lengths and error rates. For instance, the relatively low coverage of long good-quality reads (proper of Sanger technologies) paved the way to greedy and OLC-based assemblers.

When next-generation technologies appeared, however, all canonical assembly algorithms failed to adapt. As a consequence, assemblers needed to be re-designed and developed from scratch in order to handle the new features of NGS data (*i.e., very high* coverage of *very short* reads). In this new scenario, the de Bruijn graph framework definitely proved to be more suited than OLC and greedy methods.

Recently, the long and error-rich TGS reads brought new challenges for the assembly process and, specifically, for the error correction and overlap detection phases. In order to make use of such reads, hybrid pipelines were introduced in order to correct reads using NGS technologies. Effective algorithms were devised in order to make use of the OLC paradigm (set aside for NGS projects, yet more suited for long sequences) on such error-prone data-sets.

In the following sections we are now going to outline the most common assemblers developed for each of the aforementioned scenarios.

4.1.1 Greedy

In early genome sequencing projects, the number of reads produced rarely exceeded several thousands and this led to the implementation of very simple heuristics: pick a sequence and extend it until no further extension can be done. The sequence considered at a certain point (which can be either a read or a contig) is extended with the highest-score overlapping read.

It is immediately clear that the performance of greedy methods strictly depends on the efficient computation of all (or almost all) overlaps between the chosen sequence and the available (*i.e.,* unused) reads. This task had been usually carried out using, for example, hash tables. In general, for the sake of computational efficiency, a common approach for this phase is to limit the search of overlaps to only the exact ones above a minimum length threshold.

It is evident that such algorithms are inherently different from graph-theoretical approaches as they are based on *local* choices. However, they can be seen as a visit of an implicit – and highly pruned – overlap graph where only few high-scoring edges are kept [116]. It is also clear that this approach works better with limited numbers of reads and low-complexity genomes, as the abundance of very similar reads may introduce a certain level of ambiguity during sequence extension which hinders the assembly process. The main limitation of such heuristic is its lacking of a *global* perspective on the problem. As a matter of fact, genomic repeats (even short ones) constitutes a major issue: assemblers usually fail to reconstruct them on complex genomes and they likely introduce mis-assemblies. For this reason, the greedy method has been mostly used with bacterial genomes.

Well known assemblers tailored for Sanger datasets are TIGR [173], CAP3 [78], and Phusion [118].

TIGR. The TIGR Assembler was probably the earliest greedy assembler developed and it was proposed by Sutton *et al.* in 1995. Its hallmark is the ability to detect potential repeats by determining reads having unusually large numbers of overlaps than it is expected (assuming a random distribution of sequencing). These putative repeats are handled at the end of the process using stricter merging criteria.

CAP3. It is the third version of the CAP assembler and it introduces a certain number of improvements with respect to other greedy implementations: it filters reads and use a dynamic programming algorithm to compute maximal-scoring alignments only between reads which are likely to overlap. Specific parameters are set in order to separate sequencing errors from divergent copies of repeated sequences. CAP3, in addition, clips low-quality regions of reads and makes use of read-pair information.

Phusion. This long-read assembler was used to assemble the 2.6-billion-base repeat-rich mouse genome. The algorithm consists of three main phases. First, reads are corrected and those sharing low-copy-number words are clustered. Second, clusters are assembled independently using PHRAP [67] (another greedy assembler which makes use of Phred quality scores). Finally, overlapping contigs are joined with the help of shared reads and read-pair information.

The seed-and-extend framework

With the advent of NGS technologies, the number of reads in NGS projects increased by one (or more) orders of magnitude. For this reason greedy assemblers had to be redesigned in order to efficiently seek for overlapping reads and possibly account for the information coming from paired reads. The greedy approach for this scenario has been also renamed *seed-and-extend* but the only difference with respect to the general algorithm used for Sanger sequencing is in the data structures employed. More in detail, seed-and-extend algorithms often use a prefix-tree to efficiently store fixed-length prefixes and to seek for possible extensions.

Although mitigated by the use of paired reads, the seed-and-extend framework still fails to cope with error-rich datasets. However, this technique can be used as a pre-processing step for NGS datasets in order to obtain Sanger-like reads that can be used as input for an OLC assembler.

Examples of assemblers in this category are SSAKE [187], VCAKE [85], SHORTY [75], PE-Assembler [15], and GapFiller [125].

SSAKE. Released in 2007, it is the earliest short-read greedy assembler thought for NGS data. The algorithm of SSAKE is based on progressively seek for the longest possible overlap between two sequences through the use of a prefix tree. More precisely, unique sequences are stored in a hash table along with the number of occurrences in the input data-set. A prefix tree is then used to organize them and their reverse complements. Moreover, reads are processed in decreasing order with respect to their frequency to prevent the extension of reads containing errors. In order to minimize the number of mis-assembly, read extension can also be stopped when ambiguities occur. This, however, might lead to shorter contigs. SSAKE's idea has been later improved

by SHARCGS [50] which basically adopt the same strategy but adds several pre- and post-processing steps.

VCAKE. The Verified Consensus Assembly by K-mer Extension (VCAKE) algorithm is based on a modification of the simple k-mer extension proper of previous tools such as SSAKE and SHARCGS. The peculiarity of this algorithm is to work with non-perfect overlaps in order to overcome sequencing errors (and handle polyploid organisms). This is done by exploiting high-depth coverage and extending seeds one base at a time (using the most commonly represented base from all overlapping reads). It is worth to mention that VCAKE, as well as other seed-and-extend tools, has been also employed successfully in hybrid pipelines in order to combine different sequencing technologies within a single assembly project [149].

SHORTY. To the best of our knowledge, SHORTY is the first assembler that have been applied on *real* data coming from the emerging SOLiD sequencing technology and designed to make use of paired-end reads. In particular, SHORTY expects in input a deep coverage of short reads and a small collection of 300–500 bp starting *seeds* (which can be build using another assembly on the input data-set). As a pre-processing step, good-quality paired reads are stored in a prefix tree along with their reverse complement. Then, a seed is chosen and a set of neighboring paired reads (*i.e.,* such that one of the two reads in a pair maps against the seed) is built. This set can be exploited in order to extend the chosen sequence into longer contigs. This process of seed extension is repeated as long as it is possible. The ultimate phase of SHORTY is to use again the pairing information coming from the reads to build scaffolds.

PE-assembler. This is probably the latest seed-and-extend tool developed for whole-genome assembly. According to its authors, PE-Assembler is capable of handling large datasets and produces highly contiguous and accurate assemblies within reasonable time. Differently to other contemporary approaches, it does not represent the genome as a graph but, instead, it is based on a simple extension procedure and for this reason it is very similar to SSAKE, VCAKE, and SHARCGS. However, the improvement on such older tools is made by an extensive use of paired reads and parallelization. Extension ambiguities are handled using a multiple-path approach that takes into account sequence coverage, evidence from multiple paired-read libraries, and other features such as the insert size distribution of paired-end reads.

GapFiller. As opposed to the other assembly methods described up to this point, this is a *local de novo* assembly tool and it is thought to reconstruct the inserts of a paired read library. It is based on a seed-and-extend scheme in which every read is selected as seed and iteratively extended until its mate is possibly found. The extension is performed along with the computation of a consensus based on overlapping reads. The search of overlaps is carried out in a very efficient manner thanks to a hash function which guarantees a low false positive rate [182]. Moreover, the extension phase employs peculiar clustering and trimming heuristics in order to guarantee assembly correctness.

Despite being a local assembler, the distinctive hallmark of GapFiller is to produce long high-quality sequences which could be used as input for a whole-genome assembler

to enhance both contiguity and correctness of the assembly. This method can be also exploited for structural variation reconstruction.

4.1.2 Overlap-Layout-Consensus

Assemblers based on the overlap/string graph (recall the definitions in 3.3) are usually referred to as implementing the Overlap-Layout-Consensus (OLC) approach. As the name suggests, the approach consists in three main phases.

First, pairwise read overlaps are computed in order to determine graph's edges. This is also the most computationally demanding phase of the OLC approach and, for efficiency issues, tools usually compute the k-mer spectrum in order to index reads in a hash table and to seek overlaps exclusively amongst reads sharing a certain number of k-mers. It is worth to mention that for repetitive genomes this speed-up might not be of great help since the number of reads sharing the same k-mer could be quite large.

Second, a consistent layout is sought, usually among a collection of paths in the graph which satisfy certain properties. If available, the assembler can use the paired read information to accomplish this task.

The third and last phase consists in computing multiple sequence alignments in order to produce a precise layout and the consensus sequence. Again, this step is performed using approximate methods, since no efficient algorithm is known for computing the optimal solution [54].

The performance of an OLC assembler is sensitive to three parameters: the k-mer size, the minimum overlap length and sequence identity. Higher values of such parameters likely lead to correct and fragmented assemblies, while lower values diminish the computational costs but introduce the concrete risk of introducing more mis-assemblies.

The OLC approach has demonstrated its capabilities in many Sanger-based sequencing projects: ARACHNE [18] and PCAP [79] are two well known OLC assemblers.

Despite being the gold-standard for many years, OLC assemblers were not able to cope with the high abundance and the small read length of NGS technologies (which made this approach computationally demanding). Still, many assemblers based on this framework appeared, thanks also to the development of clever heuristics and data structures. Well known OLC assemblers for NGS reads are Minimus [165], Edena [73], CABOG [115], and SGA [161].

ARACHNE. The ARACHNE assembler is a software designed to analyze paired reads obtained from Sanger sequencing (although it can also be used with single-ended reads).

As a pre-processing step, ARACHNE processes reads in order to trim terminal regions of poor quality and to discard reads containing mostly low-quality bases.

Subsequently, ARACHNE detects and aligns putative pairs of overlapping reads. Some of them might be false positives (due to repeats in the genome) and they will be eliminated in a subsequent step. Rather than comparing all read pairs, overlap detection is carried out using a *sort and extend* strategy. More precisely, each k-mer is stored in a table along with the read identifier and the position it occurs. This table is then sorted so that identical k-mers appear consecutively and very high frequency k-mers are removed. In order to detect and correct sequencing errors, ARACHNE

computes multiple alignments among overlapping reads and, once a good alignment set is determined, the tool exploits paired-read information to link sequences and to create contigs.

In the absence of repeats, producing the correct layout would be easy: any two reads with significant overlap likely belong to the same genome *locus*. Nevertheless, false overlaps may arise from different repeated sequences. For this reason, ARACHNE identifies potential repeat boundaries and avoids assembling contigs across them (making it overly conservative).

PCAP. The PCAP assembler (Parallel Contig Assembly Program) is a tool aimed at processing efficiently millions of Sanger reads and addressing the issues of large WGS projects. Unlike Arachne, PCAP allows the computation of overlaps to be performed in parallel on multiple processors. More precisely, the entire dataset $\mathcal{R}$ of reads is partitioned into m subsets of similar sizes, where m is the number of available processors. Because of the huge size of the dataset, $\mathcal{R}$ is stored on the secondary memory. Instead, every partition and its data structures are kept in main memory.

PCAP's algorithm can be divided into three main phases: first, repetitive regions are identified and overlaps are computed; second, overlaps are evaluated to construct unitigs; third, contigs/scaffolds are built.

In order to identify repeats among the reads, a subset $\mathcal{R}_i$ is first compared with itself. Additional repeats are then discovered comparing (in parallel) the whole dataset $\mathcal{R}$ against each single $\mathcal{R}_i$. At the end, overlaps between reads in $\mathcal{R}$ and unique read regions in the subset are computed using a look-up table.

The construction of contigs resembles ARACHNE's algorithm. In particular, depth of coverage is exploited in order to score each overlap. In this way poor end regions (of reads) are identified and removed. Reads are then assembled into contigs based on unique overlaps. Finally, contigs are corrected and linked into scaffolds and a multiple alignment is constructed in order to derive a consensus for each contig.

Minimus. The Minimus assembler was built as a modular pipeline and with the goal to be easily (re)usable. It has been released as a component of AMOS [1]: an open-source package that provides a collection of tools and libraries for the development of assembly and analysis pipelines. In particular, Minimus consists of the combination of three AMOS modules, following the traditional OLC paradigm:

1. `hash-overlap`: a sequence overlapper that uses minimizers [150] to increase speed and decrease memory usage.

2. `tigger`: a unitigger, *i.e.* a tool which clusters reads that can be unambiguously assembled, based on the algorithms developed by Myers in [122].

3. `make-consensus`: a progressive multiple alignment tool that refines the read layout generated by `tigger` in order to generate a precise multiple alignment of reads.

Edena. The Exact *de novo* assembler (Edena) [73] is a tool conceived to process millions of reads produced by the Illumina technology.

In addition to the classical OLC approach, Edena includes two features which improve the assembly of very short sequences: exact matching and detection of spurious reads. The first one has been used for two main reasons: first, avoiding spurious overlaps due to sequencing errors; second, exact matching is considerably faster than approximate matching (using an appropriate index, overlaps between millions of short reads can be computed using reasonable amounts of computational resources).

The approach followed by Edena can be summarized in four main phases. First, the input data set is processed to remove redundant information and reads are indexed within a prefix tree. Second, all overlaps of a predetermined minimum size are computed using a suffix array to build the overlap graph G. Third, since G is likely to contain a large amount of branching paths – hindering the construction of long contigs – the overlap graph is simplified by removing transitive and spurious edges, and by resolving bubbles (*i.e.,* two paths starting and ending on the same nodes). Finally, all contigs above a certain length threshold which are unambiguously represented in the graph are provided as output.

CABOG. The Celera Assembler with the Best Overlap Graph (CABOG) is a modified version of the former Celera Assembler [123] and its pipeline was developed in order to combine reads coming from Sanger and 454 technologies.

CABOG uses seeds (*i.e.,* k-mers) to compute exact-matches in order to detect possibly overlapping reads. Seed matches are found within compressed sequences where consecutive instances of the same nucleotide are collapsed into a single one. This compression allows to better deal with 454 reads. More in detail, CABOG counts the number of instances of each distinct k-mer observed in the compressed input sequence and discards those appearing too frequently or uniquely. Then, overlaps are computed among read pairs which share enough k-mers (which are also used to *anchor* the alignments).

Using computed alignments, CABOG is then able to build an overlap graph G. More precisely, G is reduced to the Best Overlapping Graph (BOG) by keeping, for each read, only the best (*i.e.,* the longest) overlap and by discarding contained reads. All cycles are resolved by deleting a randomly chosen edge, making the graph acyclic. CABOG then sorts reads by score (defined as the number of reachable reads) and, starting from the highest scoring ones, it follows the paths in the BOG to construct unitigs. Paired-read constraints are also exploited during the unitig construction.

Finally the resulting set of sequences is provided as input to the Celera Assembler to build contigs, scaffolds, and to perform the consensus step.

SGA. The main issue of OLC assemblers is their inability to cope with large NGS datasets due to the very large amount of memory typically required. For this reason, tools like Edena cannot be used to assemble large eukaryotic genomes such as plants or mammalians. Other assemblers, instead, have been focused to take advantage of the longer (yet more expensive) reads of 454 platforms. Since the overlap detection is usually carried out by the employment of read indices, large read numbers demands clever and space-efficient data structures.

As a matter of fact, the FM-index [58] has been exploited by SGA (String Graph Assembler) in order to build a string graph in $\mathcal{O}(L)$, where L is the sum of all read lengths. More precisely, SGA implements a distributed construction of an FM-index:

the input set of reads $\mathcal{R}$ is partitioned in m subsets $\mathcal{R}_1, \ldots, \mathcal{R}_m$ and intermediate indices are merged using a BWT merging algorithm [59] until a single index for the whole dataset is built. As the memory footprint of the FM-index is typically less than an order of magnitude with respect to the suffix array, this indexing strategy allows to efficiently use the FM-index for very large sequence collections. Nevertheless, the distinctive hallmark of SGA is its ability to directly compute exclusively non-transitive edges and, therefore, immediately build the (reduced) string graph. More technical details on the implementation of this clever technique can be found in [161].

4.1.3 de Bruijn Graph

Given a set of reads, a de Bruijn Graph is built by considering all k-mers belonging to the input reads (see Section 3.4). This graph-theoretic model has two main practical advantages:

i. *computational efficiency:* the overlap detection phase (the main bottleneck of the OLC paradigm) is no more needed.

ii. *memory usage:* it is proportional to the number of distinct k-mers in the genome and not to the number of reads (as in the overlap graph).

For these two reasons, the idea of considering k-mers instead of reads turned out to be extremely successful with very-short-read technologies such as SOLiD and Illumina. Along with the aforementioned benefits, however, there are also two major downsides:

i. *higher graph complexity:* the assembly process is more difficult due to the fact that repeats longer than k base pairs and sequencing errors are likely to create more complex sub-graphs with respect to the OLC approach (see Figure 4.1). For instance, short repeats are collapsed in single paths accessed (in the graph) by many distinct paths which converge into and, subsequently, diverge out from the repeated sequence. This makes the use of long reads (as well as paired ones) crucial to simplify the graph and to obtain correct and contiguous assemblies.

ii. *memory usage:* even though it just depends on the number of distinct k-mers, in large-scale sequencing projects the amount of memory required may still be a bottleneck. This potential issue has been addressed by distributed implementations [162] and sparse/probabilistic data structures [38, 195].

Well known standard dBG assemblers are EULER [140], Velvet [196], Allpaths [30].

EULER. It was the first assembler based on the dBG formalization which models the assembly problem as finding a Eulerian walk (or super-walk). First versions of the tool were specifically designed for single and paired Sanger reads [139, 140]. However, with the advent of NGS, it was adapted to handle 454 [36] and Illumina [35] technologies.

The key idea of EULER is to make use of the information provided by reads (which is lost in the dBG construction). In particular, the idea of finding a Eulerian super-walk (*i.e.*, a path containing each *read-path*) is considered.

As opposed to many contemporary assemblers, EULER performs error correction as a first step. Specifically, EULER seeks for low-frequency k-mers which likely appear

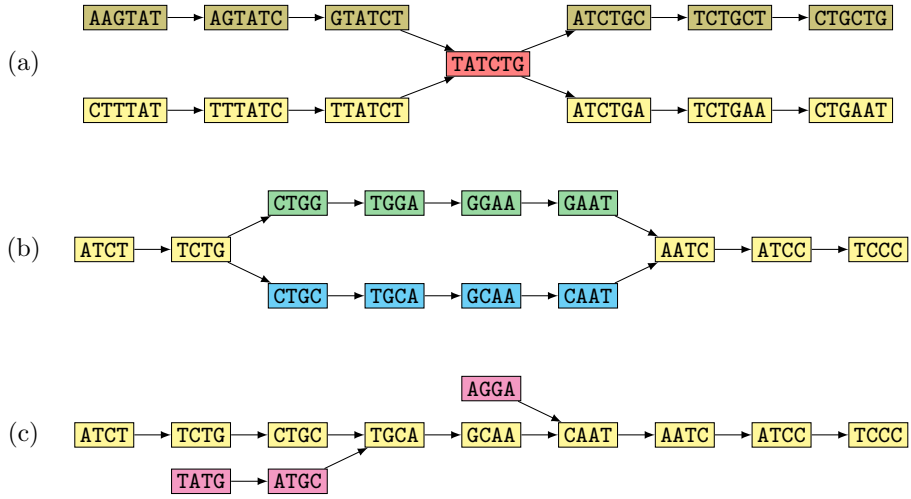


Figure 4.1: de Bruijn graph complexity which may arise due to (a) repeats, (b) variants, and (c) sequencing errors.

due to sequencing errors. This filter is also called *spectral alignment*: the rationale is to correct reads by lowering the number of (distinct) k-mers in the dataset. This correction step, however, might remove valid k-mers belonging to low-coverage *loci*. Moreover, EULER spectral alignment is only able to deal with mismatches and not with insertions or deletions.

From this corrected set of reads, EULER then builds a dBG and maps input reads on some edges in order to resolve repeat-induced (short) paths by exploiting read-paths and paired reads (see Figure 4.2). EULER also performs other graph simplifications such as the removal of short "dead-end" paths (likely due to uncorrected errors).

Since the k-mer size is a critical parameter, as in every other dBG-based assembler, EULER in practice builds two de Bruijn graphs using two different values of k. The rationale is to use a more specific (*i.e.,* with larger k) graph to resolve gaps which are filled using a more sensitive (*i.e.,* with smaller k) graph.

Velvet. This set of tools has been designed for the *de novo* assembly of very short reads. Velvet implements the dBG structure differently with respect to Pevzner's implementation: k-mers are mapped into nodes instead of arcs and reverse-complement sequences are associated to nodes in order to obtain an implicit bi-directed graph.

Velvet performs several graph simplifications. The first one consists in collapsing all unambiguous paths. Subsequently, Velvet removes "dead-end" paths shorter than $2k$ bases and bubbles. These latter ones are resolved using the so-called *Tour Bus* algorithm which is based on a Dijkstra-like breadth-first visit of the graph: just the most supported path is considered, while the other one is removed and reads supporting it are re-aligned. Velvet then uses coverage information to remove connections which might possibly lead to mis-assemblies. Finally, Velvet's last step involves mate pairs that are used to resolve complex and long repeats. The latest version of the tool implements scaffolding using a

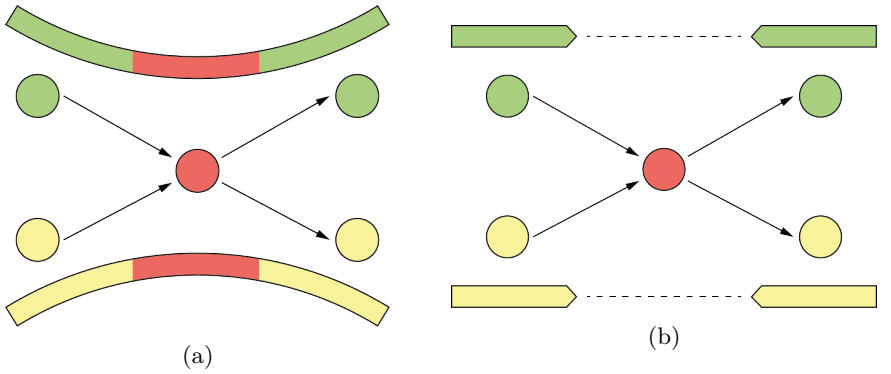


Figure 4.2: de Bruijn graph simplification methods. (a) read-paths allows to resolve repeats of k base pairs and up to the read length. (b) read pairs which span a repeat can be used to identify valid assembly paths.

sophisticated algorithm called Pebble [197].

In summary, Velvet offers a full implementation of the dBG paradigm [116]. It does not use an error-correction procedure directly on reads/k-mers but, instead, it applies a series of heuristics in order to reduce graph's complexity. These involve the use of local graph topology, read coverage, sequence identity, and paired-read constraints.

Allpaths. This assembler uses a read-correction phase similar to Euler's spectral alignment. Specifically, Allpaths considers only *trusted* k-mers, that are those exhibiting both high frequency and good base quality. After the construction of the graph and the canonical removal of sequencing-error tips (*i.e.*, dead-ends), the algorithm identifies *unipaths* (*i.e.,* paths corresponding to *unitigs*). The latter are used as seeds in order to build the assembly (starting from the longer ones). The extension of unitigs is carried out with the help of paired reads. Moreover, Allpaths partitions the graph in order to assemble in parallel regions that are *locally* unique. Small connected components, as well as paths unsupported by read pairs, are finally removed.

It is worth to mention that Allpaths was able to successfully assemble a human genome using Illumina reads, while almost reaching the quality of a Sanger-based assembly [65].

Space-efficient de Bruijn graphs

In dBG-based assembly, small values of k tend to produce a high number of branching nodes and, therefore, ambiguity in the assembly. As a consequence, in large-scale projects the memory space required for storing all k-mers can be extremely high: a canonical implementation of the dBG structure might require over 300 GB [162].

Advances on space-efficient hash schemes [71,108,114,166] can certainly improve the correction of sequencing errors and, hence, the decrease of the graph's size. Unfortunately, a standard dBG representation may still demand hundreds of gigabytes even with corrected reads [98].

Recently, Pell *et al.* [136] introduced the idea of a *probabilistic* variant of de Bruijn graphs, in which the main structure is stored as a Bloom filter [23], and showed that the graph can be encoded with a 4-bit footprint per node. Specifically, a *probabilistic dBG* is obtained by inserting all input k-mers in a Bloom filter B. This representation does not explicitly store edges but they can be implicitly retrieved by testing B for the membership of all possible extensions of a k-mer. However, it is important to mention that this variant is an over-approximation of the de Bruijn graph built from the same k-mers: the use of Bloom filters is subject to *false positive* which introduce *false nodes* (and, hence, *false branches*).

Recently, several methods (and assemblers) were designed to reduce memory requirements of de Bruijn graphs while also making sure assembly accuracy is not worsened. Among them, we mention a distributed (ABySS [162]) and three space-efficient (SparseAssembler [195], SOAPdenovo2 [104], and Minia [38]) assemblers.

ABySS. The major innovation brought by ABySS to the literature is the distributed representation of the de Bruijn graph. This peculiar feature allows to parallelize the assembly of billions of short reads across a network of computers using the MPI protocol [68]. In particular, the set of k-mers is partitioned using a simple XOR-based hashing function. Moreover, a vertex (k-mer) can have up to eight edges (one for every possible one-base extension) and this information can be efficiently stored within 8 bits. In this way, adjacent k-mers can be easily generated and their cluster locations deterministically computed by their hash values.

After the graph construction, ABySS proceeds in two stages. First, the graph is simplified by removing low covered paths and resolving bubbles. These two error removal steps are iterated several times to correct errors that are in close proximity. Second, contigs are built using only highly supported paths and mate-pair information is used to extend assembled sequence by resolving ambiguities in contig overlaps. Mate-pair reads are also exploited to create scaffolds.

SparseAssembler. In order to reduce memory usage, Ye *et al.* [195] proposed a novel approach which exploits *sparseness*: instead of storing every single k-mer (or read, in overlap graphs) as nodes, the idea is to skip a fraction of k-mers (or reads). In the sparse dBG, nodes represent a $1/g$ (approximately uniform) sub-sample of the k-mer diversity in the whole genome. With the "sparsely-spaced" nodes, the authors showed that the memory requirements of such a sparse variant can be considerably smaller than those of canonical implementations.

This different storage method has been implemented in *SparseAssembler* and, while greatly reducing the size of the overall graph, it also proved to be competitive (in terms of accuracy) to state-of-the-art tools both on simulated and real datasets.

SOAPdenovo2. The second version of the SOAPdenovo assembler differentiates itself from the first one in a series of improvements made to the former algorithm (and data structures). In particular, the major achievement consists in how the dBG is built and managed. Other enhancements involve error correction, repeat resolution, scaffolding, and gap closure. Specifically, SOAPdenovo2 is based on a *sparse* dBG [195] and, to further improve assembly's accuracy, it uses different values of k [138]: first, a small-k

graph is built in order to deal with sequencing errors and low-coverage areas; then, larger k-mers are used to rebuild the graph in order to take care of the resolution of long repeats. This last step is carried out by iteratively mapping reads to the previously created small-k graph.

Minia. Chikhi and Rizk took over the idea of probabilistic dBGs [136] and implemented a low-memory assembler (called Minia) based on this data structure. Their implementation, while lowering the memory usage by an order of magnitude with respect to standard dBGs, encloses an additional data structure for dealing with false positives (hence, solving the main issue of probabilistic dBGs). Specifically, they introduce the set of critical False Positives (cFP) k-mers which contains just a *subset* of all possible false positives. Each query to the Bloom filter is then modified such that a positive-membership answer is followed by a query to the cFP set.

The authors showed that these two data structures, along with a proper marking structure to perform the graph traversal, allowed to significantly reduce memory usage compared to assemblers like ABySS, SOAPdenovo, and SparseAssembler.

Paired de Bruijn graphs

One of the major improvements in genome assembly has been definitely brought by paired reads (either PE or MP). As a matter of fact, the use of this type of data has been incorporated in many dBG assemblers but, still, as a mere post-processing step (*e.g.,* graph simplification, scaffolding). For this reason, paired reads often fail to untangle complex repeat structures.

A more advanced formalization of the genome assembly problem in the dBG framework considers the reconstruction of a genomic string from a set of k-mer *pairs* (or k-bimers) at a certain distance. From this idea, Medvedev *et al.* [113] introduced the notion of *paired de Bruijn graph* (PdBG): a generalization of the canonical dBG which additionally incorporates pairing information within its topology. However, PdBGs were introduced more as a theoretical result rather than a practical solution due to the very limiting assumption that the exact distance between paired k-mers is known. Nevertheless, the main issue of this model has been addressed by the SPAdes assembler [16].

SPAdes. This tool recently introduced new algorithmic solutions and improvements over state-of-the-art tools to handle bacterial datasets. A hallmark of SPAdes is its use of k-mers only for building the initial graph. Subsequent steps are exclusively carried out with graph-theoretic operations based on graph topology, coverage, and sequence lengths (but not the actual sequences).

More in detail, this assembler implements two variations of the canonical framework: *paired* and *multi-sized* dBGs. The first one is considered after transforming the set of k-bimers (extracted from paired reads) into a set of *adjusted* k-bimers by estimating their distance accurately (and, hence, making Medvedev's formalism viable). The second framework involves using multiple values of the k parameter, which is tuned to deal with low and high coverage regions in order to possibly reduce fragmentation and repeat collapsing. Using such dBG variants, the algorithm of SPAdes is able to deal with issues such as sequencing errors, uneven coverage, insert-size variation, and chimeric (paired) reads.

4.1.4 Branch-and-bound assemblers

A completely different strategy, that does not focus on employing heuristic to circumvent the NP-hardness of the theoretical problem but tries to solve it in exactly, has been recently explored by Narzisi and Mishra in [127].

Their idea consists in performing a constrained search (within the solution space) while pruning implausible layouts. Such a branch-and-bound (B&B) approach is based on a collection of well-chosen score functions which characterize different structural properties.

B&B represents a different – and interesting – point of view on the assembly problem and, for this reason, it deserves a separate category. At this moment, the only B&B assembler available is SUTTA [127].

SUTTA. At a high level, SUTTA generates (potentially) all possible consistent layouts, organizing them as paths in a structure called *double-tree* (or D-tree) rooted at randomly selected seeds (*i.e.,* reads). Paths are progressively and concurrently evaluated by the aforementioned score functions. The possible exponential search space of the D-tree is addressed by pruning many redundant and uninformative branches of the tree (*i.e.*, transitively deducible paths, those unsupported by paired reads, and extremely low-coverage paths).

The strength of this scheme is that, in principle, it can be easily extended to deal with different kinds of data (paired reads, optical maps, *etc.*) and, possibly, future technologies.

4.1.5 *De novo* assembly with long and noisy reads

The availability of TGS datasets can be exploited in many ways to build – or just improve – *de novo* assemblies. Moreover, it is going to be particularly useful for large and complex genomes, where the high percentage of repeats makes short-read assemblers often inadequate for achieving satisfactory results. Yet, the high error-rate provided by currently available technology poses additional algorithmic challenges and the need to devise novel and effective methods.

Nowadays, different kinds of assembly procedures have been developed (mostly for PacBio reads) and they can be grouped according to the following four categories.

- *TGS-only*: it involves the exclusive use of TGS libraries. Sequences are usually corrected before assembling them using an OLC algorithm. HGAP [39] is probably the best known pipeline in this category. Recently, also pacBioToCA [90] implemented this strategy.

- *Hybrid*: combining the high-quality NGS reads and the large-length TGS reads allows to produce very long genomic contigs that otherwise would require costly low-throughput techniques. Cerulean [49], pacBioToCA [90], and dBG2OLC [194] are three implementations of this methodology.

- *Gap filling*: gaps inside the scaffolds of an existing paired-read-based assembly are reconstructed using TGS sequences. A tool available for this task is PBJelly2 [55].

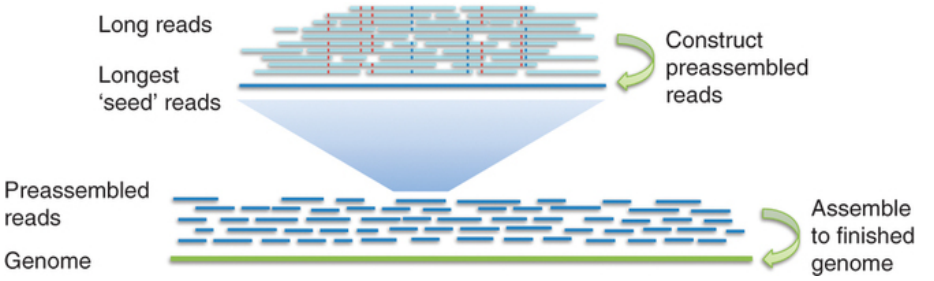


Figure 4.3: A sketch of HGAP. Source: [39].

- *Scaffolding*: scaffolds are built using TGS reads to join contigs of an assembly constructed with NGS data. Known tools for scaffolding using PacBio reads are AHA [17] and PBJelly2 [55].

HGAP. The Hierarchical Genome Assembly Pipeline involves three main phases.

First, longest reads are used as seeds and, through a directed acyclic graph-based consensus procedure, they are corrected using the smaller reads in order to build a set of highly accurate pre-assembled reads (see Figure 4.3)

Second, corrected sequences are assembled with the Celera Assembler [123]. However, in principle any assembler could be considered in the pipeline (yet OLC assemblers are better suited for this task). Indeed, the quality of the resulting draft assembly depends on coverage and length distribution with respect to the repeat content of the genome.

Third, in order to significantly reduce remaining indels and single-base errors, a quality-aware consensus algorithm that uses the quality scores provided by the PacBio platform should be used. For instance, the Quiver algorithm can be used to derive a highly accurate consensus sequence.

pacBioToCA. Thanks to the uniform distribution of errors in PacBio reads, it has been shown that such reads can be effectively corrected using an NGS dataset in order to obtain highly accurate sequences [90]. For instance, short reads can be mapped using aligners like NovoAlign [72] and GMAP [192], able to tolerate larger edit distances. This kind of alignment is however computationally expensive and requires a quite large running time even for bacterial datasets. Formerly, pacBioToCA was a pre-processing tool based on this idea. A recent version, however, employs MHAP [21] for a faster overlap detection and to perform error correction using the same input TGS reads. Corrected reads are then assembled using the latest version of the Celera Assembler.

Cerulean. Cerulean is a hybrid assembly approach to produce high-quality assemblies using Illumina PE reads and PacBio long reads. The peculiarity of this tool is to avoid using short reads directly. Instead, it requires a short-read assembly graph structure to be generated with an existing assembler (*e.g.*, ABySS [162]). Specifically, an assembly graph consists of nodes corresponding to contigs and edges representing putative adjacencies. Cerulean's output is then built using a mapping of PacBio reads to the contigs

using BLASR [34].

The alignment of long reads to very short contigs might generate several spurious connections in the assembly graph. In order to mitigate this issue, the algorithm of Cerulean initially operates with a simplified representation of the assembly graph, consisting exclusively of long contigs. The graph is then improved by iteratively adding the smaller sequences.

dBG2OLC. The relatively high sequencing depth (usually 50–100×) required in order to perform the error correction of long noisy reads is making the transition from NGS to TGS technologies quite slow [194]. In view of this, Ye *et al.* devised a novel assembly technique which maintains a comparable accuracy while requiring a significantly lower sequencing coverage (approximately 10–20×) with respect to existing solutions.

Specifically, the idea implemented in their hybrid assembler (dBG2OLC) is to *anchor* TGS reads to the dBG contigs obtained from a NGS dataset. Additionally, each long read is compressed into a list of anchors in order to allow pair-wise alignments to be computed much more efficiently. Subsequently, an overlap is built directly from the compressed reads and a consensus is devised from linear regions of the graph to finish the assembly.

Compared to other existing approaches such as pacBioToCA and HGAP, it has been shown that dBG2OLC can produce decent assembly drafts while using orders of magnitude lower computational resources (both memory and time) and requiring a much smaller TGS-read coverage.

4.2 Assembly validation

Many biological questions are tightly connected to the analysis of genome sequences and, hence, to the assembly problem. For example, studies showed that genome structural variations are related to numerous diseases [170] (*e.g.*, Parkinson's and Alzheimer's). Nevertheless, knowledge provided by the comprehension of genomes is not only important to understand (and possibly treat or prevent) disease traits but also for studying the evolution of living organisms. For these reasons, being able to obtain an *accurate* reference genome is mandatory to perform a large number of downstream analyses.

While since the first sequencing project different theoretical models and a large amount of clever heuristics have been proposed in order to handle errors and to efficiently provide a reliable genome assembly, yet a small effort has been done on the equally important *assembly validation* task.

For many years, assemblies have been evaluated using mostly inappropriate metrics (*e.g.*, NG50, mean contig size, etc.) which just reflect *contiguity* at the expense of *correctness*. Also, with the advent of high-throughput sequencing, the assembly problem became more difficult due to the shorter read length and this increased the chances of an assembler to make mistakes in the assembly process. As a matter of fact, Alkan *et al.* [12] criticized two major achievements: the assemblies of the Han Chinese and the Yoruban individuals [98] (both sequenced with Illumina reads). In particular, approximately 420 Mbp of missing (repeated) sequences were identified from the Yoruban assembly and it was estimated that both miss almost the 16% of the genome sequence. For this reasons,

as the number of NGS projects keep growing, the need of *standard* and *reliable* validation methods is becoming every day more impelling.

4.2.1 Validation metrics.

The validation of an assembly should account for three main traits: *contiguity* (*i.e.*, the length of its contigs), *completeness* (*i.e.*, the percentage of the genome that has been assembled), and *correctness* (*i.e.*, the number of errors introduced by the assembler).

One of the most used metric to estimate contiguity is certainly the N50.

Definition 4.1 (N50 and NG50)**.** Let $\mathcal{A} = \{C_1, \ldots, C_n\}$ be an assembly and assume that $|C_i| \geq |C_{i+1}|$, for $i = 1, \ldots, n$ (*i.e.*, contigs are sorted by non-increasing length). Then, the N50 is defined as the length $|C_k|$, where k is the minimum integer such that:

$$\sum_{i=1}^{k} |C_i| \geq 0.50 \cdot \sum_{i=1}^{n} |C_i|.$$

In other words, it corresponds to the maximal length N of a contig in $\mathcal{A}$ such that at least half of the total assembly size is "covered" by all contigs with length greater or equal than N. The NG50 is defined similarly, with the only difference that the *cumulative* assembly size is replaced by the *estimated* genome size.

Other widely used statistics concerning contiguity and completeness are the number of contigs, the average length of contigs, and the total assembly length. As shown in [128], however, the exclusive use of contiguity measures should be avoided and, therefore, more sophisticated metrics shall be taken into account. In fact, the main downside of length-based statistics is being completely unrelated to assembly correctness. Think of an assembler that blindly *glues* most of its contigs producing an assembly characterized by a very large N50 and few long contigs. However, these contigs most certainly will contain a too large number of mis-assemblies to be useful in downstream analyses.

Assuming a reference genome is available, it is possible to compute a precise characterization of errors by aligning the assembly to the reference sequence. A representative study of reference-based approaches is presented in [106, 154] and will be briefly discussed in Section 4.2.2. Other reference-based approaches consist in comparing a *de novo* assembly with a *similar* finished genome [97, 102] or with conserved sequences of *related* organisms [40].

If a reliable reference sequence is not available, instead, the evaluation of correctness is more difficult and it is usually carried out exploiting read mappings [82,141] (but also optical maps [199]) in order to compute quality measures – often referred to as *features* – such as:

- *read coverage*: regions exhibiting an unusual coverage may indicate mis-assemblies such as translocations (*i.e.,* junction of far apart genomic *loci*), repeat expansion, or repeat collapse.
- *paired reads accordance*: a region presenting a high-enough number of correctly aligned reads with unmapped mates (or mapped at an unexpected distance) is a good indicator that a mis-assembly occurred. This situation usually involves either translocations, insertions, deletions or inversions.

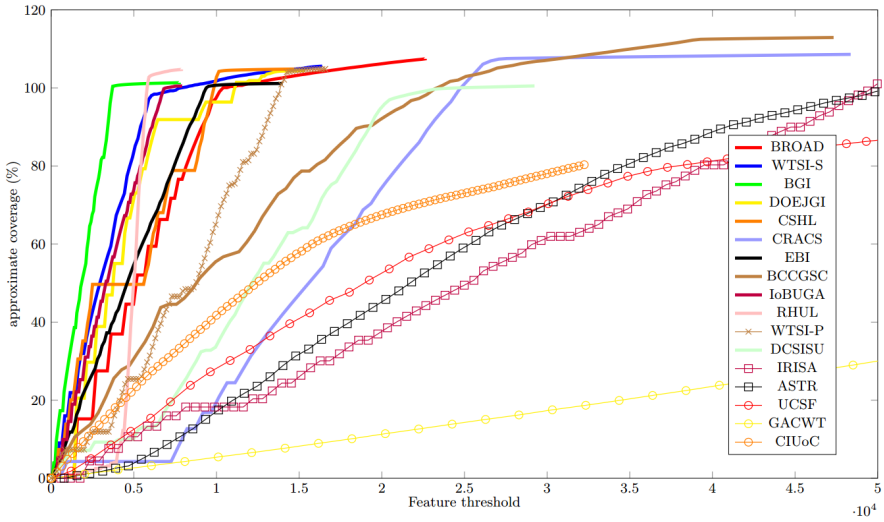


Figure 4.4: FRC example on Assemblathon 1 assemblies [184].

- *k-mer frequencies*: by comparing those of k-mers in input reads with frequencies of k-mers in the assembly, it is possible to identify putative errors in presence of unexpected k-mer multiplicities.

In [141], a tool called `amosvalidate` was proposed with the goal of identifying a set of *features* able to report regions or evidences that likely represent assembly errors in contigs. Their approach definitely represented an improvement over the traditional – often overrated – length-based evaluations.

More recently, Narzisi and Mishra [128] exploited this pipeline to introduce a novel reference-less metric which captures the trade-off between correctness and contiguity: the Feature Response Curve (FRC). More precisely, the FRC allows to characterize the sensitivity (coverage) of contigs as a function of the number of features (see Figure 4.4). The curve is built as follows: first, contigs are sorted by length in descending order and, then, for each feature threshold T, only the longest contigs whose number of features is less than T are considered to compute the genome coverage. As a rule of thumb, a steeper curve (which also approach "rapidly" the 100% coverage) usually reflects a better assembly since it means there are less negative features in the longer contigs.

4.2.2 Evaluation studies and results

Several works concerning the evaluation of assemblers have been proposed in literature with the aim of ranking the performance of available tools on different datasets: Assemblathon 1 [52], GAGE [106, 154], Assemblathon 2 [26], and the re-evaluation of GAGE datasets using the FRC [184] are probably the most important ones.

The first Assemblathon contest was made in order to assess the performance of state-of-the-art algorithms on a synthetic diploid genome (unknown to the participating teams). Within this study it has been shown that:

1. it was possible to assemble the genome to a high level of coverage and accuracy;

2. large differences exist between the assemblies (some assemblers performed well on some metrics but poorly on others), suggesting that further improvements in current methods can be achieved.

However, it is worth noticing that such evaluation was performed only on a single simulated dataset, limiting the extent to which the conclusions can be generalized [183].

Similarly, GAGE's study [154] (and GAGE-B, the bacterial counterpart [106]) evaluated several top-notch assemblers on real Illumina datasets. This work showed that data quality – more than the assembler itself – is a very important factor to obtain high quality results and it also confirmed the sensible difference of contiguity and correctness between assemblies of different tools. A criticism against GAGE is that each assembler was tuned in order to maximize the resulting NG50 (this was done to mimic the typical user behavior). Nevertheless, NG50 has been showed to be the worst quality predictor [183].

As opposed to the aforementioned studies, in Assemblathon 2 the correct genomic sequences were unknown. Because of this, various experimental datasets where used, such as fosmid sequences and optical maps, in order to asses the results. Another goal of the study was to verify the suitability of different metrics to evaluate assembly's quality. What Assemblathon 2 showed is that on challenging genomes (and with a reasonable amount of sequencing):

1. Assemblers are extremely sensitive to parameters and the performance of different assemblers varies substantially on the same dataset.

2. The same assembler with the same parameters performs differently on different datasets.

Despite the FRC allows to evaluate assemblies without the need of a reference sequence, the main limitation of the method is that it requires a read layout (missing in the majority of NGS assemblers). For this reason Vezzi *et al.* [184] developed a tool called FRC^{bam} which allows to evaluate *de novo* assemblies by computing the curve using an alignment-based layout when it is not provided by the assembler (as in many dBG-based implementations). The tool has been applied to evaluate datasets of Assemblathon 1, GAGE, and Assemblathon 2. It was then showed that in many scenarios it is straightforward to rank assemblers simply by looking at the FRCs and, even when it is unclear which assembly might be the best, this methodology still allows to highlight positive and negative aspects of assemblers with respect to each others.

5

De Novo Assembly Through Reconciliation

5.1 Background

The advent of Next Generation Sequencing (NGS) technologies made possible to sequence virtually all the organisms of the biosphere [109]. NGS technologies are characterized by extremely high data production which makes them affordable to obtain high coverage of any organism.

The ability to produce high sequence coverage for lots of genomes paved the way to a large number of *de novo* assembly projects [45, 97]. Despite this, it is now commonly accepted that *de novo* assembly with short reads is more difficult than *de novo* assembly with long Sanger reads [126]. Short read length and reduced insert size made correct assembling and positioning of repeats a very crucial and delicate issue. Even though some works presented high quality results based on NGS data [98,133], *de novo* assembly, especially for large eukaryote genomes, is still a holy grail [12, 22].

Recently, several evaluations have been presented in literature (see Section 4.2), trying to rank the performance of assemblers on different datasets. As a byproduct, these "competitions" showed the extreme difficulty to elect the best assembler. Each dataset is characterized by different peculiarities and heuristics implemented by each assembler are usually only partially able to solve the raised issues.

An interesting strategy to improve *de novo* assemblies has been proposed and goes under the name of *assembly reconciliation* [32, 200]. The goal of assembly reconciliation is to merge the assemblies produced by different tools while detecting possible misassemblies and isolating problematic regions. Such a strategy has already been proposed for Sanger-based assemblies and one of our goals is to adapt and improve it for NGS data.

Zimin *et al.* in [200] presented Reconciliator, which is based on an iteration of errors identification and correction, and merging phases. Using the so called CE statistics they identify regions likely to contain errors in the assemblies. After this, a global alignment

between the two assemblies is performed. In order to avoid problems with repeats, alignment is performed using seeds unique in both the reference and the query sequences. At this point areas marked as problematic are solved using the assembler with better CE statistics and possible gaps in the assemblies are filled. The last step consists in the validation of the merged assembly.

Casagrande and colleagues in [32] proposed GAM (GAM-NGS's ancestor), a tool similar to Reconciliator, but able to avoid the global alignment step. In order to identify similar sequences they searched for areas assembled by the same reads. Subsequently the notion of "block" is introduced to evaluate sensible local alignments and a graph is built to describe global relationships between the two assemblies. When confronted with problematic regions (*e.g.*, loops and bifurcations in the graph), GAM uses one of the assemblies as guide.

Both Reconciliator and GAM have advantages/disadvantages on one another (*e.g.*, GAM does not need a global alignment while Reconciliator does, however GAM was not able to detect and correct mis-assemblies). Nevertheless, both tools share the limitation that they are tailored for Sanger-based assemblers. As an example, they both need a layout file (usually an afg file) describing for each read the (unique) position where it has been used. In NGS assemblers, such a layout file is provided by a small minority of tools (*e.g.*, Velvet, Ray and SUTTA). Moreover, another limit of both tools is the fact that the two input assemblies must have been produced using the *same* set of reads.

Recently, several new tools appeared, tackling the problem of assembly reconciliation using NGS-like datasets: GAA [193], ZORRO [7], Mix [168], GARM [167], and Metassembler [190].

GAA performs a global alignment between two assemblies (using BLAT). The alignment is used to build the so called Accordance Graph in order to merge the assemblies. In the merging phase reads are used to solve possible inconsistent links in order to output a correct assembly. Morover, GAA focuses more on avoiding the introduction of mis-assemblies instead of correcting them.

ZORRO performs a first error correction phase directly on the original contigs and then a global alignment using *nucmer*. The alignment is used to order contigs and deriving a consensus sequence. The main drawback of both GAA and ZORRO is the mandatory global alignment phase between the assemblies, which is not only a computational expensive step, but, in presence of ortholog and paralog sequences, it may produce a large number of false links affecting merging performances. ZORRO has been explicitly designed for short genomes (as size increases, merging is not feasible).

Mix is a tool that merges two or more draft assemblies, aiming to reduce contig fragmentation and to speed up finishing of bacterial genomes. The algorithm is based on a *extension graph* which models prefix/suffix overlaps among contigs of the input assemblies. Mix then determines a set of non-overlapping maximal independent longest paths to merge contigs. The tool, however, does not try to correct errors but instead it focuses just on (blindly) improving contiguity without taking into account more qualitative metrics (we already emphasized in Section 4.2 that contiguity is a bad indicator for assessing correctness). Moreover, the computational complexity of this method can be exponential in the size of strongly connected components (SCCs) in the graph. For this reason, Mix might not be well suited to work on large and complex genomes, where large SCCs are expected.

GARM, like GAM-NGS, also treats assemblies asymmetrically but – differently from GAM-NGS – it chooses the better assembly according to several statistics. GARM's pipeline works as follows: (i) *nucmer* is used to align assemblies to each other; (ii) ambiguous overlaps and inclusions are removed; (iii) a layout is generated along with a consensus scores (iv) contigs are merged exploiting also their orders in scaffolds.

Metassembler, finally, is based on the extensive use of compression-expansion statistics in order to iteratively merge assemblies two at a time. First, Metassembler uses *nucmer* to align the input assemblies and to find *breakpoints* (*i.e.*, the boundaries of found alignments). Then, for each region between breakpoints, the assembly with the best compression-expansion statistics is chosen for generating the output.

Other tools that belong to the assembly reconciliation family are MAIA [130], e-RGA [33], and the Velvet's Columbus module. However, they focus more on enhancing *de novo* assembly results guided by a reference sequence belonging to closely related species, than on pure reconciling *de novo* assemblies.

With this picture in mind we developed GAM-NGS (Genomic Assemblies Merger for Next Generation Sequencing) whose primary goal is to merge two assemblies in order to enhance contiguity and possibly correctness. GAM-NGS does not need global alignment between contigs, making it unique among assembly reconciliation tools. In this way not only a computationally expensive and error prone alignment phase is avoided, but also much more information is used (total read length is usually one or two order of magnitude higher than the mere assembly's length). Read alignments allow the identification of regions reconstructed with the same reads, thus isolating natural candidates to represent the same genomic *locus*. GAM-NGS merge-phase is guided by an Assemblies Graph (AG). AG is a weighted graph and this is another specific feature of our tool. Weights indicate the likelihood that a link is part of a correct path. AG allows GAM-NGS to identify genomic regions in which assemblies contradict each other (loops, bifurcations, *etc.*). In all these situations weights are *locally* used to output the most reliable sequence, given the information in AG.

GAM-NGS requires as input two assemblies and a SAM-compatible alignment (*e.g.*, obtained with BWA [95], ERNE [48]) for each input read library and each assembly. GAM-NGS can also work with assemblies obtained using different datasets, as long as the set of reads aligned on the assemblies is the same. It is important to note that, mapping reads back to the assembly is practically a mandatory phase for a large number of downstream analyses (*e.g.*, SNP calling, repeat analyses, *etc.*) and therefore represents no extra cost.

We tested GAM-NGS on six datasets. We used three GAGE datasets [154] in order to evaluate GAM-NGS and to compare it with other assembly reconciliators (*i.e.*, GAA and ZORRO). Moreover, in order to show GAM-NGS data and "biological" scalability, we tested it on three large plant datasets: a *Prunus persica* genome (227 Mbp, double haploid), a *Populus nigra* genome (423 Mbp, heterozygous) and a *Picea abies* genome (20 Gbp, diploid and highly repetitive). GAM-NGS turned out to be able to correctly merge these assemblies, significantly improving the results achievable using only one assembler. Statistics computed on GAM-NGS outputs show comparable results with respect to other assembly reconciliation tools. Nevertheless, GAM-NGS is always the fastest and the least computationally demanding tool, which makes GAM-NGS the best candidate for large datasets.

5.2 GAM-NGS: efficient assembly reconciliation using read mapping

GAM-NGS's main idea is to identify highly similar fragments between two assemblies, searching for regions sharing a large amount of mapped reads. The assumption is that areas built using the same reads most likely represent the same genomic *locus*.

The vast majority of NGS assemblers does not return a layout file as output (*i.e.*, a file, usually in afg format, listing along the assembly the reads used and their positions). In order to overcome this limit, GAM-NGS approximates the layout file using reads aligned back to the assembly: an analysis step almost mandatory in all *de novo* assembly projects. Such an approximation might be prone to errors: as an example, consider a genome containing (almost) perfectly duplicated regions. In such a case genomic read belonging to any two repeated sequences will be randomly assigned to one of the two copies. In order to keep problems related with repeats, at least partially, under control, GAM-NGS uses only reads *uniquely* aligned (*i.e.,* mapped to a single position), discarding all reads that have been *ambiguously* aligned (*i.e.,* characterized by two or more high-score alignments).

As a matter of fact, since assemblers implement different heuristics – if this was not the case, merging would be trivial and pointless – they may contradict each other by presenting a diverse sequence order or erroneously splicing (*e.g.*, scaffolding) sequences belonging to different genomic regions. Thus, it is compulsory to identify these situations and, possibly, solve them. To address this problem we used a graph structure – the *Assemblies Graph* (or AG) – which records and weights the most probable order relation among regions (*blocks*) where the same reads are mapped.

Once AG is built, GAM-NGS identifies "problematic" regions, signalled by specific sub-graph structures. Such local problems are solved by selecting the path in the graph that maximizes a set of measurable and local features, suggesting the assembly's correctness. Some of these features are borrowed from [183] and are computed using pairing information coming from aligned paired-end and possibly mate-pair reads libraries. If there is not enough evidence to decide on assembly correctness (*e.g.*, weights are too close to each other), we chose to be as conservative as possible, electing one of the sequences as *master*, the other one, therefore, becoming the *slave*. In the following sections we will denote the *master* assembly as M and the *slave* one as S.

After this last phase, GAM-NGS visits the simplified graph, merges contigs finding a consensus sequence and finally outputs the improved assembly.

5.2.1 Definitions

Let Σ be an alphabet and Σ^* be the set of finite-length strings from Σ. For every $s \in \Sigma^*$ we will denote by $|s|$ the number of characters in s. In our context reads and contigs are elements of Σ^*, where $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{T}, \mathtt{G}, \mathtt{N}\}$. With $\mathcal{R} = \{r_1, r_2, \ldots, r_n\}$ we denote the set of reads aligned against both M and S, which are the master and slave assemblies, respectively. Usually $\mathcal{R}$ is the set, or a subset, of reads used to assemble both M and S and its elements may belong to different paired read and mate pair libraries. However, alignments of reads belonging to different libraries should be provided into separate alignment files, in order to exploit the information of different insert sizes.

Let r_1, r_2 be two reads aligned against the same contig C (with C belonging to either M or S). For $i \in \{1, 2\}$, let $begin(r_i)$ and $end(r_i)$ be the positions in C where the first and last base of r_i are aligned, respectively. Therefore, we can assume $begin(r_i) < end(r_i)$, for $i \in \{1, 2\}$. We say that r_1 and r_2 are *adjacent* if and only if $begin(r_2) \leq end(r_1) + 1$ and $begin(r_1) \leq end(r_2) + 1$.

Given a contig C belonging to assembly A, a *frame* over A is defined as a maximal sequence of reads $r_1, \ldots, r_n$ mapped against A where r_i, r_{i+1} are adjacent (*i.e.*, overlapping or close enough) for $i = 1, \ldots, n-1$. Thus, a frame F can be identified by the contig where its reads are aligned and the interval $[\, begin(F), end(F)\,]$, where $begin(F) = \min\{\, begin(r_i) \mid i = 1, \ldots, n\,\}$ and $end(F) = \max\{\, end(r_i) \mid i = 1, \ldots, n\,\}$. Moreover, we define the length of a frame F as $|F| = end(F) - begin(F) + 1$.

Given two different assemblies M and S, we define a *block* B as a pair of frames (one over M and one over S) consisting of the same sequence of reads $r_1, \ldots, r_n$, and the size of the block as the number of reads from which it has been constructed. If the majority of the reads r_i are aligned with opposite orientations on the two frames, we say that B is *discordant*. Otherwise, we say that B is *concordant*. We remark that we are interested in finding blocks where the sequence of reads (the frame) is maximal (*i.e.*, it cannot be extended further with other reads). Ideally, blocks should represent those fragments of the considered genome which have been built in accordance by both the assemblies.

In the following we will first explain how blocks are built from alignments and then we will show how blocks are filtered in order to avoid spurious blocks produced as consequence of the existence of similar genomic regions. After this we will illustrate the Assembly Graph construction, the handling of the problematic regions identified on the graph and, lastly, how the merging phase is carried out.

5.2.2 Blocks construction

The first, and most computational demanding, step of GAM-NGS's outer algorithm is the identification and construction of blocks between assemblies M and S. The basic input format are BAM files (*i.e.* files in the, by now, standard alignment format). Alignments are assumed to be ordered by their contig identifier and by the alignment position.

The procedure starts by loading into a hash table $\mathcal{H}$ all the reads uniquely mapped on M. Once $\mathcal{H}$ has been populated, uniquely mapped reads on S are processed. In particular, for each read r, we perform the following steps:

- if r is not present in $\mathcal{H}$, we will not use it for blocks construction;
- if r is adjacent to a previously created block B (*i.e.*, adjacent to a read contained in both its frames), then B is extended using r;
- otherwise, a new block, started by the single read r, is built.

Storing in main memory all the alignments of M and going through all the alignments of S may easily become a major computational stumbling block. For this reason we carefully designed the data structures and the relative manipulation algorithm. Each uniquely aligned read requires only 21 bytes: 8 bytes for its identifier, 4 bytes for contig's

identifier, starting and ending position, and 1 byte for mapping orientation (reverse complemented or original strand). Moreover, we decided to store them in a memory efficient hash table such as Google's *SparseHash* [3], which is characterized by a 2 bits overhead per entry.

For each processed read r mapped on a contig C of an assembly A, we define the *scope* of r as the set of blocks whose frame on C is adjacent to r. We exploit the fact that input alignments are ordered, during the blocks construction phase: if a block B is "out of scope" for the current processed read r then B will not be successively altered. If the size of B is higher than a user predefined threshold B_{min} then B is saved into secondary memory and main memory space is released. Otherwise, B is discarded. The rationale behind the B_{min} threshold is that blocks consisting of only few reads are likely to be a consequence of alignment errors or chimeric sequences.

5.2.3 Blocks filtering

A typical problem common to all assembly reconciliation tools is that, especially with highly repetitive genomes, similar regions belonging to different genomic areas might be merged (such a problem is also common among *de novo* assemblers). In particular, GAM-NGS may build blocks between regions that attract the same reads only because they are similar (note that perfect genomic repeats are not a problem because in this case reads will be ambiguously aligned). This situation not only complicates Assemblies Graph's structure, but it also suggests the presence of problematic regions (*i.e.*, errors) in sequences that are, in fact, correct. To limit this problem, GAM-NGS runs two additional filtering steps before the graph construction: one based on *depth-of-coverage* analysis, and the other one on *block-length* considerations.

More specifically, considering a block B with frames F_M, F_S, on M and S, respectively, GAM-NGS computes for each frame two different types of coverages: a *block coverage BC* and a *global coverage GC*. For instance, considering the frame on the master assembly F_M, let $\mathcal{R}_{F_M}$ be the set of *all* reads uniquely aligned on F_M, while let be $\mathcal{R}_{B_M}$ the set of reads uniquely aligned on F_M and used as part of block B. Clearly, $\mathcal{R}_{B_M} \subseteq \mathcal{R}_{F_M}$. Moreover, we define the block coverage of F_M as

$$BC_{F_M} = \frac{\sum_{r \in \mathcal{R}_{B_M}} |r|}{|F_M|}$$

and the global coverage of F_M as

$$GC_{F_M} = \frac{\sum_{r \in \mathcal{R}_{F_M}} |r|}{|F_M|}.$$

At this point, GAM-NGS keeps only blocks satisfying the following condition:

$$\max \left\{ \frac{BC_{F_M}}{GC_{F_M}}, \frac{BC_{F_S}}{GC_{F_S}} \right\} \geq T_c,$$

where T_c is a user defined real number in the interval $[0, 1]$. The idea is to get rid of blocks built using a low amount of reads compared to the number of mapped reads on

both frame intervals (see Figure 5.1).

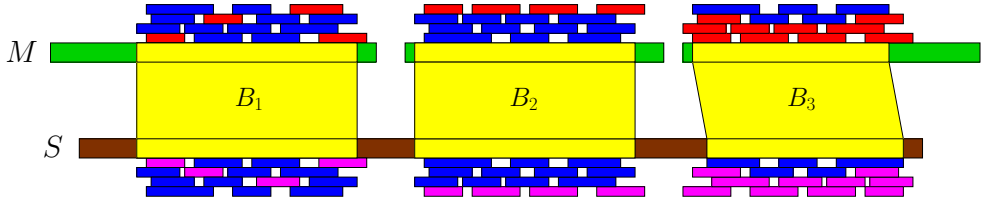


Figure 5.1: Blocks construction in GAM-NGS. Blocks are identified by regions belonging to M and S that share a relatively high amount of mapped reads. In this figure, blue reads identify clusters of adjacent reads that are uniquely mapped in the same contig of both the assemblies. Moreover, GAM-NGS discards blocks like B_3 that contains a small amount of shared reads compared to the number of reads aligned in the same regions (*e.g.*, in B_3 these are less than 35% and this block may create a wrong link between contigs).

We decided to use the maximum between the two ratios in order to avoid the removal of blocks corresponding to heterozygous regions: it may happen that one assembler returns both alleles while the other returns only one of them. In this case, the proportion of reads used in the block should be close to 1 and 0.5, respectively.

The second filtering step is based on the length of block's frames. In particular, given a block B composed of frames F_{M_i}, F_{S_j} on contigs $M_i \in M$ and $S_j \in S$ respectively, B is retained if

$$|F_{M_i}| \geq \min\{0.3 \cdot |M_i|, T_l\} \ \vee \ |F_{S_j}| \geq \min\{0.3 \cdot |S_j|, T_l\},$$

where T_l is a user-defined threshold. Nevertheless, when this condition is not satisfied we still retain the block if any of the following conditions is satisfied: there are other blocks between M_i and S_j satisfying the condition or this is the only block between the two contigs. The rationale is, again, to discard blocks that are likely to be consequences of wrong alignments or chimeric regions, while keeping small blocks that can still witness insertions or deletions by one of the two assemblies.

5.2.4 Assemblies graph construction

For each assembly, we can define a block order relative to an assembly exploiting frames' order along its contigs. In particular, consider an assembly A and two blocks B_1 and B_2 with frames F_1^A and F_2^A, respectively, both on A. We say that B_1 *comes before* B_2 with respect to A if and only if both F_1^A and F_2^A lie on the same contig C_A and F_1^A comes before F_2^A (*i.e.*, $begin(F_1^A) < begin(F_2^A)$) and there is no frame F_3^A lying over C_A for which F_1^A comes before F_3^A and F_3^A comes before F_2^A.

It is important to point out that this block order strictly depends on the considered assembly, since the same genomic region may have been reconstructed on opposite strands in the input assemblies. Thus, there may be cases where B_1 comes before B_2 with respect to M, but B_2 comes before B_1 with respect to S. In this scenario, block orders of the two assemblies may contradict each other (leading to cycles in AG) even

when there is no contradiction at all.

Our goal is to determine a consistent order of blocks among each contig of both the assemblies. To facilitate that, we build a *Contigs Graph* (CG) which consists of a vertex V_{M_i} for each contig $M_i \in M$ and a vertex V_{S_j} for each contig $S_j \in S$. Two vertices V_U and V_W are connected by an undirected edge if and only if U and W belong to different assemblies and have at least one block over them.

For each edge e connecting two vertices V_{M_i}, V_{S_j}, we assign the weight

$$w_e = \max\left(\frac{r^+}{r^+ + r^-}, \frac{r^-}{r^+ + r^-}\right),$$

where r^+ and r^- are the number of reads belonging to concordant and discordant blocks between M_i and S_j, respectively. For each vertex V the weight w_V is then computed, corresponding to the mean of its incident edges' weights (this mean is weighted on the overall size of all blocks connecting two contigs). The main idea is that edges' weights will have a value close to one when the majority of the reads composing the blocks are mapped either with the same orientation or with the opposite orientation. In the former case contigs will most likely have the same orientation, while in the latter case one of the two contigs must be complemented and reversed.

In more detail, let $\mathcal{Q}$ be the set of processed vertices. At first, for each connected component of CG, we insert into $\mathcal{Q}$ a vertex V which maximizes w_V and we set the original blocks' order for V's contig. Then, we repeat the following steps until all vertices of the graph belong to $\mathcal{Q}$:

- Pick $V \in \mathcal{Q}$ with largest w_V;
- Let $adj(V)$ be the set of the vertexes adjacent to V. For each vertex $V_U \in adj(V)$, we set the order of blocks on U depending on whether the majority of reads belongs to concordant or discordant blocks and according to blocks' order of V's contig;
- $adj(V)$'s elements are added to $\mathcal{Q}$ and we remove V's incident edges from the graph, updating vertices' weights.

The rationale behind this heuristic is that, at each iteration, we set the order of the blocks over one of the contigs for which we have the clearest evidence. However, this is a simple (yet effective) procedure to compute a consistent blocks' order among the assemblies and we plan to improve it in order to have a higher guarantee of avoiding the introduction of "false contradictions" (*i.e.*, cycles) in AG.

With the updated blocks order we are now able to build the *Assemblies Graph* (AG): a node V_B is added for each block B, while edges connect blocks that share at least one frame on the same contig. In particular, if a block B_1 comes before a block B_2 with respect to M or S we put a directed edge from V_{B_1} to V_{B_2} (see Figure 5.2). Notice that, since we are considering the merging of two assemblies, each node cannot have an input or output degree strictly greater than two.

Moreover, during AG construction, we add to each edge a weight characterized by a series of features that are evaluated within the region relative to the blocks related to the vertices connected by the edge.

Let V_{B_1}, V_{B_2} be two nodes linked by an edge (*i.e.*, B_1 comes before B_2 on a contig C of either one of M and S). Let F_1 and F_2 be, respectively, their frames on C.

Then, we compute the number of reads that have a correctly placed pair (or mate) that spans the gap between F_1 and F_2 and the number of reads that are expected to have their pair (or mate) correctly placed and crossing over F_1 and F_2 which is unmapped or mapped to a different sequence. In particular, a read r', mapped on a contig C, has a correctly placed pair (or mate) r'' if $begin(r'')$ is inside the region $[begin(r') + (m - 3 \cdot sd), begin(r') + (m + 3 \cdot sd)]$ and $|C| \geq begin(r') + (m + 3 \cdot sd)$, where m and sd are the mean and the standard deviation of the insert size of the library, respectively. Furthermore, we also compute values such as coverage and number of wrongly oriented pairs (or mates). These weights are used to determine the likelihood that a link represents a correct path allowing us to take motivated decisions in case of problematic regions witnessed by non-linear graphs (*i.e.*, bubbles, bifurcations, *etc.*).

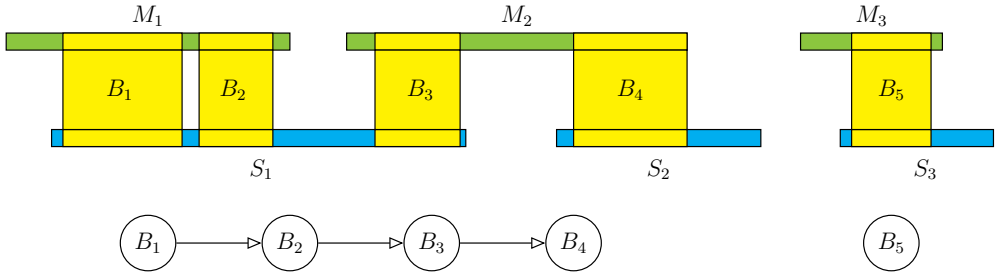


Figure 5.2: Example of AG construction. Let $M = \{M_1, M_2, M_3\}$ and $S = \{S_1, S_2, S_3\}$ be the master and slave assemblies, respectively. B_1 comes before B_2 in both S_1 and M_1 so a directed edge connects V_{B_1} and V_{B_2}. The same also applies for V_{B_2} and V_{B_3}, since B_2 comes before B_3 in S_1. Moreover, an edge is added between V_{B_3} and V_{B_4} as B_3 comes before B_4 in M_2.

Every path in AG corresponds to a sequence of blocks such that every pair of consecutive blocks lies on the same assembled sequence for at least one assembly. Thus, we can exploit AG to integrate or extend contigs.

Also, it is important to notice that if we consider AG disregarding edges' orientation, more than a single connected component can be present. We exploited this fact implementing GAM-NGS in a way that it can correct and merge contigs handling single connected components in parallel.

5.2.5 Handling problematic regions

Even if we build AG using the previously described method, block orders suggested by assemblies may contradict each other. For instance, suppose two blocks lie on a single contig in both the assemblies with opposite order with respect to M and S. This scenario will lead to a cycle in AG. Moreover, strongly connected components (SCCs) containing at least two nodes denote a situation where M and S disagree on the order of some blocks. To find these kind of contradictions we used Tarjan's algorithm [177] to determine SCCs in linear time while visiting AG.

Another possible problem is represented by divergent paths that may indicate situations where assemblies locally behaved differently: one assembler extended a sequence

in a different way with respect to the other. In particular, we can exploit edges' weights to perform choices that are locally optimal (*e.g.*, in the presence of a bifurcation the path minimizing the evidence of mis-assemblies will be chosen) in order to output a correct sequence. In situations where weights/features do not allow us to take a position (*e.g.* similar weights), we decided to be as conservative as possible, trusting only contigs belonging to the master assembly.

Among the various graph structures generated by discordant assemblies, *bubbles* and *forks* are the most common ones (see Figure 5.3 and Figure 5.4). Bubbles consist of a path that first diverges and then converges back. Forks, instead, contain only divergent or convergent paths. We can spot and distinguish these two structures with a simple depth-first traversal of AG. Such structures can nest in highly complex scenarios and, at this stage, we decided to deal only with graphs for which we have a good guarantee that they will be handled correctly. In particular, we took care only of cycles involving exactly two nodes and bifurcations not involving any bubble.

Handling cycles involving exactly two nodes

Cycles involving only two nodes may indicate inversions along the same contig in both M and S. To solve this particular kind of loop we can exploit mate-pair and pair-end reads' orientation. In [183] it has been shown how the use of mate-pair-happiness [141] is one of the best methodologies to detect mis-assemblies.

If the graph is indeed the result of two inverted blocks in one of the two assemblies, contigs pairs will be mapped with the correct orientation in only one of the two (see Figure 5.3). Hence, if we are able to find a minimum number of reads that are aligned properly in one contig and with the wrong orientation in the other one, we can include the correct sequence in the improved assembly. Otherwise, we chose to directly output the sequence of the master assembly.

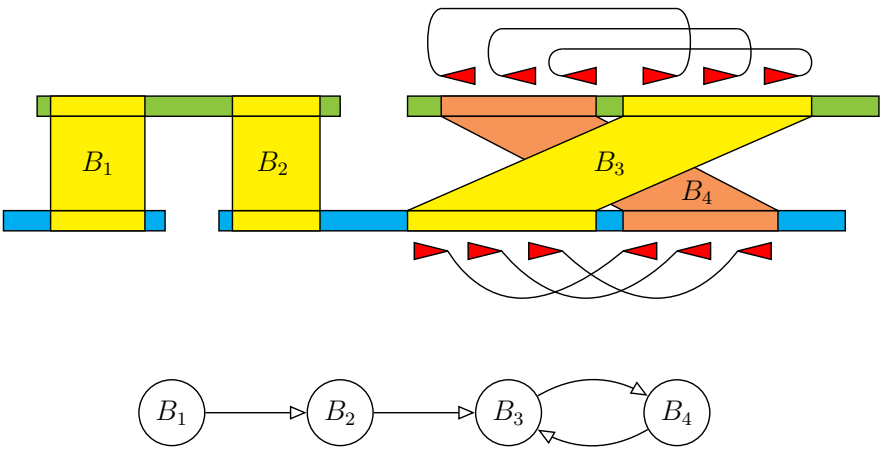


Figure 5.3: A 2-node cycle in AG witness a putative inversion along a single contig in M and S. If there actually is an inversion, then mate-pair reads are aligned with the wrong orientation in one of the two contigs. We can use this information to provide in output a correct sequence (the blue one in the picture).

Handling bifurcations

Graphs containing bifurcations may signify biological repeats or mis-assemblies. We will only show how we handle nodes with output degree equal to two, since nodes with input degree equal to two can be treated symmetrically. Let B be a block such that V_B has two outgoing edges to V_{B_M} and V_{B_S}. Let $M_i \in M$ be the contig shared between B and B_M, and $S_j \in S$ be the contig between B and B_S. In order to solve this scenario we focus on where reads placed on frames defined by B have their respective paired read (or mate): do they end up in B_M or B_S? See Figure 5.4 for an illustration of this case. Let n_M and n_S count the number of mates mapped to B_M's and B_S's frame, respectively. Given a read library with mean insert size m and standard deviation s, we define u_M (respectively u_S) as the number of reads mapped on the frame defined by B such that their pair/mate, accordingly to library orientation, is *not* aligned within a region of length $m + 3s$ (*i.e.*, insert size spanning) in B_M's frame on M_i (respectively, in B_S's frame on S_j). If M_i (or S_j) is so short that it is included within the insert size spanning of a read placement, then that read is not used to compute u_M (or u_S).

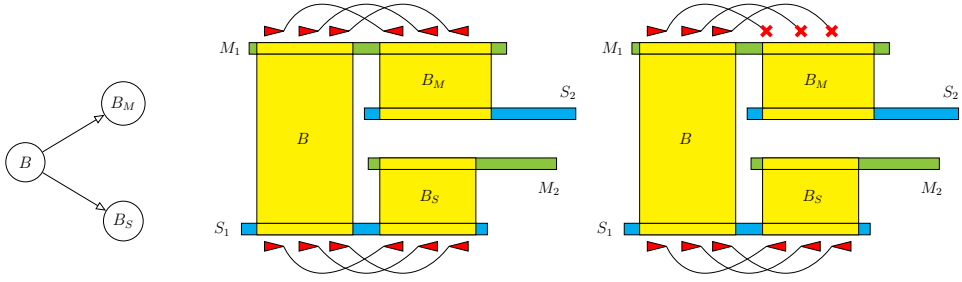
For instance, if we find that

$$\frac{n_M}{u_M} \geq T_U \quad \wedge \quad \frac{n_S}{u_S} \leq T_L,$$

where $T_U > T_L$ are two threshold values in $[0, 1]$, we may be able to spot a mis-assembly in S_j. Conversely, if we find that

$$\frac{n_S}{u_S} \geq T_U \quad \wedge \quad \frac{n_M}{u_M} \leq T_L,$$

we may be able to spot a mis-assembly in M_i, as in Figure 5.4b. If we are not in any of the two previous situations, it might mean that either blocks are too distant to let us discover the mis-assembly or B has been built due to a repetitive sequence. In this case, to avoid the introduction of errors in the improved assembly, we do not risk resolving the bifurcation and instead simply output the master's contigs.



(a) Biological repeat (b) Putative mis-assembly

Figure 5.4: Bifurcations in the Assemblies Graph may spot biological repeats or mis-assemblies. In panel (a), paired reads do not solve the bifurcation and we might face a biological repeat. In panel (b), paired reads on M_1 might help us to spot a mis-join in the assembly.

5.2.6 Merging

After solving problematic regions in AG, we can visit maximal disjoint paths in order to produce a draft alignment of contigs belonging to different assemblies. Such alignment is based on reads mapping and might be inaccurate (*e.g.*, regions having low identity). Therefore, we perform a semi-global alignment algorithm [180] (a banded variant to save memory) to make sure that contigs have a high similarity (*i.e.*, at least an identity of 95%) and should be merged.

We decided *not* to return a consensus, since there is no guarantee that it would be better than the two original sequences. Therefore, we decided to output the sequence belonging to the assembly that locally shows the best CE statistics [200] for insert sizes.

We also tried to avoid the introduction of duplicated regions, closing a gap between two contigs of M linked by a contig of S if and only if semi-global alignments on both ends of the region do not drop below 95% identity (see Figure 5.5).

After this phase, we obtain a set of *merged* contigs that we called *paired contigs*. To obtain the final improved assembly we simply output this set along with contigs of M that were not involved in any merge.

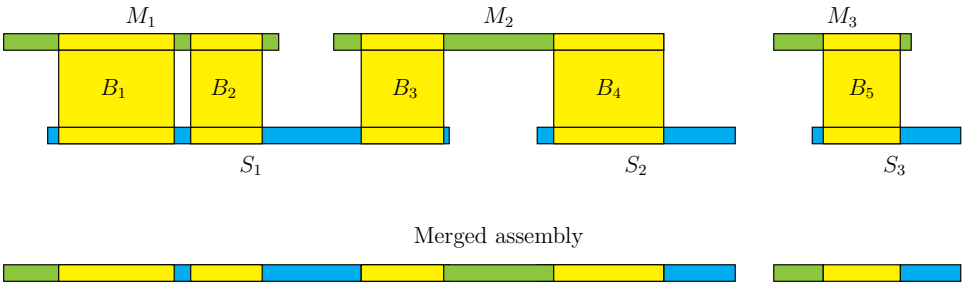


Figure 5.5: GAM-NGS merging example. During the merging phase, we fill the gaps between contigs in M and we extend a contig of M only if the corresponding sequence in S is longer and semi-global alignments at any end do not drop below 95% identity. Moreover, for regions defined by a block, we output the frame with better CE statistics.

5.3 Results

GAM-NGS's source code can be freely downloaded from [6]. It has been written in C++ and has been tested on Linux operating systems.

Validation of GAM-NGS's output has been performed on public data, for which results obtained by various assemblers are public as well. In particular, we chose three real datasets (*i.e.*, *Staphylococcus aureus, Rhodobacter sphaeroides* and human chromosome 14) downloaded from GAGE's website [5] (see Table 5.1) for which a reference genome is available. Moreover, we chose to test GAM-NGS on larger datasets such as *Prunus persica*, *Populus nigra* and *Picea abies*, in order to show our tool's scalability.

It is also important to point out that datasets provided by GAGE represent a useful instrument to evaluate GAM-NGS for a number of different reasons. First, GAGE provides state of the art datasets formed by several paired end and mate pairs libraries. Sec-

Table 5.1: Reference genomes and libraries of GAGE datasets.

Organism	Genome length (bp)	Library	Average read length (bp)	Insert size (bp)	Coverage
S. aureus	2,903,081	Fragment	101	180	29X
		Short jump	96	3500	32X
R. sphaeroides	4,603,060	Fragment	101	180	31X
		Short jump	101	3500	29X
Human chr14	88,289,540	Fragment	101	180	39X
		Short jump	96	3000	12X
		Long jump	96	35000	0.2X

ond, it provides highly reliable reference assemblies suitable for benchmarking. Third, a suite of reusable scripts is available for computing assembly metrics.

Reads available for each public dataset were error-corrected using both Quake and the Allpaths-LG error corrector. We chose to use the Allpaths-LG error-corrected reads.

Since GAM-NGS (as well as GAA) follows a master/slave approach and many assemblies are available for each GAGE datasets, we had to decide which assemblies should be merged and which should be elected as master.

Evaluating *de novo* assemblies in absence of a reference sequence is as difficult as *de novo* assembly itself. As an example, consider that Assemblathon 2 [26] required more than a year to evaluate submitted assemblies. GAGE datasets gave us the possibility to choose the two best assemblies accordingly to GAGE evaluation, however we decided to be as realistic as possible and to avoid the use of the available reference sequence. To the best of our knowledge, the only methodology available to evaluate assemblies in absence either of a reference sequence or of external-validation-data (*e.g.*, fosmid ends, physical maps, *etc.*) is based on Feature Response Curve-analysis (FRCurve-analysis) [183]. Recently, a novel tool dubbed FRC^{bam} [184], designed for computing a FRCurve from NGS-datasets, has been presented. Results summarized in [184] show that FRC^{bam} is able to effectively detect mis-assemblies. FRC^{bam} enabled us to evaluate a *de novo* assembly using only an alignment file (given in the standard SAM/BAM format) of a set of reads (usually the same reads used in the assembly), which is also the same input required by GAM-NGS.

For each GAGE dataset we plotted the FRCurve [183] using FRC^{bam}. Then we chose to merge the two assemblies having the steepest curves (*i.e.*, few negative features in the longest contigs) and whole length close to the genome size. As expected by the results shown in [184], we were always able to choose assemblies that, using GAGE's evaluation scripts, were characterized by good statistics such as number of errors and corrected NG50 (*i.e.*, NG50 of the assemblies broken in correspondence of each mis-assembly). All experiments were performed using both combinations of master/slave assemblies. We also decided to follow a common "bad practice" electing as best assemblies those characterized by the longest NG50 (without any consideration on the number of errors) and run GAM, GAA and ZORRO to merge them.

As far as the three larger datasets were concerned, we merged assemblies obtained

with CLC [9] and ABySS [162] for *Prunus persica* and *Populus nigra*, while we used GAM-NGS with a whole genome shotgun assembly and a series of fosmid-pools assemblies (all assembled with CLC assembler) for *Picea abies* that, to the best of our knowledge, represents one of the largest ever sequenced genome.

GAM-NGS's performance rely on the choice of several parameters: the minimum number of reads per block B_{min}, the threshold T_c related to blocksâĂŹ coverage filtering, the minimum block's length threshold T_l.

Low values of B_{min} increase the number of blocks which leads to a larger memory requirement and to a potentially more complex Assemblies Graph. Moreover, high values of T_c or T_l allow us to filter more blocks, running the risk of discarding significant blocks, while with low values we might keep blocks due to repeats that will complicate AG's structure. We decided to set $B_{min} = 10$, $T_c = 0.75$ and $T_l = 200$ bp for all experiments on bacteria. Instead, for human chr14, we set $B_{min} = 50$, $T_c = 0.75$ and $T_l = 500$ bp.

To evaluate correctness, we computed statistics using the same analysis script used in [154] and available for downloading at [5]. In particular, N50 sizes were computed based on the known size of the genome (NG50) and only contigs longer than 200 bp were used for the computations. As a consequence of the absence of a reference sequence in the case of the three new plants genomes we simply returned statistics showing the improvements in contiguity.

All experiments were performed on a 16 CPU machine with 128 GB of RAM, with the only exception of *Picea abies* where we used a machine equipped with 32 CPUs and 2 TB of RAM. GAM-NGS was always executed taking advantage of all available CPUs. GAA and ZORRO are designed as single-core programs. For this reason, we reported both CPU and wall clock times for each experiment. Moreover, GAA's internal call to BLAT is specified with the parameter `-fastMap` which requires input sequences to have contigs shorter than 5 Kbp. Thus, in each experiment, we had to manually run BLAT, providing its output to GAA's call. As we will show later, GAM-NGS was the fastest tool on the largest GAGE dataset (human chromosome 14).

Time of alignment was added to GAM-NGS's time but we would like to emphasize that read alignment is often required in downstream analyses and is also needed when FRC^{bam} [184] is used to evaluate assemblies' correctness.

5.3.1 Evaluation and validation on GAGE datasets

Given the availability of a reference sequence, GAGE datasets allowed us to compute the actual number of errors within an assembly. We compared GAM-NGS with GAA [193] and ZORRO [7] in order to obtain a comparison of assembly reconciliation tools as fair as possible and we used the same scripts used by Salzberg and colleagues in [154], downloadable from [5].

Staphylococcus aureus

For *Staphylococcus aureus*' dataset we chose to merge the assemblies of Allpaths-LG and MSR-CA. Looking at their FRCurves in Figure 5.6, they seem to be the best two assemblies for this dataset (SGA looks steeper, however its short contigs contains many issues according to our analysis). This situation is also confirmed by GAGE analysis,

as both Allpaths-LG and MSR-CA assemblies have a low number of errors and a large corrected NG50.

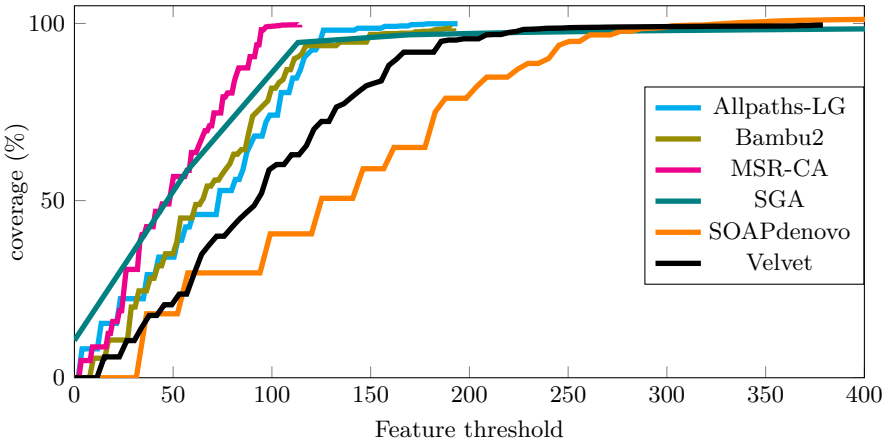


Figure 5.6: FRCurve of *Staphylococcus aureus* assemblies. Allpaths-LG and MSR-CA assemblies reach earlier a coverage close to 100% with the smallest number of features and, thus, they where chosen to be merged.

As shown in Table 5.2, using Allpaths-LG as master assembly, GAM-NGS was able to increase Allpaths-LG's NG50 by ∼40 Kbp and to decrease the number of compressed regions. Table 5.2 also shows us that GAA behaved better as far as compressed reference bases and corrected NG50 are concerned (GAA's corrected NG50 is ∼5Kbp longer than GAM-NGS one). However, GAA is affected by duplication events and, more importantly, Table 5.3 shows that it contains one misjoin more than GAM-NGS. ZORRO, instead, returned a lower NG50 (about half, compared to GAM-NGS and GAA) and a lower corrected NG50. Moreover, ZORRO's output contains more misjoins than GAM-NGS.

Using MSR-CA in place of Allpaths-LG as master assembly, GAM-NGS was able to increase NG50 by ∼30 Kbp and provide a better corrected NG50 with respect to the other tools. Moreover, GAM-NGS was able to correct the master assembly problematic regions, as GAM-NGS output as a lower number of misjoins than MSR-CA. GAA, instead, using MSR-CA as master assembly, performed better as far as compressed reference bases are concerned but returned a higher number of misjoins and indels compared to GAM-NGS. In this case ZORRO returned the minimum number of misjoins among the three tools but it is also the one with the assembly characterized by the lowest NG50 and the lowest corrected NG50.

In Tables A.1 and A.2 we summarize the results of merging the assemblies characterized by the largest NG50 (*i.e.*, Allpaths-LG and SOAPdenovo), without considering assemblies' correctness. The purpose of this test is to demonstrate how important the input assembly choice is. In particular, when using SOAPdenovo as master (*i.e.*, assembly with largest NG50) and Allpaths-LG as slave, all the three assemblies reconciliation tools return an assembly characterized by a corrected NG50 lower than master's one. Using Allpaths-LG as master, GAA and ZORRO returned a large number of dupli-

Table 5.2: GAGE statistics (contiguity, duplication and compression) on *Staphylococcus aureus*. For each assembler we report the number of contigs greater than 200 bp (Ctg), the NG50, the corrected NG50 (*i.e.,* computed breaking the assembly at each error), assembly's total length, the percentage of short (Chaff) contigs, the length of reference's regions which cannot be found in the assembly (Unaligned ref), the length of assembly's regions that cannot be found in the reference (Unaligned asm), the percentage of duplicated (Dupl) and compressed (Comp) regions in the assembly. All the percentages in the table are computed with respect to the true genome size.

Assembler	Ctg num	NG50 (kb)	NG50 corr. (kb)	Assembly size (%)	Chaff size (%)	Unaligned ref (%)	Unaligned asm (%)	Dupl (%)	Comp (%)
Allpaths-LG	60	96.74	66.23	98.88	0.03	0.61	0.01	0.04	1.26
MSR-CA	94	59.15	48.23	98.60	0.01	1.28	0.00	0.71	0.88
Allpaths-LG + MSR-CA									
GAM-NGS	44	141.54	75.82	100.49	0.00	0.44	0.01	0.26	0.99
GAA	40	139.48	80.68	99.52	0.03	0.37	0.01	0.32	0.88
ZORRO	81	74.68	62.85	99.70	0.16	0.32	0.04	0.59	0.88
MSR-CA + Allpaths-LG									
GAM-NGS	66	90.47	66.44	100.21	0.01	1.01	0.00	2.03	0.89
GAA	53	131.65	64.43	100.66	0.01	0.95	0.00	1.90	0.79
ZORRO	80	74.64	62.85	99.63	0.14	0.32	0.05	0.53	1.11

Table 5.3: GAGE statistics (SNPs, indels and misjoins) on *Staphylococcus aureus*. For each assembly we show the number of SNPs, the number of indels shorter than 5 bp and greater (or equal) than 5 bp. The number of misjoins is computed as the sum of inversions (parts of contigs reversed with respect to the reference genome) and relocations (rearrangements moving a contig within/between chromosomes).

Assembler	SNPs	Indels < 5 bp	Indels ≥ 5 bp	Misjoins	Inv	Reloc
Allpaths-LG	79	4	12	4	0	4
MSR-CA	191	23	10	13	6	7
Allpaths-LG + MSR-CA						
GAM-NGS	137	9	15	5	0	5
GAA	145	8	16	6	0	6
ZORRO	133	12	8	6	2	4
MSR-CA + Allpaths-LG						
GAM-NGS	214	19	10	9	2	7
GAA	206	22	15	11	2	9
ZORRO	262	24	9	7	4	3

Table 5.4: Assembly reconciliation tools performances on *Staphylococcus aureus*. In GAM-NGS's entries the first value indicates the time spent in alignment phase, while the second one is GAM-NGS's run time.

Tool	User (CPU) time	Wall clock time
Allpaths-LG + MSR-CA		
GAM-NGS	1h 10m 19s + 51s	4m 10s + 17s
GAA	1m 20s	1m 20s
ZORRO	3m 04s	3m 04s
MSR-CA + Allpaths-LG		
GAM-NGS	1h 10m 19s + 49s	4m 10s + 17s
GAA	1m 11s	1m 11s
ZORRO	14m 18s	14m 18s

cated regions (providing an assembly much longer than the reference) and they both introduced more misjoins than GAM-NGS.

Table 5.4 shows running times of the three assembly reconciliation tools. If we consider the CPU time, then GAM-NGS is definitely affected by the required reads alignment phase. Instead, if we consider wall time, GAM-NGS's performance is in line with the other tools.

Rhodobacter sphaeroides

For *Rhodobacter sphaeroides*' dataset we chose to merge Allpaths-LG and MSR-CA assemblies. Looking at their FRCurves in Figure 5.7, they seem the best two assemblies to be merged. CABOG and Bambus2 also provide sharp FRCurves on this dataset, however both assemblies are characterized by a large number of short contigs with many features (*i.e.*, long tail), and they both fail to fully assemble the genome, as the total assembly's length is approximately 90% of the expected one. For these reasons we discarded CABOG and Bambus2.

As shown in Table 5.5, using Allpaths-LG as master assembly, we were able to increase its NG50 by ∼10 Kbp. While GAA behaved better than GAM-NGS in terms of corrected NG50 as its value is ∼3Kbp longer, our tool behaved slightly better with consideration of duplication and compression events. Also in this case, ZORRO has worse performance among tested tools in terms of contiguity (both NG50 and corrected NG50). More importantly, Table 5.6 shows that both GAM-NGS and GAA were able to lower the number of misjoins, while ZORRO introduced a relocation.

When using MSR-CA as master assembly, GAM-NGS was able to increase MSR-CA's NG50 by ∼27 Kbp, providing a longer corrected NG50 with respect to the two merged assemblies. Also with this master/slave combination, GAA's assembly is characterized by a corrected NG50 slightly better than GAM-NGS's one. Both GAM-NGS and GAA introduced one additional misjoin with respect to MSR-CA, while ZORRO was able to correct the master assembly.

Tables A.4 and A.5 also show the results of merging the assemblies with the highest NG50 (*i.e.*, Bambus2 and SOAPdenovo). GAM-NGS and GAA have very similar statistics and for both of them the difference between the NG50 and its corrected value

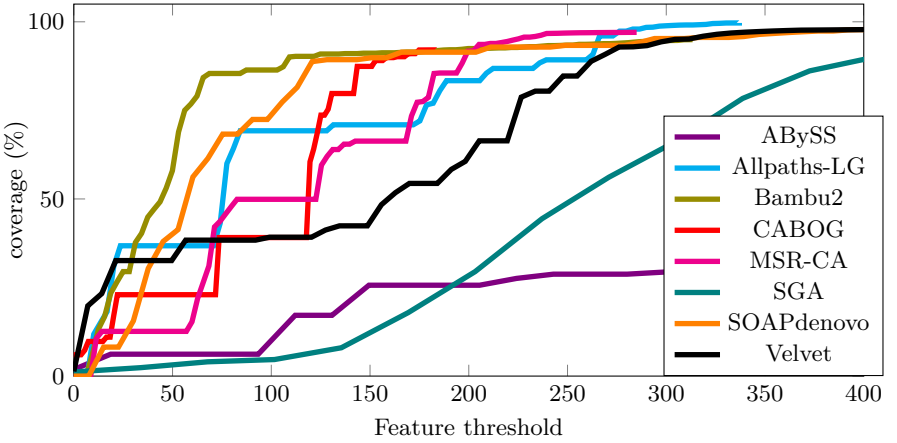


Figure 5.7: FRCurve of *Rhodobacter sphaeroides* assemblies. Allpaths-LG and MSR-CA assemblies reach earlier a coverage close to 100% with the smallest number of features and, thus, they where chosen to be merged. CABOG's assembly seems better but provides a low coverage of the genome and, for this reason, it was not taken into account.

Table 5.5: GAGE statistics (contiguity, duplication and compression) on *Rhodobacter sphaeroides*. Columns are the same as in Table 5.3.

Assembler	Ctg num	NG50 (kb)	NG50 corr. (kb)	Assembly size (%)	Chaff size (%)	Unaligned ref (%)	Unaligned asm (%)	Dupl (%)	Comp (%)
Allpaths-LG	204	42.45	34.42	99.68	0.01	0.45	0.01	0.38	0.31
MSR-CA	395	22.12	19.08	97.02	0.01	3.47	0.04	1.05	0.53
Allpaths-LG + MSR-CA									
GAM-NGS	168	51.12	37.88	99.97	0.00	0.28	0.01	0.61	0.31
GAA	164	53.82	40.55	100.07	0.01	0.20	0.01	0.63	0.32
ZORRO	216	38.87	30.64	100.41	0.03	0.36	0.02	0.43	0.48
MSR-CA + Allpaths-LG									
GAM-NGS	199	49.61	37.88	97.95	0.01	3.10	0.04	1.58	0.61
GAA	177	54.71	40.55	99.74	0.01	1.61	0.04	1.08	0.35
ZORRO	206	44.61	38.79	101.14	0.09	0.21	0.06	1.64	0.25

is substantial. ZORRO, instead, tends to output a highly fragmented assembly lowering the number of indels but without correcting any misjoin.

Table 5.7 shows running times of the three assembly reconciliation tools. Also in this dataset, if we consider the CPU time, then GAM-NGS is definitely affected by the required reads alignment phase and requires much more time than GAA and ZORRO. If we consider wall time, instead, GAM-NGS runs in less than 8 minutes, comparable, if not better, than the other tools.

Table 5.6: GAGE statistics (SNPs, indels and misjoins) on *Rhodobacter sphaeroides*. Columns are the same as in Table 5.4.

Assembler	SNPs	Indels < 5 bp	Indels ≥ 5 bp	Misjoins	Inv	Reloc
Allpaths-LG	218	150	37	6	0	6
MSR-CA	807	179	32	9	1	8
Allpaths-LG + MSR-CA						
GAM-NGS	250	157	44	5	0	5
GAA	345	162	48	5	0	5
ZORRO	263	153	35	7	0	7
MSR-CA + Allpaths-LG						
GAM-NGS	842	198	46	10	1	9
GAA	802	187	49	10	1	9
ZORRO	928	215	29	7	0	7

Table 5.7: Assembly reconciliation tools performances on *Rhodobacter sphaeroides*. In GAM-NGS's entries the first value indicates the time spent in alignment phase, while the second one is GAM-NGS's run time.

Tool	User (CPU) time	Wall clock time
Allpaths-LG + MSR-CA		
GAM-NGS	1h 21' 09" + 2' 20"	5' 03" + 43"
GAA	17"	17"
ZORRO	14' 46"	14' 46"
MSR-CA + Allpaths-LG		
GAM-NGS	1h 21' 09" + 2' 19"	5' 03" + 48"
GAA	19"	19"
ZORRO	16' 15"	16' 15"

Human chromosome 14

These first two bacteria datasets are small and time might not be considered an issue (each assembly reconciliation tool was able to run in reasonable time). The third GAGE dataset on which we tested our tool was the human chromosome 14 (characterized by an ungapped 88 Mbp size). This dataset is not only ∼20 times larger than the other two, but it is also more complex (*e.g.*, containing repeats, afflicted by heterozygosity). Moreover, in this scenario GAM-NGS starts to show its real potential: assembling large datasets using a relatively low amount of resources, while preserving correctness.

ZORRO output is not shown in Table 5.8 as, after two weeks of computation, it was not able to provide an output. Thus, we limit our evaluation to only GAM-NGS and GAA.

For this dataset we chose to merge Allpaths-LG and CABOG assemblies. Looking at their FRCurves in Figure 5.8, they are clearly the best two assemblies to be merged. GAGE's statistics also show that Allpaths-LG and CABOG assemblers produce the best two assemblies for this dataset (*i.e.*, highest NG50 and low number of misjoins).

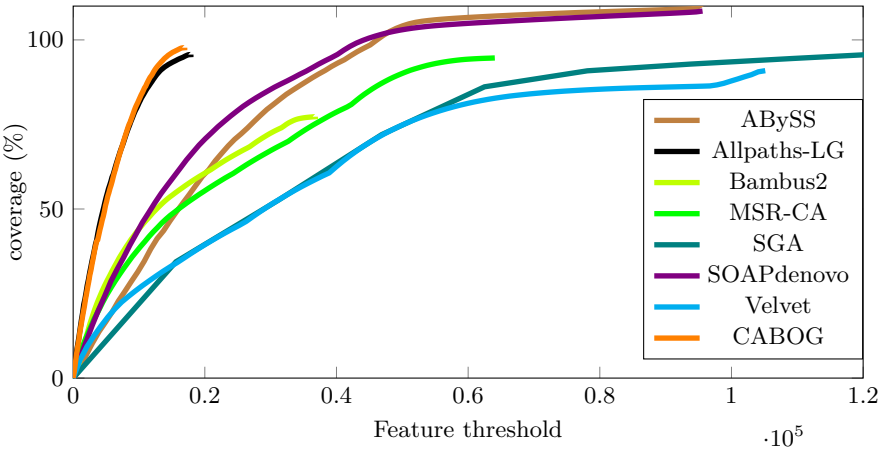


Figure 5.8: FRCurve of human chromosome 14 assemlies. Allpaths-LG and CABOG contain definitely the lowest numbers of features with respect to the other assemblers.

Table 5.8: GAGE statistics (contiguity, duplication and compression) on human chromosome 14. Columns are the same as in Table 5.3. All the statistics were computed using the same script with the gapped reference genome (107,349,540 bp).

Assembler	Ctg num	NG50 (kb)	NG50 corr. (kb)	Assembly size (%)	Chaff size (%)	Unaligned ref (%)	Unaligned asm (%)	Dupl (%)	Comp (%)
Allpaths-LG	4529	27.96	15.69	78.67	0.02	20.03	0.04	0.23	2.11
CABOG	3361	35.86	18.63	80.34	0.02	19.13	0.07	0.13	1.39
Allpaths-LG + CABOG									
GAM-NGS	2235	61.64	21.91	80.94	0.02	19.08	0.10	0.88	1.43
GAA	1989	69.40	23.04	82.08	0.02	18.92	0.09	1.52	1.39
CABOG + Allpaths-LG									
GAM-NGS	1979	66.29	23.63	81.00	0.02	19.00	0.06	0.74	1.37
GAA	1903	70.39	23.89	81.89	0.02	18.98	0.07	1.21	1.36

Tables 5.8 and 5.9 show how, using Allpaths-LG as master assembly, GAM-NGS was able to increase NG50 by $\sim$ 32 Kbp and the corrected NG50 by $\sim$ 6 Kbp. GAA returned better NG50 values but it produced more duplicated regions and it was afflicted by a larger amount of mis-joins and indels compared to GAM-NGS.

We also want to point out that the corrected NG50 is certainly an important statistic to evaluate the improvement of a merge with respect to the master assembly but it only indicates whether the longest contigs are affected by errors and does not tell how the assembler behaves on short contigs (which are also important to assess assemblies' quality, as FRCurve demonstrates). We finally plot the FRCurve to globally estimate the quality of the merged assemblies. Figure 5.9 shows that GAM-NGS globally behaved better and, in particular, seems to introduce less features (especially in the shortest contigs of the assembly).

Table 5.10 shows running times of the two assembly reconciliation tools used with

Table 5.9: GAGE statistics (SNPs, indels and misjoins) on human chromosome 14. Columns are the same as in Table 5.4. All the statistics were computed using the same script with the gapped reference genome (107,349,540 bp).

Assembler	SNPs	Indels < 5 bp	Indels ≥ 5 bp	Misjoins	Inv	Reloc
Allpaths-LG	55319	27563	2558	101	44	57
CABOG	81151	28438	2884	149	46	103
Allpaths-LG + CABOG						
GAM-NGS	61725	29936	2950	119	32	87
GAA	63835	30151	2990	123	29	94
CABOG + Allpaths						
GAM-NGS	79478	29653	3021	154	43	111
GAA	81763	29812	3008	134	31	103

Table 5.10: Assembly reconciliation tools performances on human chromosome 14. In GAM-NGS's entries the first value indicates the time spent in alignment phase, while the second one is GAM-NGS's run time. Due to the size of the assemblies, we parallelized BLAT's execution to get GAA's output in a reasonable time. ZORRO results are not shown due to the fact that the tool cannot run in parallel and, after more than a week of computation, was still not able to provide an output.

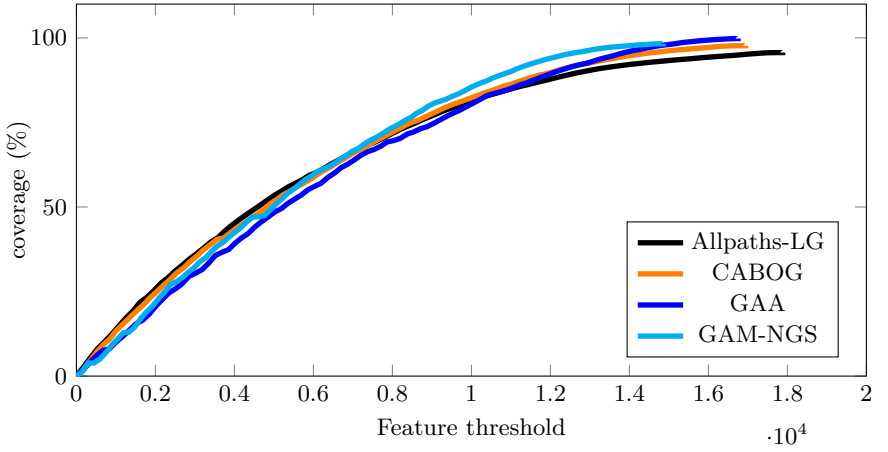
Tool	User (CPU) time	Wall clock time
Allpaths-LG + CABOG		
GAM-NGS	4h 24' 59" + 1h 14' 41"	45' 56" + 18' 16"
GAA	452h 18'	14h 16' 4"
CABOG + Allpaths-LG		
GAM-NGS	4h 24' 59" + 1h 12' 35"	45' 56" + 19' 21"
GAA	467h 40'	13h 44' 58'

this dataset. GAM-NGS required about 1 hour to accomplish its task (reads' alignments included), while GAA required about 13 hours (manually running multiple BLAT alignments in parallel).

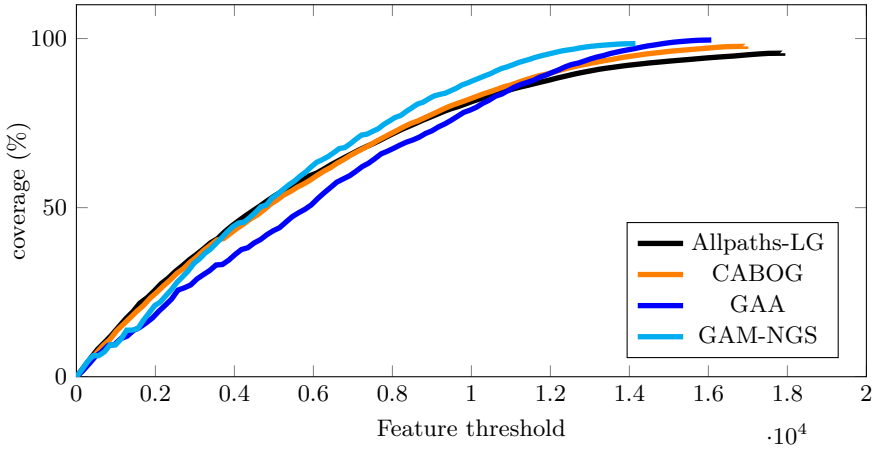
This characteristic may not be very important for short genomes but, as the size increases, it becomes of crucial importance. As we will show in the tests on some large plant genomes, GAM-NGS is able to merge even 20 Gbp assemblies using a relatively low amount of memory and time.

5.3.2 Performance of GAM-NGS on large datasets

On small datasets, all the assembly reconciliation tools provide an output in reasonable time. However, when we consider the human chromosome 14 we observe how GAA runs at least 10 times slower than GAM-NGS (if we consider also the mandatory reads' alignment step) while ZORRO, after two weeks, is not even able to provide us a partial output. This proves that the major bottleneck consists in the global alignment phase of these tools.



(a) Merging with Allpaths-LG as master.



(b) Merging with CABOG as master

Figure 5.9: FRCurve of assembly reconciliation tools on human chromosome 14, using (a) Allpaths-LG and (b) CABOG as master assembly. Despite the lower corrected NG50 (which means errors in the longest contigs), considering the whole assembly, GAM-NGS seems to behave globally better than GAA and the input assemblies.

On the contrary, GAM-NGS's approximation (using read's alignment back to the assemblies) coupled with the implementation of a weighted graph, achieves similar results in a reasonable amount of time. In order to show GAM-NGS's scalability, we tested it on three large plants genomes whose sizes vary from 227 Mbp to 20 Gbp.

The first of these datasets is *Prunus persica*, characterized by a genome size of 227 Mbp. The best assemblies we were able to compute were produced with CLC and ABySS assemblers, which were similar in length and number of contigs. We chose to use

Table 5.11: Contiguity statistics of GAM-NGS on large plants datasets.

Assembler	Total Length (Kbp)	Contigs	N50 (bp)
Prunus persica			
ABySS (M)	177,460	33,949	10,895
CLC (S)	179,151	41,684	8,654
GAM-NGS	**184,735**	**27,445**	**13,410**
Populus nigra			
CLC (M)	339,551	104,432	6,130
ABySS (S)	296,245	83,564	5,357
GAM-NGS	**359,795**	**78,366**	**10,018**

ABySS as master, since it was more contiguous. As shown in Table 5.11, we were able to increase NG50 (of ∼3 Kbp with respect to the master) and provide a more contiguous assembly compared to both CLC and ABySS. After mapping a 65× coverage of Illumina paired-end reads (which required 4 hours and 37 minutes), GAM-NGS took less than 2 hours using at most 19.6 GB of RAM.

The second large dataset we used is *Populus nigra*, characterized by a genome size of ∼423 Mbp. Also in this case, as for *Prunus persica*, the assemblies we had at our disposal were made with CLC and ABySS. This time, CLC's assembler looked better for its total length and NG50 and, thus, we decided to use it as master. As shown in Table 11, even with this dataset, we were able to increase NG50 (by ∼4 Kbp with respect to the master) and to provide a more contiguous assembly. To perform the mandatory alignment step we used a 80× coverage of Illumina paired-end reads, which required about 8 hours. Then, GAM-NGS took less than 4 hours using at most 34.5 GB of RAM to perform the merge. In order to save memory we could have decreased the reads coverage (at least 30× is suggested at the cost of a lower assembly improvement).

As a demonstration of GAM-NGS's flexibility, consider that GAM-NGS has also been used to obtain a draft assembly of the 20-gigabase genome of *Picea abies* (*i.e.*, the Norway spruce), where performing a global alignment is impracticable. GAM-NGS was able to run in less than 3 days (6 days, taking into account also the mandatory alignment phase) using at most 612 GB of RAM. This is certainly a low amount of resources, considering the dataset's size (almost a Terabyte) and that building a WGS assembly took one week and required more than 1 TB of RAM.

5.4 Conclusions

In this chapter we presented GAM-NGS: a *de novo* graph-based assembler which is able to merge assemblies using a relatively low amount of computational resources. Its strength relies on the fact that it does not need a global alignment to be performed and that makes its strategy unique among the other assembly reconciliation tools. In fact, GAM-NGS finds regions belonging to the same DNA *locus* using reads aligned back to the assembly, which is also an almost mandatory analysis step in all *de novo* assembly projects. The order in which these regions have been assembled is exploited to build

a locally weighted graph that GAM-NGS uses to fill gaps between sequences and to correct putative mis-assemblies. Moreover, mapping reads to the assemblies – without knowing how they have been placed by the assemblers – may lead to complex graph sub-structures (*e.g.*, bubbles, bifurcations, cycles) due to alignment errors or chimeric assembled sequences. Resolving these types of sub-graphs is far from being a trivial task, as in certain regions there may be lack of any possible evidence. In these kind of situations (which, for instance, represented 40% of the problematic cases for the human GAGE's dataset) we decided to be as conservative as possible, returning the sequences of one of the input assemblies (the one elected as master by the user).

The approach has been validated using GAGE [154] datasets, proving its effectiveness and reliability. Results showed that, for each GAGE dataset, GAM-NGS was always able to improve master assembly's NG50 and corrected NG50 (*i.e.*, NG50 of the assembly broken in correspondence of the errors), hence providing a globally more correct output (even if some errors were carried by the slave assembly). Although GAA provided better statistics in some cases, GAM-NGS gives comparable results and offers excellent scalability. As a matter of fact, GAM-NGS yields an improved assembly in reasonable time on large datasets (especially if used on a multi-core computer) for which competing tools are impractical. In particular, we showed GAM-NGS's scalability on large plant datasets (genome size up to 20 Gbp), where our tool required a low amount of computational resources compared to the dataset sizes and assembly requirements.

The presented algorithm performs a merge of two assemblies, returning the sequences of one of them in those problematic regions where we are not able to determine the most correct sequence between the two assemblies. We plan to investigate the use of further weights in AG that will allow us to solve more "difficult" regions, allowing us to completely replace the master-slave approach with a strategy that provides a more correct output.

We also plan to exploit GAM-NGS in a strategy thought to improve and correct a Whole Genome Shotgun assembly along with multiple sets of well assembled fosmid (or BAC) pools which constitute a hierarchically simplified version of the same genome.

III

Large-scale Genome Assembly

6

Hierarchical Assembly of Large Genomes

In the last decade sequencing costs have been continuously dropping down with the advent of NGS technologies. Nevertheless, as we already emphasized in the first two parts of this thesis, cost-effective technologies created new challenges for the *de novo* assembly problem, especially for large and complex genomes [11]. Indeed, resolving repeats longer than read length is often unfeasible, particularly in repeat-rich datasets. Several algorithms have been proposed to increase assembly's contiguity and correctness, though, the quality of the reconstructed sequences is often unsatisfactory for downstream analyses.

A recent approach, seeking for a trade-off between sequencing costs and assembly's quality, consists in sequencing long-insert DNA fragments (*e.g.*, fosmids and BAC clones) in pools using NGS technologies. In this way, since a pool represents just a small subset of the entire genome, the complexity of the assembly process highly decreases. In particular, compared to the canonical whole-genome-shotgun sequencing (where repeats are the main cause of fragmentation), a higher quality is expected in terms of sampling, repeats resolution, and allele reconstruction [11]. Also, pools introduce helpful constraints that can be exploited during reconciliation.

In Section 6.1 we present the methodology adopted in [134] to carry out the draft assembly of the 20-gigabase *Picea abies* (commonly known as the Norway spruce), which is also one of the largest species ever being sequenced. One of the distinctive hallmarks of the employed strategy (which is different from the one presented in the previous chapter) has been the application of GAM-NGS to *merge* and *improve* a WGS assembly with pool assemblies. The use of GAM-NGS was suitable due to the fact that input coverage of assembled pools was quite low and did not cover the entire genome length.

In Section 6.2 we outline a simple formalization – and generalization – of a method to address the hierarchical pool-based assembly. In particular, we assume that the available (assembled) sequences provide a higher coverage of the underlying genome (say, 4–8×). We also provide some preliminary results on a small-scale scenario. The method is however intended to be used on a spruce-like dataset.

6.1 The Norway spruce assembly

In order to assemble the 20-gigabase, repeat-rich, and heterozygous *Picea abies* genome, Nystedt *et al.* [134] developed a hierarchical strategy combining fosmid pools with both haploid and diploid whole-genome shotgun data (adopted also in [198]). The assembly process consisted of three major stages: fosmid pool assembly, hierarchical assembly, and validation.

6.1.1 Fosmid pool sequencing and assembly.

Fosmid libraries were constructed from a diploid tissue of the spruce and assigned to 450 pools. Based on a small preliminary study, consisting of just 5 pools of varying sizes, it was decided to generate approximately 1000 (random) fosmids per pool as a reasonable trade-off between sequencing time, cost, and assembly performance. Equivalently, we can say that each one of these sets covered approximately the 0.2% of the entire genome.

Each pool dataset was separately assembled starting from PE-read libraries of 300-base reads and using a proprietary assembler (*i.e.*, CLC [9]). Further scaffolding was performed using BESST [153] (a stand-alone tool specifically developed for this project) with two paired-read libraries with 625-base and 2.4-kilobase insert sizes.

The N50 and assembly size varied significantly among pools and because of this, while the sequenced libraries theoretically reflected a $1\times$ coverage, the assemblies represented approximately a $0.5\times$ coverage of the genome. Nevertheless, using the pool strategy allowed to obtain more assembled sequences longer than 10 Kbp compared to whole-genome assemblies of the haploid sequence.

6.1.2 Hierarchical assembly.

In order to obtain the first *Picea abies* draft reference, fosmid-pool assemblies were combined with both haploid and diploid whole-genome shotgun assemblies. Due to the fact that available assembly softwares were not able to cope with the integration such assembled pools in a satisfactory way [198], a *ad hoc* pipeline based on both GAM-NGS [185] and BESST [153] has been developed.

In order to apply GAM-NGS, the idea was to improve a whole-genome-shotgun (WGS) assembly $\mathcal{A}_{WGS}$ with the set of fosmid pool scaffolds $\mathcal{FP}$ sampled from the same genome. Each fosmid pool was sequenced and assembled separately using a $80\times$ coverage. Then, the $50\times$ coverage of Illumina reads used to assemble $\mathcal{A}_{WGS}$ has been mapped on both $\mathcal{A}_{WGS}$ and $\mathcal{FP}$ for the blocks construction phase. GAM-NGS was able to increase the assembly length by 1.4 Gbp with a NG50 that was 1.42 times greater than the one of the WGS assembly [134]. Fosmid-pool scaffolds unused by GAM-NGS and with no hit of more than 95% over 30% length against the merged assembly were also added to the output. The merged assembly reached a length of 12.0 Gbp, hence improving the original WGS assembly.

Finally, several paired-read libraries (with insert size from 300 bp to 10 Kbp) were aligned to GAM-NGS's output and scaffolded with BESST. This allowed to improve even more the contiguity of the sequence while also improving assembly's quality: the resulting output included 4.3 Gb in scaffolds longer than 10 Kbp.

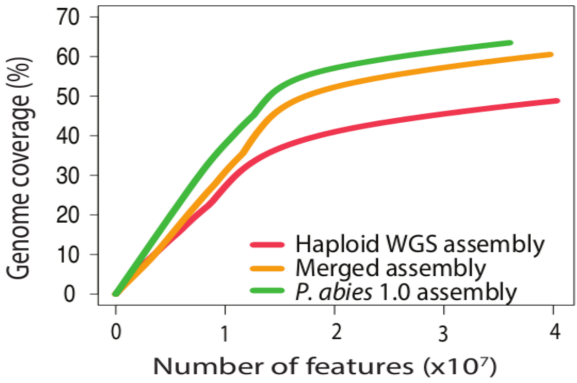


Figure 6.1: Feature-Response curves of Norway spruce assemblies [134]. The graph depicts, from the starting WGS assembly (red curve), the improvements achieved first thanks to GAM-NGS's merging (yellow curve) and, then, with scaffolding (green curve).

6.1.3 Assembly validation

The final assembly was validated using several metrics. Among them, the Feature-Response curve presented in Section 4.2.1 was used to evaluate the assembly after each step of the pipline. Figure 6.1 indicates clear improvements (*i.e.,* increasingly steeper curves) during both GAM-NGS's merging and the scaffolding stages. This supports the validity of the assembly strategy and also proves the efficacy of GAM-NGS in such a large-scale scenario.

As an additional proof of the benefits of the pool-based strategy, protein-coding fractions of the genome were analyzed using sequences from *Picea sitchensis* (or Sitka spruce) [148]. It was then estimated that approximately 63% of such protein-coding genes were fully covered ($> 90\%$ of their length), and 96% partially covered ($> 30\%$ of their length) within single scaffolds of the final assembly.

6.2 Hierarchical pool reconciliation

To the best of our knowledge, when facing either a large or a complex genome, where pool sequencing proved to be a viable and promising approach [134,198] and despite the progresses in sequencing long-insert pools, a standard methodology which systematically handles these kinds of datasets has not been proposed. Available assembly reconciliation tools such as GAA [193], GAM-NGS [185], and Mix [168], in fact, have not been designed for this kind of task. In particular, the first two are based on a master-slave approach and they are able to reconcile just two WGS assemblies at a time. The third one, instead, while being able to accept multiple assemblies as input, had been specifically thought for small bacterial genomes. Moreover, its negligence in dealing with mis-joins makes it unsuitable for the job.

For these reasons we propose a hierarchical scheme whose goal is to build a draft as-

sembly from multiple sets of independently assembled pools which form a hierarchically simplified version of a genome. Our idea relies on the effectiveness of the methods used to tackle two main sub-problems: overlap detection and merging of input sequences. For the first one, we depict a fingerprinting-based solution which let us to quickly carry out the task. We describe then a possible data structure and heuristics in order to deal with the second one. Some preliminary results are reported, obtained using a prototype tool we named *Hierarchical Assemblies Merger* (HAM). We want also to remark that this work might also take advantage of latest sequencing technologies, which produce long reads and are affected by high error rates.

6.2.1 Overview of the method

Let $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_m\}$ be a collection of assembled pools (in the following the adjective "assembled" may be omitted). Each $\mathcal{P}_i$ is the result of the *de novo* assembly of multiple long-insert fragments and it is supposed to be obtained by a state-of-the-art *de novo* assembler, using a high-coverage set $\mathcal{R}_i$ of short reads.

In order to guide the reconciliation, we are going to make use of two assumptions about pools. First, a pool covers a small percentage of the genome. Thus, two contigs $C_1, C_2 \in \mathcal{P}_i$ most likely *do not* belong to the same genomic *locus* (or at least we expect this would occur rarely), unless they represent the same insert that could not be entirely assembled. Second, a contig cannot be longer than the maximum size of the sequenced fragments ($\sim$ 40 Kbp for a fosmid and $\sim$ 100 Kbp for a BAC clone).

Our hierarchical assembly scheme can be naturally depicted as a binary tree $\mathcal{T}$ which recursively decompose the problem. $\mathcal{T}$ has m leaves and minimum height (*i.e.,* $\mathcal{O}(\log m)$). More precisely, the ith leaf is labeled $\mathcal{P}_i$, while an internal node corresponds to the result of the reconciliation of two or more pools and it is labeled $\mathcal{L}_1 \oplus \mathcal{L}_2$, where $\mathcal{L}_1$ and $\mathcal{L}_2$ identify the "partial" assemblies of its children and $\oplus$ is the merging operation. The root, thus, consists in the final assembly of the collection $\mathcal{P}$. $\mathcal{T}$ is also partitioned "vertically" with respect to the depth of its vertices. More precisely, we propose to consider three major classes of nodes, namely $\mathcal{A}_h$, $\mathcal{A}_m$, and $\mathcal{A}_l$, where different strategies could be applied (see Fig. 6.2).

$\mathcal{A}_h$ comprises nodes with highest depth. They correspond to the reconciliation of assemblies which still cover a quite small part of the genome. Thus, in this scenario we can afford to use a less sophisticated (even quadratic) algorithm for the overlap detection and a simple merging algorithm based on minimum length and identity. However, the more a node is closer to the root the more an unsophisticated/greedy method would be computationally expensive and error-prone.

For this reason we introduce $\mathcal{A}_m$ and $\mathcal{A}_l$, whose reconciliations are thought to be done with more efficient techniques. In the following sections we define an original method for $\mathcal{A}_m$ (valid also for $\mathcal{A}_l$) based on the construction of smaller objects – fingerprints – to be used in place of the entire contigs for overlap detection. We then formulate an open problem whose solution could improve the performances for $\mathcal{A}_l$. Finally, we present an algorithm based on the String Graph [122] used to carry out the actual merge.

In conclusion, we can summarize our hierarchical assembly approach in:

1. a pool pre-processing stage;

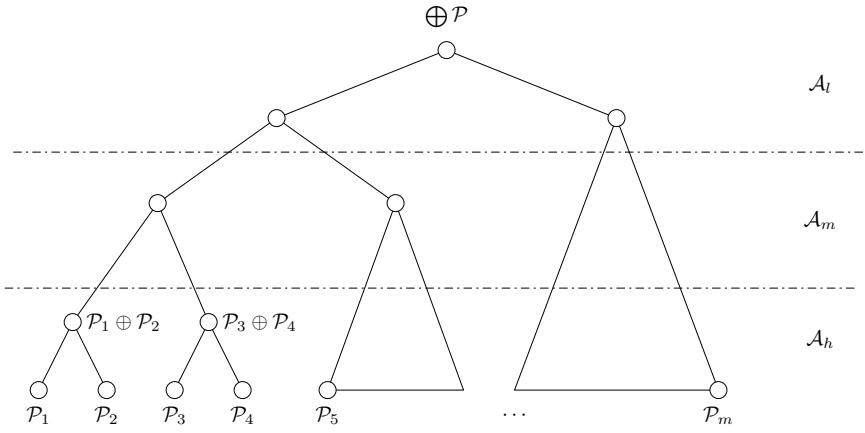


Figure 6.2: Hierarchical reconciliation of pools guided by a binary tree $\mathcal{T}$ with reconciliation strategies depending on node depth (*i.e.*, regions $\mathcal{A}_h$, $\mathcal{A}_m$, $\mathcal{A}_l$)

2. several reconciliation phases consisting in a depth-based strategy for overlap detection and merging (see Fig. 6.2).

6.2.2 Pool pre-processing

This phase is thought to be performed on leaves of $\mathcal{T}$. The goal is to filter poorly assembled pools, exploiting features (*e.g.,* paired reads mapping, k-mer analyses) which may spot errors in input contigs. However, we do not want to discard these sequences completely but we can try to integrate them in a later stage (as soon as a reliable draft assembly has been achieved).

An intuitive pre-processing procedure to find putative mis-assemblies consists in using a mapping of $\mathcal{R}_i$ against $\mathcal{P}_i$ in order to break contigs in regions showing a low physical coverage (*i.e.,* uncovered by paired-read inserts). This idea has already been exploited in tools like REAPR [81] and FRC^{bam} [184] – which also take advantage of other "bad-mapping" evidences – for correction and evaluation purposes, respectively.

Finally, a length threshold can be applied to keep only long-enough contigs. Since we expect good quality assemblies for most of the pools, for instance, we can set it to a fairly high value (*e.g.,* 5 Kbp).

6.2.3 Overlap detection

Due to the very small size of pools with respect to the genome, sequence similarity of two contigs from the same pool can, with highest likelihood, be attributed to paralogy rather than to an allelic difference. For this reason, we make the reasonable assumption that pools are unlikely to contain overlapping fragments. Thus, we can formulate the constraint that any pair of contigs from the same pool are distinct.

A simple approach to solve the problem could be performing an all-against-all alignment among the pools to be merged, retaining overlaps above user-defined length and

identity thresholds. Indeed, using this approach could be computationally demanding for large numbers of sequences and could be affordable only for merges in $\mathcal{A}_h$.

In order to perform the task in $\mathcal{A}_m$ and $\mathcal{A}_l$ one may think to use overlap detection algorithms of state-of-the-art assemblers. However, to the best of our knowledge, the presence of significant indels is not kept into account by them. More precisely, NGS assemblers usually expect *short good-quality* reads as input and not *long pre-assembled contigs*. The idea we explored consists in replacing each contig C with a k-mer-based *fingerprint* $\mathcal{F}(C)$ when seeking for sequences C' that are likely to overlap with C. This idea should help in quickly finding putative approximate overlaps between contigs. Two fingerprinting algorithm for this task will be analyzed in depth in Chapter 7.

6.2.4 A merging strategy

A reasonable structure to reconcile assemblies would certainly be the String Graph (SG) [122] which we already described in Section 3.3.1. We recall that the SG is a *bi-directed graph*, that is a graph where a directed head is attached to both ends of an edge. There are four kinds of edges and they correspond to the different types of overlaps [122]. We can thus define the *in-degree* of a vertex as the number of incident heads that point inwards and the *out-degree* as the number of incident heads that point outwards with respect to the vertex.

The SG construction is done as follows: a vertex is put for each contigs, while edges link overlapping pairs; transitive edges are removed; simple paths are collapsed in unitigs. The main idea of the SG is to represent sequences as vertices and prefix-suffix overlaps as edges. We expect that input sequences (*i.e.,* assembled pools) might contain different kind of errors: simple mismatches, insertions/deletions and misjoins. Thus, before connecting two vertices (contigs), we check whether their approximate overlap exceeds a user-defined identity threshold. The alignment is then retained only if the "overhangs" introduced by the overlap's intervals are short enough (*e.g.*, less than 100 bp). Moreover, fully included contigs are discarded and not represented as vertices. However, they will be used to compute the sequence coverage of the graph node in which they are included.

6.2.5 Graph simplification.

The SG is simplified removing transitive edges using Myers's algorithm [122] and unambiguous (*i.e.,* non-branching) paths can be collapsed into single nodes. At this point, we can take care of some graph topologies which may arise due to putative misjoins and small indels: bubbles and cut vertices adjacent to bifurcations.

Bubbles are characterized by two (or more) paths starting and ending at the same nodes. They are usually caused by small errors (*e.g.*, insertions/deletions) or biological variants. We can simply rely on algorithms used by state-of-the-art assemblers to take care of these structures. For instance, we may perform a linear visit of the graph (*e.g.,* depth-first) and when a bubble is found, if the identity of the branches is above a certain threshold, we retain only the path which is better covered and remove the other ones. Otherwise, if branches diverge too much, we do not remove any of them.

When an input sequence contains a mis-join such as a relocation (*i.e.*, regions far apart within the same chromosome are spliced together) or a translocation (*i.e.*, regions

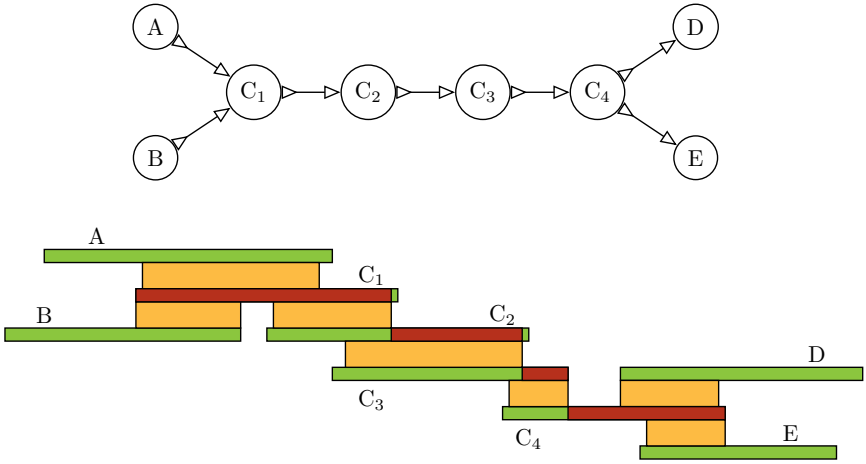


Figure 6.3: Example of the String Graph built in HAM. A possible consensus sequence is colored in red.

belonging to different chromosomes are joined) we can witness long almost-unambiguous paths which are connected by a single vertex (the contig containing the mis-join) which causes a bifurcation in each path. These vertices can bee seen as *cut vertices* (*i.e.,* their removal increases by one the number of connected components of the graph) with in and out degrees equal to 1 and adjacent to nodes characterized by a bifurcation.

Finally, an additional constraint could be exploited to identify consistent (or inconsistent) paths: giving the assumption that pool inserts have been sampled uniformly and independently, we expect contigs belonging to the same pool to be found far apart in a path and close enough contigs (*i.e.,* below the insert-size) are likely to represent the assembly of a single insert.

6.2.6 Consensus sequence.

Due to the use of approximate alignments to compute overlaps, we decided to output the sequence for each remaining vertex as follows. Each vertex corresponds to a simple unambiguous path in the former SG. Thus, we simply start from the first contigs in the path and we extend with the following one (see Figure 6.3). A better approach might be weighting the edges (possibly considering also transitive ones) and providing the best path according to a certain function.

6.3 Results

All the results are based on an early implementation of a tool we named Hierarchical Assemblies Merger (HAM) which performs the overlap detection and the merge strategies in a single step. The first one is carried out by the fingerprint-based overlap detection we will later depict in Chapter 7, while the second one takes advantage of the SG and the heuristics presented in Section 6.2.4.

Table 6.1: GAGE statistics of HAM on the human chromosome 14 (percentages refers to the ungapped reference genome size).

Ctg num	NG50 (kb)	Assembly size (%)	Unaligned reference (%)	Unaligned assembly (%)	Duplication (%)	Compression (%)
453	293	96.59	2.85	0.03	0.03	1.18

SNPs	Indels < 5 bp	Indels ≥ 5 bp	Mis-joins	Invertions	Relocations
4339	542	230	20	2	18

All the experiments were run on the same machine using 8 threads and the k-mer-based overlap detection algorithm has been tuned to consider k-mers with frequency lower (or equal) than 20 and to seek overlaps that are at least 1 Kbp long and with 95% identity.

A simulated datasets based on the 88-Mbp-long Human chromosome 14 has been built. More precisely, it consisted of a 8× coverage of 40-Kbp-long inserts randomly assigned to 353 pools (each one containing approximately 50 sequences and representing the 2.27% of the reference genome). We chose the insert size to follow a normal distribution with a mean of 40 Kbp and a standard deviation of 5Kbp. For each pool, we simulated a 42×-coverage PE-read library of Illumina fragments with (500 ± 25)-base-long insert-size and 100-base-long reads. For this task we chose the tool pIRS [77]. Then, in order to obtain good-quality assemblies, pools have been independently assembled using two state-of-the-art *de novo* assemblers: ABySS (version 1.5.2) and MaSuRCA (version 2.3.1).

We validated the assemblies of both tools using GAGE's validation script [154] against the available references for each pool. In this way, the choice of the assembler has been pretty clear. ABySS returned very contiguous assemblies with good quality metrics, while MaSuRCA returned results with also good quality metrics but they were more fragmented and presented on average a higher percentage of missing reference sequence.

ABySS has been first executed with mostly default parameters using a k-mer size of 71bp ($k = 71$), a higher maximum bubble length ($b = 1,000,000$), and a higher threshold for the unitig size required to build contigs ($s = 500$). A second run has been carried out with two additional parameters: a lower minimum alignment length of a read ($l = 1$ instead $l = k$) and a higher minimum sequence identity for a bubble ($p = 0.95$ instead of $p = 0.9$). Both executions achieved similar results as inversions and relocations are concerned. The second one, however, led to a significantly higher number of translocations (*i.e.,* rearrangements moving sequences between different inserts). Therefore, the first one has been selected.

We finally run HAM with the aforementioned mentioned parameters and, using again GAGE's validation script, we computed assembly correctness and contiguity statistics. As shown in Table 6.1, we were able to reconstruct most of the genome with a low number of mis-joins and with good contiguity statistics.

6.4 Remarks

This chapter introduced a novel approach to build a draft *de novo* assembly of complex genomes when a collection of well-assembled long-insert pools is available. Moreover, sequencing and assembling a collection of such pools has been proven to be a viable strategy for improving downstream analyses in several sequencing projects (*e.g.*, the Norway spruce and the Pacific oyster genomes). The main advantage is that pool assemblies are less likely hindered by repeats and allelic differences.

In order to exploit these kinds of datasets properly, we designed a strategy to perform reconciliation in a hierarchical manner. Specifically, we exploited a method based on fingerprints to carry out the overlap detection, while we relied on the String Graph to merge assemblies.

While still being a proof of concept, we were able to obtain promising preliminary results on a relatively small dataset based on the human chromosome 14. In the future, while improving the implementation of HAM to reflect precisely the scheme depicted in this work, our intent is to devise additional heuristics based on the particular input dataset. Specifically, an assembled contig should not exceed the length of the fosmid/BAC clone. Moreover, sequences belonging to different inserts of the same pool are expected to be far apart in the genome (unless they are close and short enough) and, hence, we do not expect to find them close in the graph.

7

Fingerprint-based Overlap Detection

As outlined in Chapter 6, the assembly of large and complex datasets could definitely benefit from the availability of longer sequences in order to disambiguate difficult regions of the genome (*e.g.*, repetitive region resolution) while increasing contiguity and correctness. There are two possibilities:

1. use third-generation sequencing technologies which produce long reads at the cost of a very high error rate (*e.g.,* up to 15% for PacBio).

2. sequence *and* assemble long-insert DNA fragments (*e.g.*, fosmids or BAC clones) in pools using second-generation technologies, where each pool represents a very small subset of the genome [11]. With this approach lower error rates are expected, however, the drawback is to deal with possible mis-assemblies.

The availability of long sequences allows to re-consider the OLC paradigm for the genome assembly problem, the main computational bottleneck of this approach being the detection of *all* pairwise overlaps between input sequences. A few state-of-the-art tools are available for the task of approximate long-sequence alignment. Some of those have been thought to align input sequences to a single reference (*e.g.*, BWA-MEM [94]), while others (*e.g.,* BLASR [34], DALIGNER [124], MHAP [21]) have been specifically designed to deal with the high error rates of TGS reads and may need huge amounts of computational resources (memory and/or time).

On this landscape we propose two novel methods (one being an improvement on the other) to effectively detect approximate overlaps among kilobase-long contigs, using *reasonable* amounts of computational resources. The idea we propose consists in (cleverly) *fingerprinting* contigs and the algorithms we describe are mainly thought to be used for the reconciliation (assembly) of assembled long-insert pools.

In Section 7.1 we start by describing a *local* non-deterministic strategy to compute, for a collection of input sequences, a set of k-mer-based *fingerprints* which are thought to be used to efficiently find overlaps using a fingerprint-vs-sequence approach.

In Section 7.2 we further improve the fingerprint construction using a *global* deterministic strategy. The method dispenses with the idea of mapping fingerprints to entire sequences (required by the "blindness" of the local choices) and, instead, it favors a more efficient fingerprint-vs-fingerprint approach that leads to find highly probable overlaps using even less computational resources compared to both the non-deterministic approach and available aligners.

7.1 A *local non-deterministic* approach

In this section we will propose a non-deterministic method to compute fingerprints using *local* choices of k-mers.

Let $\mathcal{S} = \{S_1, \ldots, S_N\}$ be a set of input sequences (*e.g.*, assembled contigs), where $S_i \in \Sigma^*$, $i = 1, \ldots, N$, and the alphabet is $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. Let $h : \Sigma^k \to \mathbb{N}$ be a function which maps each k-mer to its *global* frequency, *i.e.* its frequency computed with respect to all k-mers in $\mathcal{S}$. More precisely, being k_i^S the k-mer starting at $S[i]$,

$$h(x) = \left| \{k_i^S \mid S \in \mathcal{S} \ \wedge \ i \in [1, |S| - k + 1] \ \wedge \ x = k_i^S\} \right|.$$

The idea is to compute a significantly smaller structure $\mathcal{F}(S)$ to be used in place of $S \in \mathcal{S}$, which shall allow us to find overlaps with high probability. From now on, we will refer to $\mathcal{F}(S)$ as the fingerprint of S.

$\mathcal{F}(S)$ is built by wisely picking an ordered list of S's k-mers. We also assume the global frequency $h(k_i)$ to be available for every k-mer k_i in the dataset. The fingerprint computation problem is defined as follows.

Problem. Let S be a sequence, T_{freq} be a k-mer frequency threshold, and T_{gap} be a maximum distance threshold. We define n_{HF} as the number of k-mers $k_i \in S$ for which $h(k_i) > T_{freq}$ and n_{LF} as the number of k-mers $k_i \in S$ for which $h(k_i) \leq T_{freq}$. The problem is to seek for an ordered list of k-mers $\mathcal{F}(S) = \langle k_{i_1}, \ldots, k_{i_z} \rangle$ where $i_j \in \{1, \ldots, |S| - k + 1\}$, $i_1 < i_2 < \cdots < i_z$, and such that the following constraints are fulfilled:

- $i_{j+1} - i_j \leq T_{gap}$ for $j = 1, \ldots, z - 1$ (*i.e.*, two consecutive k-mers are not too far apart in S);
- n_{HF} is minimum;
- n_{LF} is minimum among those lists which minimize n_{HF}.

In other words, the problem is building a fingerprint which fulfills the gap constraint and minimizes the pair (n_{HF}, n_{LF}) in the lexicographic order.

The rationale behind this approach is to take as many low-frequency k-mers as possible while assuring that long sub-sequences are not left "uncovered". Keep in mind that these k-mers will be aligned and minimizing the number of highly frequent k-mers improves the performance.

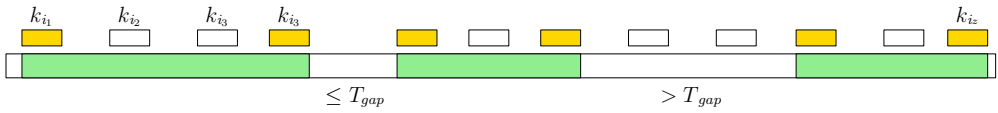


Figure 7.1: Example of fingerprint construction. Low-frequency regions are depicted with a green tint. First, yellow-colored k-mers are added to $\mathcal{F}(S)$. Second, remaining k-mers are chosen in order to fulfill the gap constraints (*i.e.,* $i_{j+1} - i_j < T_{gap}$). The output fingerprint is then $\mathcal{F}(S) = \langle k_{i_1}, \ldots, k_{i_z} \rangle$.

7.1.1 Fingerprint construction.

An approximate solution can be found linearly with two scans of the ordered list of S's k-mers. In the first one, we just pick those which are boundaries of *low-frequency regions* (*i.e.,* maximal sub-sequences comprising exclusively k-mers t such that $h(t) \leq T_{freq}$). In the second one, whenever two subsequent k-mers in $\mathcal{F}(S)$ violate the gap constraint, a minimal list of k-mers is added between them in order to satisfy the gap threshold. This solution, while not being optimal, has the advantage of providing the minimum number of high-frequency k-mers. It is also pretty straightforward and does not take much computational effort (see Figure 7.1).

7.1.2 Fingerprint-based overlaps detection.

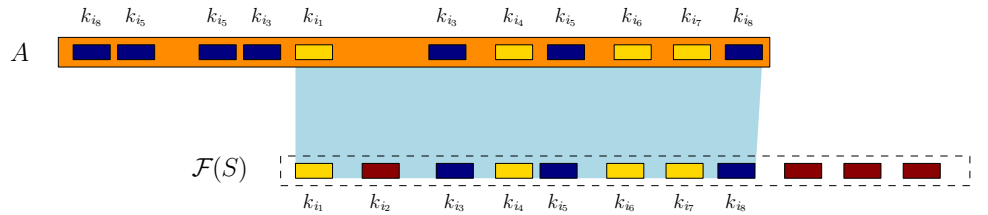
After the fingerprints are built, we map each k-mer in $\mathcal{F}(S)$ against the set of sequences S' for which we want to detect overlaps. Taking into account the distances between the mapped k-mers and their mapping order allows us to reduce the number of false positives (*i.e.,* sequences which do not overlap). S' can be indexed either using a db-Hash [145] or a FM-index [58]. The purpose of this mapping, however, is just to identify putative overlaps while reducing at the same time the number of exhaustive alignment computations (*i.e.,* performed using a banded Smith-Waterman algorithm).

First, we introduce a parameter c_{min}, which is the minimum number of shared k-mers required to check whether two contigs overlap. This parameter should be chosen in order to guarantee we are able to find true overlaps with high probability and a low false positive rate.

Second, we take into account the distances and the order of the mapped k-mer. Let A be a contig sharing at least c_{min} k-mers with a fingerprint $\mathcal{F}(S)$ and let $\mathcal{M}_A$ be a list of pairs (k_{i_j}, w), also referred as *hits*, such that $k_{i_j} \in \mathcal{F}(S)$ and w is the position where k_{i_j} occurs in A. After sorting $\mathcal{M}_A$ according to j (*i.e.,* the index of k_{i_j} in $\mathcal{F}(S)$), we seek for a long enough interval I of hits with the following constraints:

1. the k-mers of two consecutive hits reflect the order in $\mathcal{F}(S)$;
2. the actual number of k-mers in $\mathcal{M}_A$ between two *uniquely* mapped k-mers differs at most by 3 from the expected number (*i.e.,* the one in $\mathcal{F}(S)$);
3. all k-mers should be mapped with the same orientation.

Since I corresponds to a region in both A and S, we can think of it as an approximate overlap and we want to choose the one which minimizes the sum of the left and right



$$\mathcal{M}_A = \langle \underline{(k_{i_1}, w_5)}, (k_{i_3}, w_4), (k_{i_3}, w_6), \underline{(k_{i_4}, w_7)}, (k_{i_5}, w_2), (k_{i_5}, w_3), (k_{i_5}, w_8), \underline{(k_{i_6}, w_9)}, \underline{(k_{i_7}, w_{10})}, (k_{i_8}, w_1), (k_{i_8}, w_{11}) \rangle$$

Figure 7.2: Example of overlap detection. The k-mers which identify unique hits (underlined in $\mathcal{M}_A$) are colored in yellow, while those mapped in multiple position are colored in blue. Red-colored k-mers, instead, are absent in the sequence. The first interval computed is $[k_{i_1}, k_{i_7}]$, which is then extended with k_{i_8}. The approximate overlap reported consists of the sequences $S[i_1, i_8 + k - 1]$ and $A[w_5, w_{11} + k - 1]$.

tips of A and S (these tips are accounted with respect to I). An interval fulfilling these constraints is then extended from both ends considering also non-uniquely mapped k-mers (see Figure 7.2). This putative overlap is then assessed using Smith-Waterman algorithm and retained only if its length is greater than T_L and the identity exceed id_{min}, where T_L and id_{min} are two user-defined thresholds.

7.2 A *global deterministic* approach

The method depicted in Section 7.1 has two main limitations. First, it is necessary to build an index in order to efficiently map k-mers. Moreover, the index construction does not fit well in the hierarchical scheme presented in Section 6.2, as it might still demand some computational effort on large datasets in the last levels of the hierarchy (*i.e.*, $\mathcal{A}_l$ and $\mathcal{A}_m$). Second, overlaps are computed comparing fingerprints and sequences and this is forced by the *local* choices for $\mathcal{S}$.

We believed that a more clever fingerprint construction could be achieved in order to compare mere fingerprints and (in the first stage of the process) avoid the use of entire sequences. For this reason we came up with the idea of making a better use of k-mer frequencies in order to devise a "deterministic" algorithm that picks k-mers in a way that fingerprints of sequences with an actual overlap are likely to be built using the same seeds. This strategy solves both the aforementioned problems related to the *non-deterministic* approach: it does not require an index and it accelerates the computation of pairs of sequences which overlap (with high probability).

The algorithm presented in this section consists of three main stages:

i. after choosing a reasonably sized threshold k, we compute the *global* frequency $h(k_i)$ of each k-mer k_i with respect to all input sequences in $\mathcal{S}$;

ii. given a sequence S, the key idea is to cluster its k-mers in order to find regions consisting of successive k-mers whose (global) frequency remains sufficiently "stable";

iii. in each one of the regions identified in (ii.), we pick the k-mers that will constitute $\mathcal{F}(S)$ with a *deterministic* procedure.

The rationale behind this idea is that, chosen a proper value of k, the frequency of a k-mer in S is likely to reflect the number of overlapping sequences. In other words, the frequency of k-mers will remain (almost) constant until a new overlap starts (or ends). From that point on the frequency will be "stable" until another overlap starts (or ends). Therefore, in order to build fingerprints deterministically we can just keep track of these starting/ending points and – rigidly – choose between them.

As soon as the fingerprints have been built, we then seek for contig pairs whose fingerprints share a minimum amount of k-mers. For each such pair, we may then verify the presence of a proper overlap using classical dynamic programming approaches (*e.g.*, a banded Smith-Waterman algorithm).

7.2.1 An algorithm to build deterministic fingerprints

Definitions

Given $S \in \mathcal{S}$ and $f \in \mathbb{N}$, we define $\mathcal{I}_f^S$ as the set of maximal endpoints (l, r) such that *each* k-mer in $S[l, r]$ has global frequency f. Formally,

$$\mathcal{I}_f^S = \Big\{ (l,r) \;\Big|\; l, r \in [1, |S|] \;\wedge\; l + k \leq r \;\wedge\; (\forall i, j \in [l, r-k+1]) \\ h(k_i) = h(k_j) = f \;\wedge\; h(k_{l-1}) \neq h(k_l) \;\wedge\; h(k_{r-k+2}) \neq h(k_l) \Big\}.$$

$\mathcal{F}(S)$ will be determined by carefully choosing in an ordered list of its k-mers. We assume the global frequency $h(t)$ to be available for every k-mer t in the dataset.

Problem 7.1 (Deterministic Fingerprint Computation Problem (DFCP))**.** Given $S \in \mathcal{S}$ and $L \in \mathbb{N}$, the problem is to find an ordered list $\mathcal{F}(S) = \langle k_{i_1}, \ldots, k_{i_z} \rangle$ of k-mers in S, such that

- $1 \leq i_1 < \cdots < i_z \leq |S| - k + 1$ and
- if $A, B \in \mathcal{S}$ overlap by at least L bases, then $\mathcal{F}(A)$ and $\mathcal{F}(B)$ share at least a k-mer picked from the overlapping region.

Fingerprint construction.

Given a sequence S, we orderly process all its k-mers from the leftmost one. During this scan, $\mathcal{I}_f^S$ is built for $f > 2$. Then we consider the following set of k-mers as the fingerprint of S:

$$\mathcal{F}(S) = \{(k_i, i) \mid (p, q) \in \mathcal{I}_f^S \;\wedge\; q - p + 1 \geq T_L \;\wedge\; (i = p \;\vee\; i = q - k + 1)\},$$

where T_L is a user-defined threshold that specifies the minimum length of a region to be accounted in fingerprint construction. In words: the fingerprint is built choosing the first and last k-mer of every maximal stable frequency interval.

Assuming k has a value such that we have a fairly high probability that a k-mer is unique in the reference sequence (*i.e.* in the genome) and with low error rate, we expect two overlapping contigs $A, B \in \mathcal{S}$ to share most of the k-mers of $\mathcal{F}(A)$ and $\mathcal{F}(B)$ which can be found in the overlapping region.

A sketch of the fingerprint construction is depicted in Figure 7.3.

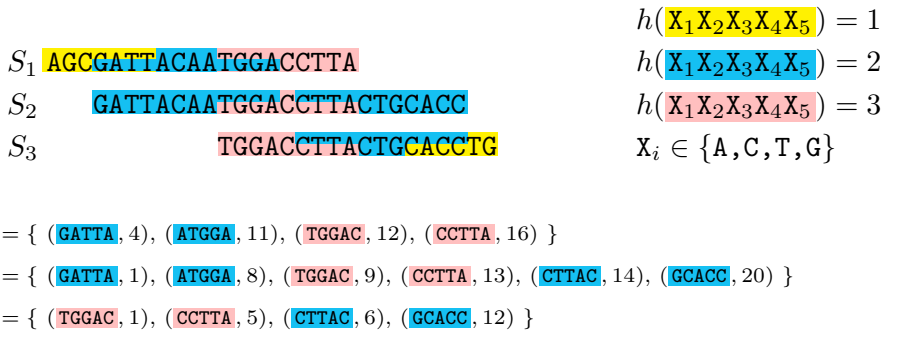


Figure 7.3: Deterministic fingerprint construction. In this example we assume there are three error-free sequences and that $k = 5$ is large enough (*i.e.*, each k-mer in the reference sequence is unique). Yellow colored regions correspond to $\mathcal{I}_1^{S_i}$ (for $i = 1, 3$), cyan colored regions correspond to $\mathcal{I}_2^{S_i}$ (for $i = 1, 2, 3$), and pink colored regions correspond to $\mathcal{I}_3^{S_i}$ (for $i = 1, 2, 3$). Notice that k-mers in fingerprints might also overlap in the full sequence.

The problem with the above fingerprint construction technique is that it does not take into account errors. In fact, the presence of erroneous bases, insertions, or deletions most likely will increase the cardinality of $\mathcal{I}_f^S$ and will lower the "size" of its elements (*e.g.,* modifying a single base in the region $S[p, q]$, where $(p, q) \in \mathcal{I}_f^S$, might split (p, q) into two sub-intervals (p, m), $(m + 2, q)$).

In order to diminish the chances of missing short intervals due to errors, we introduce a "second order" version of $\mathcal{I}_f^S$ in which neighboring elements at a distance below a predetermined threshold are *glued* into single intervals. Formally, we consider the following set:

$$\mathcal{I'}_f^S = \left\{ (l, r) \;\middle|\; \begin{array}{c} (l, p_1), (p_2, p_3), \ldots, (p_m, r) \in \mathcal{I}_f^S \text{ is a maximal sequence for which} \\ l < p_2 < p_4 < \cdots < p_m < r \;\wedge\; p_{i+1} - p_i \leq T_G \;\wedge\; i = 1, 3, \ldots, m-1 \end{array} \right\},$$

where $T_G \in \mathbb{Z}$ is the maximum distance allowed to join two consecutive intervals. This definition of $\mathcal{I'}_f^S$ might as well help in dealing with single k-mers whose frequency is unrelated with the mean coverage of a region (*e.g.*, a very frequent k-base long pattern in the genome that appears in a region represented by a number of sequences that reflect sequencing coverage).

Finding candidate sequences.

After building the set of all fingerprints $\mathcal{F}(\mathcal{S})$, we want to use it to find all the pairs (i, j) – where we assume, w.l.o.g., that $i < j$ – for which $\mathcal{F}(S_i)$ and $\mathcal{F}(S_j)$ share at

least a k-mer. However, a higher threshold for the minimum number of shared k-mer could be set. Briefly, we proceed in a way similarly to DALIGNER: we build a sorted table of k-mers to find pairs of candidate overlapping sequences (steps 1-3) and we apply additional steps to filter out putative false positives:

1. the list $\mathcal{K}_{\mathcal{F}(\mathcal{S})} = \{(k_i, a, i) \mid (k_i, i) \in \mathcal{F}(S_a)\}$ is built and sorted according to the following order relation

$$(k_i, a, i) < (k_j, b, j) \quad \Leftrightarrow \quad k_i \lhd k_j \vee (k_i = k_j \wedge a < b),$$

   where $\lhd$ stands for the lexicographical order among k-mers;

2. all tuples containing a k-mer which is *not* locally unique (*i.e.,* it is found multiple times in the same fingerprint) are discarded: $\mathcal{U}_{\mathcal{F}(\mathcal{S})}$ is the resulting list;

3. from $\mathcal{U}_{\mathcal{F}(\mathcal{S})}$ we can easily build the list $\mathcal{M}$ of tuples corresponding to pair of contigs whose fingerprints share a k-mer at a certain position. More precisely,

$$\mathcal{M} = \{(a, b, i, j) \mid (k_i, a, i), (k_j, b, j) \in \mathcal{U}_{\mathcal{F}(\mathcal{S})} \ \wedge \ k_i = k_j \ \wedge \ a < b\}.$$

   The list is lexicographically sorted on a, b, and i.

Approximate overlap detection.

Let $\mathcal{F}(S_a)$, $\mathcal{F}(S_b)$ be two fingerprints sharing m k-mers $k_{i_1}, \ldots, k_{i_m}$ and $k_{j_1}, \ldots, k_{j_m}$ from S_a and S_b respectively, where $i_1 < \cdots < i_m$ and $k_{i_z} = k_{j_z}$ for $z = 1, \ldots, m$. We also define $hp_z = \langle i_z, j_z \rangle$ as a *hit-pair*, where $k_{i_z} = k_{j_z}$. It might happen that shared k-mers do not strictly appear in the same order in S_a and S_b, due to errors or by chance. Thus, we need to seek for a proper sub-sequence of $hp_1, \ldots, hp_m$ such that (k-mer) indices appear in the same order both in S_a and S_b, and which likely identify an overlap. We will call this sub-sequence a *consistent chain of k-mers* or, simply, chain (see Figure 7.4).

A possible strategy to find a good chain could be computing the longest strictly increasing sub-sequence (LSIS) of $j_1, \ldots, j_m$. This, however, may not correspond to an actual overlap. Moreover, in contigs corresponding to repetitive parts of the genome, the suffix (or prefix) of a sequence may be similar to both the prefix and suffix of another one. Thus, finding just one chain might not be enough.

A different strategy, which is also the one we chose to adopt, is to keep a set of candidate chains $\mathcal{C}$ (initially empty) and process each hp_z in order to possibly extend an element of $\mathcal{C}$ or, otherwise, to build a new singleton chain. More precisely, we say that a chain $C = (hp_{z_1}, \ldots, hp_{z_t})$ can be safely extended by hp_{z_u} when the following two conditions apply:

1. $i_{z_t} < i_{z_u}$ and $j_{z_t} < i_{j_u}$ (*i.e.,* k-mers of hp_{z_u} are not strictly included in the region of the chain C);

2. the relative difference between the lengths of the regions of the chain $|i_{z_1} - i_{z_u} + k - 1|$ and $|j_{z_1}, j_{z_u} + k - 1|$ is at most 5%.

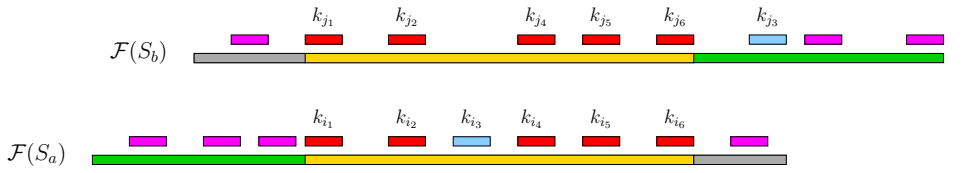


Figure 7.4: Example of fingerprint-based overlap detection. A consistent chain of k-mers might be $C = (hp_1, hp_2, hp_4, hp_5, hp_6)$ which is then used to roughly identify a putative overlapping region between S_a and S_b (the yellow colored one).

This way of building chains takes $\mathcal{O}(m^2)$ time. However, fingerprints are on average quite small (see Figure 7.5) and, therefore, also m will be small in the majority of the cases. We then filter out chains whose overhangs (*i.e.,* gray regions depicted in Figure 7.4) are longer than the 20% of the chain region's length and that also contain more than 5 fingerprint k-mers (which were not shared between the sequences). Finally, in order to compute overlap's end points, we extend retained chains using a banded Smith-Waterman alignment.

7.2.2 Implementation

In the description of the method we considered the DNA alphabet $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{T}, \mathtt{G}\}$. In practice, however, the input sequences may contain a certain amount of ambiguous characters (*e.g.*, gaps introduced during the scaffolding phase of the assembly). We identify any of them as the character `N` and we force $h(t) = 0$, for every ambiguous k-mer $t \in (\Sigma \cup \{\mathtt{N}\})^k \setminus \Sigma^k$.

The DNA is a double-stranded molecule: each strand is connected to a complementary one. Thus, in order to cope with overlaps between contigs belonging to different strands, the (global) frequency of a k-mer is computed considering its *canonical* representation (*i.e.*, the smaller string between the k-mer itself and its reverse complement in the lexicographic order). Moreover, the computation of frequencies has been carried out using Jellyfish's efficient hash-table implementation [108], which allowed us to complete this task exploiting parallelism and using a moderate amount of time and memory.

The "list" $\mathcal{K}_{\mathcal{F}(\mathcal{S})}$ is implemented using a vector in which a tuple of the form (k_i, y, i) is added for each $(k_i, i) \in \mathcal{F}(S_y)$. The vector is then sorted using the aforementioned lexicographic order relation in linearithmic time. Thus, $\mathcal{U}_{\mathcal{F}(\mathcal{S})}$ could be built in-place with a linear complexity in a single sweep of $\mathcal{K}_{\mathcal{F}(\mathcal{S})}$.

Finally, given a tuple $(k_i, a, i) \in \mathcal{U}_{\mathcal{F}(\mathcal{S})}$, for each subsequent $(k_j, b, j) \in \mathcal{U}_{\mathcal{F}(\mathcal{S})}$ where $k_i = k_j$, the tuple (a, b, i, j) is added to a vector which corresponds to $\mathcal{M}$. The expected time to carry out this task is linear in the number of true overlaps between input sequences. As we will show in the next section, the number of false positives is much lower than the number of overlaps found. $\mathcal{M}$ is then sorted accordingly in order to cluster shared k-mers between fingerprints.

7.2.3 Experimental results

The method we described has been implemented in C++11 and tested on Linux operating systems. A prototype of its implementation, hereafter named DFP, can be downloaded from [2].

Datasets

In order to test our method, DFP has been applied on four datasets of different size and type: *Escherichia coli* K12 MG1655 (4.6 Mbp), *Drosophila melanogaster* ISO1 ($\sim$ 130 Mbp), the human chromosome 14 (88 Mbp, ungapped length), and *Crassostrea gigas* ($\sim$ 637 Mbp). The four experiments were carried out in the two applicative scenarios for which the method is proposed. The first two datasets consisted of a 26$\times$ and a 10$\times$ coverage of corrected PacBio reads, respectively, and retrieved from [8]. The third and the fourth ones, instead, represented the pool-sequencing scenario. In particular, from the human chromosome 14 reference we extracted a 8$\times$ coverage of 40 Kbp inserts which were randomly assigned to 353 pools. For each pool, we simulated a 42$\times$ coverage paired-end read library of Illumina fragments with (500 ± 25)-bp insert size and 100-bp reads. Then, each library has been independently assembled using ABySS [162]. Finally, the *C. gigas* dataset was based on real data and consisted in a 4$\times$ coverage of contigs obtained from the assembly of 1600 fosmid pools. For each dataset, scaffolds were broken and only sequences longer than 5 Kbp were considered.

Output validation and DFP performance.

DFP's results have been compared against two state-of-the-art alignment tools: MHAP [21] (version 1.6) and DALIGNER [124] (version 1.0). To the best of our knowledge, we believe they are currently the most appropriate choice for detecting overlaps in our application scenario. The reader, however, should keep in mind that these two tools were designed to work with the high error rates of PacBio reads, while our method has been mainly designed for long high-quality sequences. BWA-MEM and BLASR were also considered, nevertheless, we discarded them from the comparison because they are primarily designed for mapping contigs/reads to a reference sequence and they are not well suited for a pair-wise comparison of sequences. In particular, the first one tends to discard too many true overlaps, while the second one requires a large amount of time as the dataset grows.

MHAP follows the same philosophy of DFP: build and compare fingerprints instead of sequences. More precisely, it is based on a probabilistic dimensionality reduction approach called MinHash [27] along with the computation of Jaccard similarity between fingerprints. DALIGNER, instead, potentially considers all k-mers in all input sequences, rather than the reduced set of MHAP and DFP. In order to detect putative overlapping sequences, it relies first on filtering repetitive k-mers and second on a parallel and cache-friendly radix sort. Subsequently, shared k-mers (seeds) are extended using a $\mathcal{O}(ND)$-time difference algorithm [121] to precisely compute overlaps.

We run each tool varying the k-mer size from 19 to 31 (from 15 to 27 for *E. coli*, due to the smaller genome size) and we sought for overlaps long at least 2 Kbp. *D. melanogaster* and *C. gigas*, due to the large size, were partitioned in 20 000-sequence sets to be sure tools could finish using at most 32 GB of RAM.

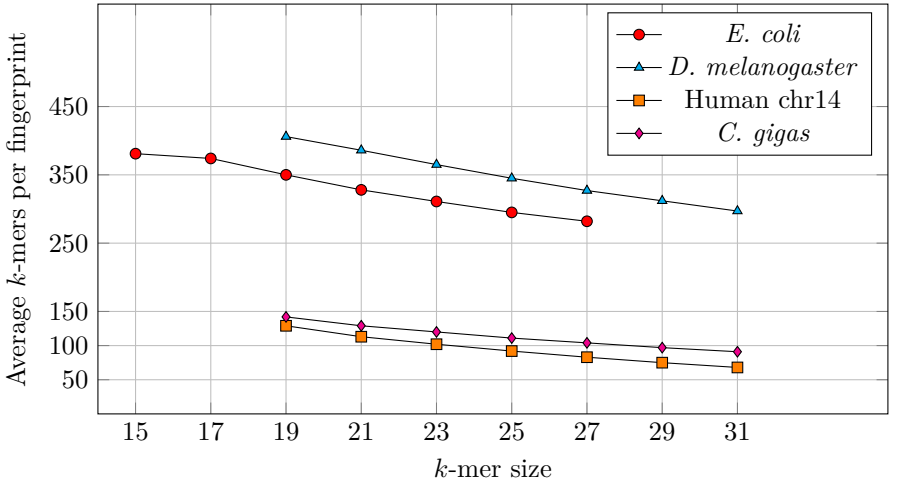


Figure 7.5: Average fingerprint size in function of k.

Parameters were set in order to achieve a good trade-off between required computational resources and output accuracy, and also taking into account that input sequences are expected to have high quality. DFP particularly relies on three parameters: k-mer size, T_L, and T_G (see Section 7.2.1). While the first one was varied accordingly to the dataset (to have a reasonable assurance for a k-mer to be almost unique in the genome), T_L and T_G where set to the default values of $2k + 1$ and -5 respectively, as they lead to a good balance between fingerprint size, speed, and precision (see Figure 7.5 and Table 7.1). MHAP was run increasing both the minimum number of matches and the similarity score cutoff as they affect the filtering and, thus, the output. DALIGNER was run with non-default parameters more suited for the alignment of assembled contigs and corrected reads. Moreover, its repetitive k-mer filter was tuned to use at most 30 GB of RAM (memory was limited to 16 GB for *D. melanogaster* as the tool crashed using higher bounds).

The performance of each tool was then evaluated comparing the output to a set of true overlaps inferred from the mapping of the input dataset to the reference genome. More precisely, we mapped input sequences against references using Nucmer (*E. coli* and Human chromosome 14) and BLASR (*D. melanogaster* and *C. gigas*), keeping only whole-sequence matches with identity greater or equal than 95%.

Sensitivity, specificity and positive predictive value (PPV) were then computed with the software used to evaluate MHAP and which is based on random sampling. In a nutshell, for a randomly chosen sequence, all other overlapping sequences are extracted from the reference matches. Every actual overlap is then counted as a true positive, while any missing one as a false negative. In order to estimate PPV, instead, a random overlap is compared to the reference mapping and, if the overlap is not deducible from the mapping, it is assessed using a local dynamic programming alignment. The PPV value is finally computed dividing the number of true overlaps for the number of overlaps evaluated. The sample size had been chosen in order to estimate PPV and specificity up to $\pm 1\%$.

Table 7.1: Overlap validation and performance of DFP, DALIGNER, and MHAP: sensitivity (TPR), specificity (SPC), positive predictive value (PPV), resident set size (RSS) peak in GB, wall clock time (WCT), and CPU time.

Data set	Tool	k	TPR	SPC	PPV	RSS	WCT	CPU
E. coli	DFP	17	99.6	100.0	100.0	3.21	59s	5m 08s
	DALIGNER	17	99.7	100.0	100.0	28.46	57s	11m 40s
	MHAP	19	100.0	100.0	99.5	9.82	1m 35s	24m 41s
D. melanogaster	DFP	19	98.1	100.0	99.5	10.03	16m 28s	2h 21m 7s
	DALIGNER	19	99.4	100.0	100.0	16.40	30m 03s	6h 34m 43s
	MHAP	19	99.0	100.0	100.0	18.85	32m 33s	8h 44m 3s
Human chr 14	DFP	31	99.8	100.0	100.0	1.51	1m 33s	16m 48s
	DALIGNER	31	100.0	100.0	100.0	18.99	9m 31s	2h 04m 45s
	MHAP	31	99.3	100.0	99.7	18.85	18m 13s	4h 57m 22s
C. gigas	DFP	29	99.8	100.0	99.9	8.95	25m 48s	4h 31m 54s
	DALIGNER	31	96.5	100.0	100.0	12.46	1h 09m 27s	17h 9m 0s
	MHAP	21	99.8	100.0	100.0	16.81	51m 30s	13h 45m 02s

Table 7.1 shows, for each tool, the statistics of the best execution (the others are reported in Appendix B). On all the datasets our method proved to be the fastest one, while also being the less memory greedy. This is due to the fact that, on average, our fingerprints are very small (see Figure 7.5). DFP's accuracy, however, is comparable with the other methods analyzed (see Table 7.1). DALIGNER, instead, achieved the best sensitivity on almost all datasets and was the second best tool when running time is concerned. Its performance, however, is tailored to its filtering phase and, therefore, on memory usage. In fact, adapting its filter to very low memory boundaries is likely to decrease its sensitivity (that is also the reason we decided to run it using almost all the 32 GB available). MHAP achieved also quite good statistics on all data-sets. The tool, however, is penalized by its Java implementation which is not optimal in terms of computational resources management. As also DFP does, it also pays a higher initialization cost with respect to DALIGNER. In particular, we believe that, if we did not impose memory limitations, it would use much less computational resources (both time and RAM) than DALIGNER and get closer to DFP's run time.

Conclusions

In this dissertation we propose three original contributions to the genome reconstruction problem: a whole-genome assembly reconciliation method, a hierarchical pool-based assembly strategy, and the efficient overlap detection between long sequences.

The first one deals with the merging *de novo* assemblies in order to improve assembly's quality. As a matter of fact, the availability of a large number of different heuristic methods paved the way to reconciliation techniques [32, 200] which were initially tailored to Sanger-based assemblies. For this reason, we developed GAM-NGS [185], a tool which extends this approach to be used regardless of the typology of dataset from which the input sequences have been built. In particular, our method effectively relies on the mapping of a read dataset against the input assemblies. Such a mapping, besides being used to identify regions representing the same genomic *locus*, allowed us to exploit the information coming from paired reads in order to identify putative mis-assemblies and, hence, improve assembly's correctness. Due to the particular effort done in its implementation, GAM-NGS has also been developed to carry out the reconciliation of very large assemblies using moderate amounts of computational resources. Furthermore, GAM-NGS has been used to aid the definition of the first draft sequence of the 20-Gbp Norway spruce genome [134]. At the moment, GAM-NGS is only able to merge two assemblies at a time. A further improvement would be to handle more assemblies (*e.g.*, iteratively) and introduce additional heuristics to resolve some complex graph structures which are not currently "untangled". In this way it would be possible to abandon the master-slave approach described in Chapter 5 and whose performance is strictly related to the choice of the input assemblies.

The second strategy we proposed in order to tackle the genome assembly problem (on a large scale) has been thought to exploit the specific methodology used to sequence the Norway spruce genome. Specifically, in order to deal with the abundance of repetitive regions, the genome has been sequenced in fosmid pools, each one assembled independently. In a scenario where a 4–8× coverage of such pools is available, GAM-NGS is no longer suitable for the task. For this reason, we devised a novel framework [186], which extends the assembly reconciliation idea in the case of a hierarchical sequencing of long-insert pools. The result is a prototype tool called HAM and based on an Overlap-Layout-Consensus (OLC) paradigm which leads to achieve promising results concerning assembly's quality on simulated pool-based datasets.

Finally, the third problem we faced is strictly related to the hierarchical reconciliation tool just mentioned. As a matter of fact, the critical part of every OLC-based assembler is the computation of overlaps among reads (or, more in general, sequences). In order to detect overlaps within large datasets we then defined a novel method which computes k-mer-based *fingerprints* of the input sequences and uses them to efficiently seek for

overlaps. The main advantage of this technique is to replace each sequence with its fingerprints in the identification of putative overlaps. We were able to show empirically that our method fits well in the hierarchical reconciliation framework as well as with corrected PacBio reads. Moreover, the method compares favorably (in terms of both running time and memory usage) with respect to other methods developed for the task of kilobase-read alignment. Unfortunately, the main limitation of our method is the need of good quality sequences in input and, for this reason, it is not suited to work with raw third-generation-sequencing reads. An interesting future perspective would be to understand whether we could extended our method to handle also error-rich datasets.

IV

Appendices

A

GAM-NGS supplementary tables

Table A.1: GAGE statistics (contiguity, duplication and compression) on *Staphylococcus aureus* of the merging between assemblies with the largest N50.

Assembler	Ctg num	NG50 (kb)	NG50 corr. (kb)	Assembly size (%)	Chaff size (%)	Unaligned ref (%)	Unaligned asm (%)	Dupl (%)	Comp (%)
Allpaths-LG	60	96.74	66.23	98.88	0.03	0.61	0.01	0.04	1.26
SOAPdenovo	107	288.18	62.68	100.55	0.34	0.22	0.02	1.66	1.45
Allpaths-LG + SOAPdenovo									
GAM-NGS	56	107.12	69.39	99.52	0.03	0.56	0.01	0.34	1.26
GAA	40	255.66	83.67	108.10	0.06	0.25	0.01	2.78	1.31
ZORRO	104	76.94	65.83	105.59	0.31	0.15	0.10	5.19	1.36
SOAPdenovo + Allpaths-LG									
GAM-NGS	93	288.18	62.68	100.92	0.32	0.20	0.02	1.88	1.40
GAA	74	294.96	62.87	101.92	0.34	0.16	0.02	2.62	1.37
ZORRO	107	76.94	62.68	105.63	0.29	0.16	0.09	5.17	1.50

Table A.2: GAGE statistics (SNPs, indels and misjoins) on *Staphylococcus aureus* of the merging between assemblies with the largest N50.

Assembler	SNPs	Indels < 5 bp	Indels ≥ 5 bp	Misjoins	Inv	Reloc
Allpaths-LG	79	4	12	4	0	4
SOAPdenovo	247	25	31	15	1	14
Allpaths-LG + SOAPdenovo						
GAM-NGS	88	5	14	4	0	4
GAA	100	9	19	10	1	9
ZORRO	227	19	12	6	1	5
SOAPdenovo + Allpaths-LG						
GAM-NGS	304	27	29	15	1	14
GAA	314	32	30	12	1	11
ZORRO	299	28	11	13	2	11

Table A.3: Assembly reconciliation tools performances on *Staphylococcus aureus* of the merging between assemblies with the largest N50. In GAM-NGS's entries the first value indicates the time spent in alignment phase, while the second one is GAM-NGS's run time.

Tool	User (CPU) time	Wall clock time
Allpaths-LG + SOAPdenovo		
GAM-NGS	1h 10m 53s + 33s	5m 12s + 24s
GAA	5m 04s	5m 04s
ZORRO	7m 08s	7m 08s
SOAPdenovo + Allpaths-LG		
GAM-NGS	1h 10m 53s + 34s	5m 12s + 25s
GAA	4m 49s	4m 49s
ZORRO	9m 52s	9m 52s

Table A.4: GAGE statistics (contiguity, duplication and compression) on *Rhodobacter sphaeroides* of the merging between assemblies with the largest N50. Columns are the same as in Table 5.3.

Assembler	Ctg num	NG50 (kb)	NG50 corr. (kb)	Assembly size (%)	Chaff size (%)	Unaligned ref (%)	Unaligned asm (%)	Dupl (%)	Comp (%)
Bambus2	177	93.19	12.78	94.97	0.00	4.92	0.00	0.00	0.24
SOAPdenovo	202	131.68	14.34	100.29	0.44	0.76	0.01	1.30	0.46
Bambus2 + SOAPdenovo									
GAM-NGS	83	149.75	14.16	98.32	0.00	3.02	0.00	1.59	0.63
GAA	100	194.16	14.74	98.35	0.13	2.28	0.01	0.63	0.58
ZORRO	711	16.56	13.18	100.48	0.89	0.66	0.25	1.05	0.59
SOAPdenovo + Bambus2									
GAM-NGS	177	154.47	15.17	100.41	0.42	0.82	0.01	1.67	0.48
GAA	174	188.18	14.54	100.35	0.44	0.76	0.01	1.38	0.48
ZORRO	720	16.56	12.78	100.48	0.84	0.69	0.24	1.14	0.56

Table A.5: GAGE statistics (SNPs, indels and misjoins) on *Rhodobacter sphaeroides* of the merging between assemblies with the largest N50. Columns are the same as in Table 5.4.

Assembler	SNPs	Indels < 5 bp	Indels ≥ 5 bp	Misjoins	Inv	Reloc
Bambus2	189	149	363	5	0	5
SOAPdenovo	534	155	404	8	0	8
Bambus2 + SOAPdenovo						
GAM-NGS	431	173	406	10	0	10
GAA	581	177	404	10	0	10
ZORRO	546	196	84	8	0	8
SOAPdenovo + Bambus2						
GAM-NGS	534	153	393	8	0	8
GAA	532	155	407	8	0	8
ZORRO	513	175	111	9	0	9

Table A.6: Assembly reconciliation tools performances on *Rhodobacter sphaeroides* of the merging between assemblies with the largest N50. In GAM-NGS's entries the first value indicates the time spent in alignment phase, while the second one is GAM-NGS's run time.

Tool	User (CPU) time	Wall clock time
Bambus2 + SOAPdenovo		
GAM-NGS	1h 26' 47" + 2' 35"	5' 53" + 1' 13"
GAA	3' 59"	3' 59"
ZORRO	8' 22"	8' 22'
SOAPdenovo + Bambus2		
GAM-NGS	1h 26' 47" + 2' 23"	5' 53" + 1' 09"
GAA	3' 47"	3' 47"
ZORRO	7' 44"	7' 44"

B

DFP supplementary tables

Table B.1: Overlap validation and performance of DFP, DALIGNER, and MHAP on the *E. coli* dataset for several values of k: sensitivity (TPR, True Positive Rate), specificity (SPC), positive predictive value (PPV), resident set size (RSS) peak, wall clock time (WCT), and CPU time. DALIGNER was run with the non-default parameters: `-e.9 -k`k `-w5 -h60 -s500 -M30`. MHAP was run with the non-default parameters: `-k` k `--num-threads 16 --num-min-matches 8 --threshold 0.16`.

Tool	k	TPR	SPC	PPV	RSS Peak	WCT	CPU
DFP	15	99.6	100.0	100.0	3.31 GB	58s	5m 08s
	17	**99.6**	**100.0**	**100.0**	**3.21 GB**	**59s**	**5m 08s**
	19	99.4	100.0	100.0	2.94 GB	53s	5m 03s
	21	99.3	100.0	100.0	2.67 GB	54s	5m 10s
	23	99.1	100.0	100.0	2.54 GB	50s	5m 16s
	25	98.9	100.0	100.0	2.35 GB	50s	5m 38s
	27	98.6	100.0	100.0	2.21 GB	47s	5m 07s
DALIGNER	15	99.7	100.0	100.0	29.56 GB	1m 58s	18m 28s
	17	**99.7**	**100.0**	**100.0**	**28.46 GB**	**57s**	**11m 40s**
	19	99.6	100.0	100.0	28.0 GB	55s	11m 25s
	21	99.6	100.0	100.0	27.6 GB	58s	11m 26s
	23	99.5	100.0	100.0	27.22 GB	57s	11m 19s
	25	99.5	100.0	100.0	26.86 GB	57s	11m 22s
	27	99.5	100.0	100.0	26.5 GB	56s	11m 17s
MHAP	15	100.0	100.0	99.5	9.86 GB	1m 35s	24m 42s
	17	100.0	100.0	99.4	9.86 GB	1m 36s	24m 53s
	19	**100.0**	**100.0**	**99.5**	**9.82 GB**	**1m 35s**	**24m 41s**
	21	100.0	100.0	99.4	9.77 GB	1m 39s	25m 55s
	23	100.0	100.0	99.4	9.83 GB	1m 38s	25m 40s
	25	100.0	100.0	99.4	9.84 GB	1m 38s	25m 47s
	27	100.0	100.0	99.4	9.81 GB	1m 38s	25m 42s

Table B.2: Overlap validation and performance of DFP, DALIGNER, and MHAP on the *D. melanogaster* ISO1 dataset for several values of k. Statistics were computed using the *Release 6* of the reference genome. Columns are the same as in Table B.1. DALIGNER was run with the non-default parameters: `-e.9 -k`k `-w5 -h60 -s500 -M16` (higher values of `-M` did not allow the program to terminate correctly). MHAP was run with the non-default parameters: `-k` k `--num-threads 16 --num-min-matches 5 --threshold 0.5`.

Tool	k	TPR	SPC	PPV	RSS Peak	WCT	CPU
DFP	**19**	**98.1**	**100.0**	**99.5**	**10.03 GB**	**16m 28s**	**2h 21m 7s**
	21	98.0	100.0	99.6	10.03 GB	16m 23s	2h 24m 8s
	23	97.8	100.0	99.7	8.80 GB	16m 02s	2h 23m 32s
	25	97.6	100.0	99.7	8.35 GB	15m 50s	2h 27m 11s
	27	97.4	100.0	99.5	7.92 GB	15m 44s	2h 25m 28s
	29	97.1	100.0	99.5	7.55 GB	15m 46s	2h 30m 26s
	31	96.8	100.0	99.7	7.23 GB	16m 19s	2h 44m 11s
DALIGNER	**19**	**99.4**	**100.0**	**100.0**	**16.40 GB**	**30m 03s**	**6h 34m 43s**
	21	99.4	100.0	100.0	16.40 GB	30m 07s	6h 32m 07s
	23	99.4	100.0	100.0	16.41 GB	29m 42s	6h 24m 50s
	25	99.4	100.0	100.0	16.41 GB	30m 23s	6h 32m 57s
	27	99.4	100.0	100.0	16.40 GB	30m 11s	6h 30m 17s
	29	99.3	100.0	100.0	16.41 GB	30m 47s	6h 34m 56s
	31	99.3	100.0	100.0	16.41 GB	30m 04s	6h 27m 42s
MHAP	**19**	**99.0**	**100.0**	**100.0**	**18.85 GB**	**32m 33s**	**8h 44m 3s**
	21	99.0	100.0	100.0	18.96 GB	32m 26s	8h 43m 07s
	23	99.0	100.0	100.0	18.75 GB	31m 37s	8h 29m 18s
	25	99.0	100.0	100.0	18.97 GB	32m 19s	8h 41m 54s
	27	99.0	100.0	100.0	18.97 GB	32m 03s	8h 37m 50s
	29	99.0	100.0	100.0	18.86 GB	31m 34s	8h 28m 57s
	31	99.0	100.0	100.0	19.00 GB	32m 30s	8h 44m 49s

Table B.3: Overlap validation and performance of DFP, DALIGNER, and MHAP on the Human chromosome 14 dataset for several values of k. Statistics were computed using the ungapped reference genome. Columns are the same as in Table B.1. DALIGNER was run with the non-default parameters: `-e.9 -k`k `-w5 -h60 -s500 -M30`. MHAP was run with the non-default parameters: `-k` k `--num-threads 16 --num-min-matches 8 --threshold 0.16`.

Tool	k	TPR	SPC	PPV	RSS Peak	WCT	CPU
DFP	19	99.0	100.0	99.8	1.92 GB	1m 41s	17m 2s
	21	99.2	100.0	99.9	1.58 GB	1m 37s	16m 56s
	23	99.4	100.0	100.0	1.50 GB	1m 37s	16m 44s
	25	99.5	100.0	100.0	1.38 GB	1m 35s	16m 41s
	27	99.7	100.0	100.0	1.40 GB	1m 34s	16m 40s
	29	99.7	100.0	100.0	1.46 GB	1m 35s	16m 51s
	31	**99.8**	**100.0**	**100.0**	**1.51 GB**	**1m 33s**	**16m 48s**
DALIGNER	19	100.0	100.0	100.0	23.43 GB	12m 57s	2h 50m 45s
	21	100.0	100.0	100.0	22.37 GB	11m 25s	2h 29m 29s
	23	100.0	100.0	100.0	21.51 GB	10m 22s	2h 14m 32s
	25	100.0	100.0	100.0	20.69 GB	10m 04s	2h 10m 47s
	27	100.0	100.0	100.0	20.05 GB	9m 54s	2h 09m 28s
	29	100.0	100.0	100.0	19.49 GB	9m 54s	2h 11m 36s
	31	**100.0**	**100.0**	**100.0**	**18.99 GB**	**9m 31s**	**2h 04m 45s**
MHAP	19	99.3	100.0	99.4	19.57 GB	18m 45s	5h 3m 57s
	21	99.3	100.0	99.4	19.21 GB	20m 34s	5h 34m 17s
	23	99.3	100.0	99.5	19.04 GB	20m 51s	5h 38m 37s
	25	99.3	100.0	99.6	18.90 GB	17m 37s	4h 46m 6s
	27	99.3	100.0	99.5	18.93 GB	18m 08s	4h 54m 29s
	29	99.3	100.0	99.6	18.93 GB	21m 05s	5h 42m 48s
	31	**99.3**	**100.0**	**99.7**	**18.85 GB**	**18m 13s**	**4h 57m 22s**

Table B.4: Overlap validation and performance of DFP, DALIGNER, and MHAP on the *C. gigas* dataset for several values of k. Statistics were computed using the ungapped reference genome. Columns are the same as in Table B.1. DALIGNER was run with the non-default parameters: `-e.9 -k`k `-w5 -h60 -s500 -M30`. MHAP was run with the non-default parameters: `-k` k `--num-threads 16 --num-min-matches 50 --threshold 0.5`.

Tool	k	TPR	SPC	PPV	RSS Peak	WCT	CPU
DFP	19	99.5	100.0	99.9	6.90 GB	27m 33s	4h 37m 9s
	21	99.6	100.0	99.9	7.27 GB	26m 49s	4h 31m 26s
	23	99.7	100.0	99.9	7.71 GB	26m 17s	4h 28m 02s
	25	99.7	100.0	99.9	8.08 GB	26m 11s	4h 30m 26s
	27	99.7	100.0	99.8	8.53 GB	25m 57s	4h 31m 23s
	29	**99.8**	**100.0**	**99.9**	**8.95 GB**	**25m 48s**	**4h 31m 54s**
	31	99.8	100.0	99.9	9.38 GB	25m 45s	4h 32m 29s
DALIGNER	19	96.5	100.0	100.0	18.89 GB	2h 07m 17s	1d 8h 28m 0s
	21	96.5	100.0	100.0	17.20 GB	1h 47m 29s	1d 3h 18m 8s
	23	96.5	100.0	100.0	15.86 GB	1h 39m 15s	1d 37m 19s
	25	96.5	100.0	100.0	14.74 GB	1h 28m 26s	22h 5m 18s
	27	96.5	100.0	100.0	13.80 GB	1h 23m 41s	20h 35m 37s
	29	96.5	100.0	100.0	12.98 GB	1h 16m 05s	18h 52m 19s
	31	**96.5**	**100.0**	**100.0**	**12.46 GB**	**1h 09m 27s**	**17h 9m 0s**
MHAP	19	99.8	100.0	100.0	16.79 GB	51m 47s	13h 49m 07s
	21	**99.8**	**100.0**	**100.0**	**16.81 GB**	**51m 30s**	**13h 45m 02s**
	23	99.8	100.0	100.0	16.84 GB	52m 11s	13h 56m 11s
	25	99.8	100.0	100.0	16.85 GB	52m 59s	14h 10m 00s
	27	99.8	100.0	100.0	16.90 GB	53m 05s	14h 10m 58s
	29	99.7	100.0	100.0	16.93 GB	53m 28s	14h 16m 28s
	31	99.7	100.0	100.0	16.91 GB	53m 52s	14h 23m 27s

Bibliography

[1] AMOS - A Modular Open-Source Assembler [http://amos.sourceforge.net].

[2] http://bitbucket.org/vice1987/fingerprint.

[3] http://code.google.com/p/sparsehash/.

[4] http://dazzlerblog.wordpress.com/2014/05/15/on-perfect-assembly/.

[5] http://gage.cbcb.umd.edu.

[6] http://github.com/vice87/gam-ngs.

[7] http://lge.ibi.unicamp.br/zorro/.

[8] http://www.cbcb.umd.edu/software/PBcR/MHAP.

[9] http://www.clcdenovo.com/.

[10] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53 – 86, 2004. The 9th International Symposium on String Processing and Information Retrieval.

[11] Andrey Alexeyenko, Björn Nystedt, Francesco Vezzi, Ellen Sherwood, Rosa Ye, Bjarne Knudsen, Martin Simonsen, Benjamin Turner, Pieter de Jong, Cheng-Cang Wu, and Joakim Lundeberg. Efficient *de novo* assembly of large and complex genomes by massively parallel sequencing of Fosmid pools. *BMC Genomics*, 15(1):439, 2014.

[12] C. Alkan, S. Sajjadian, and E.E. Eichler. Limitations of next-generation genome sequence assembly. *Nature methods*, 8(1):61–65, 2010.

[13] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403 – 410, 1990.

[14] S Anderson. Shotgun DNA sequencing using cloned DNase I-generated fragments. *Nucleic Acids Research*, 9(13):3015–3027, July 1981.

[15] Pramila Nuwantha Ariyaratne and Wing-Kin Sung. PE-Assembler: de novo assembler using short paired-end reads. *Bioinformatics*, 27(2):167–174, 2011.

[16] Anton Bankevich, Sergey Nurk, Dmitry Antipov, Alexey A. Gurevich, Mikhail Dvorkin, Alexander S. Kulikov, Valery M. Lesin, Sergey I. Nikolenko, Son Pham, Andrey D. Prjibelski, Alexey V. Pyshkin, Alexander V. Sirotkin, Nikolay Vyahhi, Glenn Tesler, Max A. Alekseyev, and Pavel A. Pevzner. SPAdes: A New Genome Assembly Algorithm and Its Applications to Single-Cell Sequencing. *Journal of Computational Biology*, 19(5):455–477, April 2012.

[17] Ali Bashir, Aaron A Klammer, William P Robins, Chen-Shan Chin, Dale Webster, Ellen Paxinos, David Hsu, Meredith Ashby, Susana Wang, Paul Peluso, Robert Sebra, Jon Sorenson, James Bullard, Jackie Yen, Marie Valdovino, Emilia Mollova, Khai Luong, Steven Lin, Brianna LaMay, Amruta Joshi, Lori Rowe, Michael Frace, Cheryl L Tarr, Maryann Turnsek, Brigid M Davis, Andrew Kasarskis, John J Mekalanos, Matthew K Waldor, and Eric E Schadt. A hybrid approach for the automated finishing of bacterial genomes. *Nat Biotech*, 30(7):701–707, July 2012.

[18] Serafim Batzoglou, David B. Jaffe, Ken Stanley, Jonathan Butler, Sante Gnerre, Evan Mauceli, Bonnie Berger, Jill P. Mesirov, and Eric S. Lander. ARACHNE: A Whole-Genome Shotgun Assembler. *Genome Research*, 12(1):177–189, 2002.

[19] David R. Bentley. Whole-genome re-sequencing. *Current opinion in genetics & development*, 16(6):545–552, December 2006.

[20] David R. Bentley, Shankar Balasubramanian, Harold P. Swerdlow, Geoffrey P. Smith, John Milton, Clive G. Brown, Kevin P. Hall, Dirk J. Evers, Colin L. Barnes, Helen R. Bignell, Jonathan M. Boutell, Jason Bryant, Richard J. Carter, R. Keira Cheetham, Anthony J. Cox, Darren J. Ellis, Michael R. Flatbush, Niall A. Gormley, Sean J. Humphray, Leslie J. Irving, Mirian S. Karbelashvili, Scott M. Kirk, Heng Li, Xiaohai Liu, Klaus S. Maisinger, Lisa J. Murray, Bojan Obradovic, Tobias Ost, Michael L. Parkinson, Mark R. Pratt, Isabelle M. J. Rasolonjatovo, Mark T. Reed, Roberto Rigatti, Chiara Rodighiero, Mark T. Ross, Andrea Sabot, Subramanian V. Sankar, Aylwyn Scally, Gary P. Schroth, Mark E. Smith, Vincent P. Smith, Anastassia Spiridou, Peta E. Torrance, Svilen S. Tzonev, Eric H. Vermaas, Klaudia Walter, Xiaolin Wu, Lu Zhang, Mohammed D. Alam, Carole Anastasi, Ify C. Aniebo, David M. D. Bailey, Iain R. Bancarz, Saibal Banerjee, Selena G. Barbour, Primo A. Baybayan, Vincent A. Benoit, Kevin F. Benson, Claire Bevis, Phillip J. Black, Asha Boodhun, Joe S. Brennan, John A. Bridgham, Rob C. Brown, Andrew A. Brown, Dale H. Buermann, Abass A. Bundu, James C. Burrows, Nigel P. Carter, Nestor Castillo, Maria Chiara E. Catenazzi, Simon Chang, R. Neil Cooley, Natasha R. Crake, Olubunmi O. Dada, Konstantinos D. Diakoumakos, Belen Dominguez-Fernandez, David J. Earnshaw, Ugonna C. Egbujor, David W. Elmore, Sergey S. Etchin, Mark R. Ewan, Milan Fedurco, Louise J. Fraser, Karin V. Fuentes Fajardo, W. Scott Furey, David George, Kimberley J. Gietzen, Colin P. Goddard, George S. Golda, Philip A. Granieri, David E. Green, David L. Gustafson, Nancy F. Hansen, Kevin Harnish, Christian D. Haudenschild, Narinder I. Heyer, Matthew M. Hims, Johnny T. Ho, Adrian M. Horgan, Katya Hoschler, Steve Hurwitz, Denis V. Ivanov, Maria Q. Johnson, Terena James, T. A. Huw Jones, Gyoung-Dong Kang, Tzvetana H. Kerelska, Alan D. Kersey, Irina Khrebtukova, Alex P. Kindwall, Zoya Kingsbury,

Paula I. Kokko-Gonzales, Anil Kumar, Marc A. Laurent, Cynthia T. Lawley, Sarah E. Lee, Xavier Lee, Arnold K. Liao, Jennifer A. Loch, Mitch Lok, Shujun Luo, Radhika M. Mammen, John W. Martin, Patrick G. McCauley, Paul McNitt, Parul Mehta, Keith W. Moon, Joe W. Mullens, Taksina Newington, Zemin Ning, Bee Ling Ng, Sonia M. Novo, Michael J. O'Neill, Mark A. Osborne, Andrew Osnowski, Omead Ostadan, Lambros L. Paraschos, Lea Pickering, Andrew C. Pike, Alger C. Pike, D. Chris Pinkard, Daniel P. Pliskin, Joe Podhasky, Victor J. Quijano, Come Raczy, Vicki H. Rae, Stephen R. Rawlings, Ana Chiva Rodriguez, Phyllida M. Roe, John Rogers, Maria C. Rogert Bacigalupo, Nikolai Romanov, Anthony Romieu, Rithy K. Roth, Natalie J. Rourke, Silke T. Ruediger, Eli Rusman, Raquel M. Sanches-Kuiper, Martin R. Schenker, Josefina M. Seoane, Richard J. Shaw, Mitch K. Shiver, Steven W. Short, Ning L. Sizto, Johannes P. Sluis, Melanie A. Smith, Jean Ernest Sohna Sohna, Eric J. Spence, Kim Stevens, Neil Sutton, Lukasz Szajkowski, Carolyn L. Tregidgo, Gerardo Turcatti, Stephanie VandeVondele, Yuli Verhovsky, Selene M. Virk, Suzanne Wakelin, Gregory C. Walcott, Jingwen Wang, Graham J. Worsley, Juying Yan, Ling Yau, Mike Zuerlein, Jane Rogers, James C. Mullikin, Matthew E. Hurles, Nick J. McCooke, John S. West, Frank L. Oaks, Peter L. Lundberg, David Klenerman, Richard Durbin, and Anthony J. Smith. Accurate whole human genome sequencing using reversible terminator chemistry. *Nature*, 456(7218):53–59, November 2008.

[21] Konstantin Berlin, Sergey Koren, Chen-Shan Chin, James P Drake, Jane M Landolin, and Adam M Phillippy. Assembling large genomes with single-molecule sequencing and locality-sensitive hashing. *Nature Biotechnology*, 2015.

[22] Ewan Birney. Assemblies: the good, the bad, the ugly. *Nature methods*, 8(1):59–60, January 2011.

[23] Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM*, 13(7):422–426, July 1970.

[24] Marten Boetzer, Christiaan V. Henkel, Hans J. Jansen, Derek Butler, and Walter Pirovano. Scaffolding pre-assembled contigs using SSPACE. *Bioinformatics*, 27(4):578–579, 2011.

[25] Jayson Bowers, Judith Mitchell, Eric Beer, Philip R Buzby, Marie Causey, J William Efcavitch, Mirna Jarosz, Edyta Krzymanska-Olejnik, Li Kung, Doron Lipson, Geoffrey M Lowman, Subramanian Marappan, Peter McInerney, Adam Platt, Atanu Roy, Suhaib M Siddiqi, Kathleen Steinmann, and John F Thompson. Virtual terminator nucleotides for next-generation DNA sequencing. *Nat Meth*, 6(8):593–595, August 2009.

[26] Keith Bradnam, Joseph Fass, Anton Alexandrov, Paul Baranay, Michael Bechner, Inanc Birol, Sebastien Boisvert, Jarrod Chapman, Guillaume Chapuis, Rayan Chikhi, Hamidreza Chitsaz, Wen-Chi Chou, Jacques Corbeil, Cristian Del Fabbro, T Docking, Richard Durbin, Dent Earl, Scott Emrich, Pavel Fedotov, Nuno Fonseca, Ganeshkumar Ganapathy, Richard Gibbs, Sante Gnerre, Elenie Godzaridis, Steve Goldstein, Matthias Haimel, Giles Hall, David Haussler, Joseph Hiatt, and

Isaac Ho. Assemblathon 2: evaluating de novo methods of genome assembly in three vertebrate species. *GigaScience*, 2(1):10, 2013.

[27] Andrei Z. Broder. On the Resemblance and Containment of Documents. In *In Compression and Complexity of Sequences (SEQUENCES'97)*, pages 21–29. IEEE Computer Society, 1997.

[28] Jeremy Buhler, Uri Keich, and Yanni Sun. Designing seeds for similarity search in genomic DNA. *Journal of Computer and System Sciences*, 70(3):342 – 363, 2005. Special Issue on Bioinformatics {II}.

[29] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, 1994.

[30] Jonathan Butler, Iain MacCallum, Michael Kleber, Ilya A. Shlyakhter, Matthew K. Belmonte, Eric S. Lander, Chad Nusbaum, and David B. Jaffe. ALLPATHS: *De novo* assembly of whole-genome shotgun microreads. *Genome Research*, 18(5):810–820, 2008.

[31] A. Califano and I. Rigoutsos. FLASH: a fast look-up algorithm for string homology. In *Computer Vision and Pattern Recognition, 1993. Proceedings CVPR '93., 1993 IEEE Computer Society Conference on*, pages 353–359, Jun 1993.

[32] Alberto Casagrande, Cristian Del Fabbro, Simone Scalabrin, and Alberto Policriti. GAM: Genomic Assemblies Merger: A Graph Based Method to Integrate Different Assemblies. *2009 IEEE International Conference on Bioinformatics and Biomedicine*, pages 321–326, November 2009.

[33] Federica Cattonaro, Alberto Policriti, and Francesco Vezzi. *Enhanced reference guided assembly.* IEEE, December 2010.

[34] Mark Chaisson and Glenn Tesler. Mapping single molecule sequencing reads using basic local alignment with successive refinement (BLASR): application and theory. *BMC Bioinformatics*, 13(1):238, 2012.

[35] Mark J. Chaisson, Dumitru Brinza, and Pavel A. Pevzner. De novo fragment assembly with short mate-paired reads: Does the read length matter? *Genome Research*, 19(2):336–346, 2009.

[36] Mark J. Chaisson and Pavel A. Pevzner. Short read fragment assembly of bacterial genomes. *Genome Research*, 18(2):000, 2007.

[37] Yangho Chen, Tate Souaiaia, and Ting Chen. PerM: efficient mapping of short sequencing reads with periodic full sensitive spaced seeds. *Bioinformatics*, 25(19):2514–2521, October 2009.

[38] Rayan Chikhi and Guillaume Rizk. Space-Efficient and Exact de Bruijn Graph Representation Based on a Bloom Filter. In *WABI*, volume 7534 of *Lecture Notes in Computer Science*, pages 236–248. Springer, 2012.

[39] Chen-Shan Chin, David H Alexander, Patrick Marks, Aaron A Klammer, James Drake, Cheryl Heiner, Alicia Clum, Alex Copeland, John Huddleston, Evan E Eichler, Stephen W Turner, and Jonas Korlach. Nonhybrid, finished microbial genome assemblies from long-read SMRT sequencing data. *Nat Meth*, 10(6):563–569, June 2013.

[40] Deanna M. Church, Leo Goodstadt, LaDeana W. Hillier, Michael C. Zody, Steve Goldstein, Xinwe She, Carol J. Bult, Richa Agarwala, Joshua L. Cherry, Michael DiCuccio, Wratko Hlavina, Yuri Kapustin, Peter Meric, Donna Maglott, ZoÃń Birtle, Ana C. Marques, Tina Graves, Shiguo Zhou, Brian Teague, Konstantinos Potamousis, Christopher Churas, Michael Place, Jill Herschleb, Ron Runnheim, Daniel Forrest, James Amos-Landgraf, David C. Schwartz, Ze Cheng, Kerstin Lindblad-Toh, Evan E. Eichler, Chris P. Ponting, and The Mouse Genome Sequencing Consortium. Lineage-Specific Biology Revealed by a Finished Genome Assembly of the Mouse. *PLoS Biol*, 7(5):e1000112, 05 2009.

[41] Gary A. Churchill and Michael S. Waterman. The accuracy of DNA sequences: Estimating sequence quality. *Genomics*, 14(1):89 – 98, 1992.

[42] James Clarke, Hai-Chen Wu, Lakmal Jayasinghe, Alpesh Patel, Stuart Reid, and Hagan Bayley. Continuous base identification for single-molecule nanopore DNA sequencing. *Nat Nano*, 4(4):265–270, April 2009.

[43] Francis S. Collins, Michael Morgan, and Aristides Patrinos. The Human Genome Project: Lessons from Large-Scale Biology. *Science*, 300(5617):286–290, 2003.

[44] Human Genome Sequencing ConsortiumInternational. Finishing the euchromatic sequence of the human genome. *Nature*, 431(7011):931–945, October 2004.

[45] Rami a. Dalloul, Julie a. Long, Aleksey V. Zimin, Luqman Aslam, Kathryn Beal, Le Ann Blomberg, Pascal Bouffard, David W. Burt, Oswald Crasta, Richard P. M. a. Crooijmans, Kristal Cooper, Roger a. Coulombe, Supriyo De, Mary E. Delany, Jerry B. Dodgson, Jennifer J. Dong, Clive Evans, Karin M. Frederickson, Paul Flicek, Liliana Florea, Otto Folkerts, Martien a. M. Groenen, Tim T. Harkins, Javier Herrero, Steve Hoffmann, Hendrik-Jan Megens, Andrew Jiang, Pieter de Jong, Pete Kaiser, Heebal Kim, Kyu-Won Kim, Sungwon Kim, David Langenberger, Mi-Kyung Lee, Taeheon Lee, Shrinivasrao Mane, Guillaume Marcais, Manja Marz, Audrey P. McElroy, Thero Modise, Mikhail Nefedov, Cédric Notredame, Ian R. Paton, William S. Payne, Geo Pertea, Dennis Prickett, Daniela Puiu, Dan Qioa, Emanuele Raineri, Magali Ruffier, Steven L. Salzberg, Michael C. Schatz, Chantel Scheuring, Carl J. Schmidt, Steven Schroeder, Stephen M. J. Searle, Edward J. Smith, Jacqueline Smith, Tad S. Sonstegard, Peter F. Stadler, Hakim Tafer, Zhijian (Jake) Tu, Curtis P. Van Tassell, Albert J. Vilella, Kelly P. Williams, James a. Yorke, Liqing Zhang, Hong-Bin Zhang, Xiaojun Zhang, Yang Zhang, and Kent M. Reed. Multi-Platform Next-Generation Sequencing of the Domestic Turkey (Meleagris gallopavo): Genome Assembly and Analysis. *PLoS Biology*, 8(9):e1000475, September 2010.

[46] Adel Dayarian, Todd Michael, and Anirvan Sengupta. SOPRA: Scaffolding algorithm for paired reads via statistical optimization. *BMC Bioinformatics*, 11(1):345, 2010.

[47] N. G. de Bruijn. A Combinatorial Problem. *Koninklijke Nederlandsche Akademie Van Wetenschappen*, 49(6):758–764, June 1946.

[48] C. Del Fabbro, F. Tardivo, and A. Policriti. A Parallel Algorithm for the Best k-Mismatches Alignment Problem. In *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*, pages 586–589, Feb 2014.

[49] V. Deshpande, E. D. Fung, S. Pham, and V. Bafna. Cerulean: A hybrid assembly using high throughput short and long reads. *ArXiv e-prints*, July 2013.

[50] Juliane C. Dohm, Claudio Lottaz, Tatiana Borodina, and Heinz Himmelbauer. SHARCGS, a fast and highly accurate short-read assembly algorithm for de novo genomic sequencing. *Genome Research*, 17(11):1697–1706, 2007.

[51] Nilgun Donmez and Michael Brudno. SCARPA: scaffolding reads with practical algorithms. *Bioinformatics*, 29(4):428–434, 2013.

[52] D. a. Earl, K. Bradnam, J. St. John, a. Darling, D. Lin, J. Faas, H. O. K. Yu, B. Vince, D. R. Zerbino, M. Diekhans, N. Nguyen, P. Nuwantha, a. W.-K. Sung, Z. Ning, M. Haimel, J. T. Simpson, N. a. Fronseca, I. Birol, T. R. Docking, I. Y. Ho, D. S. Rokhsar, R. Chikhi, D. Lavenier, G. Chapuis, D. Naquin, N. Maillet, M. C. Schatz, D. R. Kelly, a. M. Phillippy, S. Koren, S.-P. Yang, W. Wu, W.-C. Chou, a. Srivastava, T. I. Shaw, J. G. Ruby, P. Skewes-Cox, M. Betegon, M. T. Dimon, V. Solovyev, P. Kosarev, D. Vorobyev, R. Ramirez-Gonzalez, R. Leggett, D. MacLean, F. Xia, R. Luo, Z. L, Y. Xie, B. Liu, S. Gnerre, I. MacCallum, D. Przybylski, F. J. Ribeiro, S. Yin, T. Sharpe, G. Hall, P. J. Kersey, R. Durbin, S. D. Jackman, J. a. Chapman, X. Huang, J. L. DeRisi, M. Caccamo, Y. Li, D. B. Jaffe, R. Green, D. Haussler, I. Korf, and B. Paten. Assemblathon 1: A competitive assessment of de novo short read assembly methods. *Genome Research*, September 2011.

[53] John Eid, Adrian Fehr, Jeremy Gray, Khai Luong, John Lyle, Geoff Otto, Paul Peluso, David Rank, Primo Baybayan, Brad Bettman, Arkadiusz Bibillo, Keith Bjornson, Bidhan Chaudhuri, Frederick Christians, Ronald Cicero, Sonya Clark, Ravindra Dalal, Alex deWinter, John Dixon, Mathieu Foquet, Alfred Gaertner, Paul Hardenbol, Cheryl Heiner, Kevin Hester, David Holden, Gregory Kearns, Xiangxu Kong, Ronald Kuse, Yves Lacroix, Steven Lin, Paul Lundquist, Congcong Ma, Patrick Marks, Mark Maxham, Devon Murphy, Insil Park, Thang Pham, Michael Phillips, Joy Roy, Robert Sebra, Gene Shen, Jon Sorenson, Austin Tomaney, Kevin Travers, Mark Trulson, John Vieceli, Jeffrey Wegener, Dawn Wu, Alicia Yang, Denis Zaccarin, Peter Zhao, Frank Zhong, Jonas Korlach, and Stephen Turner. Real-Time DNA Sequencing from Single Polymerase Molecules. *Science*, 323(5910):133–138, 2009.

[54] Isaac Elias. Settling the Intractability of Multiple Alignment. *Journal of Computational Biology*, 13(7):1323–1339, September 2006.

[55] Adam C. English, Stephen Richards, Yi Han, Min Wang, Vanesa Vee, Jiaxin Qu, Xiang Qin, Donna M. Muzny, Jeffrey G. Reid, Kim C. Worley, and Richard A. Gibbs. Mind the Gap: Upgrading Genomes with Pacific Biosciences RS Long-Read Sequencing Technology. *PLoS ONE*, 7(11):e47768, 11 2012.

[56] Brent Ewing and Phil Green. Base-Calling of Automated Sequencer Traces Using Phred. II. Error Probabilities. *Genome Research*, 8(3):186–194, 1998.

[57] Michael Farrar. Striped Smith-Waterman speeds database searches six times over other SIMD implementations. *Bioinformatics*, 23(2):156–161, 2007.

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

[59] Paolo Ferragina, Travis Gagie, and Giovanni Manzini. Lightweight Data Indexing and Compression in External Memory. *Algorithmica*, 63(3):707–730, 2012.

[60] Paul Flicek and Ewan Birney. Sense from sequence reads: methods for alignment and assembly. *Nature methods*, 6:S6–S12, 2009.

[61] John Gallant, David Maier, and James Astorer. On finding minimal length superstrings. *Journal of Computer and System Sciences*, 20(1):50 – 58, 1980.

[62] Song Gao, Niranjan Nagarajan, and Wing-Kin Sung. Opera: Reconstructing Optimal Genomic Scaffolds with High-Throughput Paired-End Sequences. In Vineet Bafna and S.Cenk Sahinalp, editors, *Research in Computational Molecular Biology*, volume 6577 of *Lecture Notes in Computer Science*, pages 437–451. Springer Berlin Heidelberg, 2011.

[63] Walter Gilbert and Allan Maxam. The Nucleotide Sequence of the lac Operator. *Proceedings of the National Academy of Sciences of the United States of America*, 70(12 Pt 1-2):3581–3584, December 1973.

[64] T R Gingeras, J P Milazzo, D Sciaky, and R J Roberts. Computer programs for the assembly of DNA sequences. *Nucleic Acids Research*, 7(2):529–545, September 1979.

[65] Sante Gnerre, Iain MacCallum, Dariusz Przybylski, Filipe J. Ribeiro, Joshua N. Burton, Bruce J. Walker, Ted Sharpe, Giles Hall, Terrance P. Shea, Sean Sykes, Aaron M. Berlin, Daniel Aird, Maura Costello, Riza Daza, Louise Williams, Robert Nicol, Andreas Gnirke, Chad Nusbaum, Eric S. Lander, and David B. Jaffe. High-quality draft assemblies of mammalian genomes from massively parallel sequence data. *Proceedings of the National Academy of Sciences*, 108(4):1513–1518, 2011.

[66] Sara Goodwin, James Gurtowski, Scott Ethe-Sayers, Panchajanya Deshpande, Michael C. Schatz, and W. Richard McCombie. Oxford Nanopore sequencing, hybrid error correction, and de novo assembly of a eukaryotic genome. *Genome Research*, 2015.

[67] Phil Green. PHRAP. [http://www.phrap.org], 2002.

[68] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI (2Nd Ed.): Portable Parallel Programming with the Message-passing Interface*. MIT Press, Cambridge, MA, USA, 1999.

[69] R.S. Harris. *Improved pairwise alignment of genomic DNA*. PhD thesis, The Pennsylvania State University, 2007.

[70] Timothy D. Harris, Phillip R. Buzby, Hazen Babcock, Eric Beer, Jayson Bowers, Ido Braslavsky, Marie Causey, Jennifer Colonell, James DiMeo, J. William Efcavitch, Eldar Giladi, Jaime Gill, John Healy, Mirna Jarosz, Dan Lapen, Keith Moulton, Stephen R. Quake, Kathleen Steinmann, Edward Thayer, Anastasia Tyurina, Rebecca Ward, Howard Weiss, and Zheng Xie. Single-Molecule DNA Sequencing of a Viral Genome. *Science*, 320(5872):106–109, 2008.

[71] Yun Heo, Xiao-Long Wu, Deming Chen, Jian Ma, and Wen-Mei Hwu. BLESS: Bloom filter-based error correction solution for high-throughput sequencing reads. *Bioinformatics*, 30(10):1354–1362, 2014.

[72] C. Hercus. Novocraft short read alignment package. `http://www.novocraft.com`, 2009.

[73] David Hernandez, Patrice François, Laurent Farinelli, Magne Østerås, and Jacques Schrenzel. *De novo* bacterial genome sequencing: Millions of very short reads assembled on a desktop computer. *Genome Research*, 2008.

[74] Wing-Kai Hon, Tak-Wah Lam, Kunihiko Sadakane, Wing-Kin Sung, and Siu-Ming Yiu. A Space and Time Efficient Algorithm for Constructing Compressed Suffix Arrays. *Algorithmica*, 48(1):23–36, 2007.

[75] Mohammad Hossain, Navid Azimi, and Steven Skiena. Crystallizing short-read assemblies around seeds. *BMC Bioinformatics*, 10(Suppl 1):S16, 2009.

[76] Stefan Howorka, Stephen Cheley, and Hagan Bayley. Sequence-specific detection of individual DNA strands using engineered nanopores. *Nat Biotech*, 19(7):636–639, July 2001.

[77] Xuesong Hu, Jianying Yuan, Yujian Shi, Jianliang Lu, Binghang Liu, Zhenyu Li, Yanxiang Chen, Desheng Mu, Hao Zhang, Nan Li, Zhen Yue, Fan Bai, Heng Li, and Wei Fan. pIRS: Profile-based Illumina pair-end reads simulator. *Bioinformatics*, 28(11):1533–1535, 2012.

[78] Xiaoqiu Huang and Anup Madan. CAP3: A DNA Sequence Assembly Program. *Genome Research*, 9(9):868–877, 1999.

[79] Xiaoqiu Huang, Jianmin Wang, Srinivas Aluru, Shiaw-Pyng Yang, and LaDeana Hillier. PCAP: A Whole-Genome Assembly Program. *Genome Research*, 13(9):2164–2170, 2003.

[80] T Hunkapiller, RJ Kaiser, BF Koop, and L Hood. Large-scale and automated DNA sequence determination. *Science*, 254(5028):59–67, 1991.

[81] Martin Hunt, Taisei Kikuchi, Mandy Sanders, Chris Newbold, Matthew Berriman, and Thomas Otto. REAPR: a universal tool for genome assembly evaluation. *Genome Biology*, 14(5):R47, 2013.

[82] Daniel H. Huson, Aaron L. Halpern, Zhongwu Lai, Eugene W. Myers, Knut Reinert, and Granger G. Sutton. Comparing Assemblies Using Fragments and Mate-Pairs. In Olivier Gascuel and BernardM.E. Moret, editors, *Algorithms in Bioinformatics*, volume 2149 of *Lecture Notes in Computer Science*, pages 294–306. Springer Berlin Heidelberg, 2001.

[83] Ramana M. Idury and Michael S. Waterman. A New Algorithm for DNA Sequence Assembly. *Journal of Computational Biology*, 2(2):291–306, January 1995.

[84] Lucian Ilie, Silvana Ilie, and Anahita Mansouri Bigvand. SpEED: fast computation of sensitive spaced seeds. *Bioinformatics*, 27(17):2433–2434, 2011.

[85] William R. Jeck, Josephine A. Reinhardt, David A. Baltrus, Matthew T. Hickenbotham, Vincent Magrini, Elaine R. Mardis, Jeffery L. Dangl, and Corbin D. Jones. Extending assembly of short DNA sequences to handle error. *Bioinformatics*, 23(21):2942–2944, 2007.

[86] Richard M. Karp and M.O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, March 1987.

[87] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. In *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching*, CPM '01, pages 181–192, London, UK, UK, 2001. Springer-Verlag.

[88] J. D. Kececioglu and E. W. Myers. Combinatorial algorithms for DNA sequence assembly. *Algorithmica*, 13(1-2):7–51, 1995.

[89] W. James Kent. BLAT – The BLAST-Like Alignment Tool. *Genome Research*, 12(4):656–664, 2002.

[90] Sergey Koren, Michael C Schatz, Brian P Walenz, Jeffrey Martin, Jason T Howard, Ganeshkumar Ganapathy, Zhong Wang, David A Rasko, W Richard McCombie, Erich D Jarvis, and Adam M Phillippy. Hybrid error correction and de novo assembly of single-molecule sequencing reads. *Nat Biotech*, 30(7):693–700, July 2012.

[91] Eric S. Lander and Michael S. Waterman. Genomic mapping by fingerprinting random clones: A mathematical analysis. *Genomics*, 2(3):231 – 239, 1988.

[92] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(3):R25, 2009.

[93] M. J. Levene, J. Korlach, S. W. Turner, M. Foquet, H. G. Craighead, and W. W. Webb. Zero-Mode Waveguides for Single-Molecule Analysis at High Concentrations. *Science*, 299(5607):682–686, 2003.

[94] H. Li. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. *ArXiv e-prints*, March 2013.

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

[96] Heng Li and Richard Durbin. Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinformatics*, 26(5):589–595, 2010.

[97] R Li, Wei Fan, G Tian, Hongmei Zhu, Lin He, J Cai, Q Huang, and Q. The sequence and de novo assembly of the giant panda genome. *Nature*, 463(January):311–317, 2009.

[98] R. Li, H. Zhu, J. Ruan, W. Qian, X. Fang, Z. Shi, Y. Li, S. Li, G. Shan, K. Kristiansen, S. Li, H. Yang, J. Wang, and J. Wang. De novo assembly of human genomes with massively parallel short read sequencing. *Genome*, doi:101101/gr097261109, 2010.

[99] Ruiqiang Li, Jia Ye, Songgang Li, Jing Wang, Yujun Han, Chen Ye, Jian Wang, Huanming Yang, Jun Yu, Gane Ka-Shu Wong, and Jun Wang. ReAS: Recovery of Ancestral Sequences for Transposable Elements from the Unassembled Reads of a Whole Genome Shotgun. *PLoS Comput Biol*, 1(4):e43, 09 2005.

[100] Ruiqiang Li, Chang Yu, Yingrui Li, Tak-Wah Lam, Siu-Ming Yiu, Karsten Kristiansen, and Jun Wang. SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009.

[101] Hao Lin, Zefeng Zhang, Michael Q. Zhang, Bin Ma, and Ming Li. ZOOM! Zillions Of Oligos Mapped. *Bioinformatics*, 24(21):2431–2437, November 2008.

[102] Kerstin Lindblad-Toh, Claire M Wade, Tarjei S. Mikkelsen, Elinor K. Karlsson, David B. Jaffe, Michael Kamal, Michele Clamp, Jean L. Chang, Edward J. Kulbokas, Michael C. Zody, Evan Mauceli, Xiaohui Xie, Matthew Breen, Robert K. Wayne, Elaine A. Ostrander, Chris P. Ponting, Francis Galibert, Douglas R. Smith, Pieter J. deJong, Ewen Kirkness, Pablo Alvarez, Tara Biagi, William Brockman, Jonathan Butler, Chee-Wye Chin, April Cook, James Cuff, Mark J. Daly, David DeCaprio, Sante Gnerre, Manfred Grabherr, Manolis Kellis, Michael Kleber, Carolyne Bardeleben, Leo Goodstadt, Andreas Heger, Christophe Hitte, Lisa Kim, Klaus-Peter Koepfli, Heidi G. Parker, John P. Pollinger, Stephen M. J. Searle, Nathan B. Sutter, Rachael Thomas, Caleb Webber, and Eric S. Lander. Genome sequence, comparative analysis and haplotype structure of the domestic dog. *Nature*, 438(7069):803–819, December 2005.

[103] Lin Liu, Yinhu Li, Siliang Li, Ni Hu, Yimin He, Ray Pong, Danni Lin, Lihua Lu, and Maggie Law. Comparison of Next-Generation Sequencing Systems, 2012.

[104] Ruibang Luo, Binghang Liu, Yinlong Xie, Zhenyu Li, Weihua Huang, Jianying Yuan, Guangzhu He, Yanxiang Chen, Qi Pan, Yunjie Liu, Jingbo Tang, Gengxiong Wu, Hao Zhang, Yujian Shi, Yong Liu, Chang Yu, Bo Wang, Yao Lu, Changlei Han, David Cheung, Siu-Ming Yiu, Shaoliang Peng, Zhu Xiaoqian, Guangming Liu, Xiangke Liao, Yingrui Li, Huanming Yang, Jian Wang, Tak-Wah Lam, and Jun Wang. SOAPdenovo2: an empirically improved memory-efficient short-read de novo assembler. *GigaScience*, 1(1):18, 2012.

[105] Bin Ma, John Tromp, and Ming Li. PatternHunter: faster and more sensitive homology search. *Bioinformatics*, 18(3):440–445, 2002.

[106] Tanja Magoc, Stephan Pabinger, Stefan Canzar, Xinyue Liu, Qi Su, Daniela Puiu, Luke J. Tallon, and Steven L. Salzberg. GAGE-B: an evaluation of genome assemblers for bacterial organisms. *Bioinformatics*, 29(14):1718–1725, 2013.

[107] Udi Manber and Gene Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[108] Guillaume Marçais and Carl Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, 2011.

[109] Elaine R Mardis. The impact of next-generation sequencing technology on genetics. *Trends in genetics : TIG*, 24(3):133–41, March 2008.

[110] Marcel Margulies, Michael Egholm, William E. Altman, Said Attiya, Joel S. Bader, Lisa A. Bemben, Jan Berka, Michael S. Braverman, Yi-Ju Chen, Zhoutao Chen, Scott B. Dewell, Lei Du, Joseph M. Fierro, Xavier V. Gomes, Brian C. Godwin, Wen He, Scott Helgesen, Chun He Ho, Gerard P. Irzyk, Szilveszter C. Jando, Maria L. I. Alenquer, Thomas P. Jarvie, Kshama B. Jirage, Jong-Bum Kim, James R. Knight, Janna R. Lanza, John H. Leamon, Steven M. Lefkowitz, Ming Lei, Jing Li, Kenton L. Lohman, Hong Lu, Vinod B. Makhijani, Keith E. McDade, Michael P. McKenna, Eugene W. Myers, Elizabeth Nickerson, John R. Nobile, Ramona Plant, Bernard P. Puc, Michael T. Ronan, George T. Roth, Gary J. Sarkis, Jan Fredrik Simons, John W. Simpson, Maithreyan Srinivasan, Karrie R. Tartaro, Alexander Tomasz, Kari A. Vogt, Greg A. Volkmer, Shally H. Wang, Yong Wang, Michael P. Weiner, Pengguang Yu, Richard F. Begley, and Jonathan M. Rothberg. Genome sequencing in microfabricated high-density picolitre reactors. *Nature*, 437(7057):376–380, September 2005.

[111] Kevin Judd McKernan, Heather E. Peckham, Gina L. Costa, Stephen F. McLaughlin, Yutao Fu, Eric F. Tsung, Christopher R. Clouser, Cisyla Duncan, Jeffrey K. Ichikawa, Clarence C. Lee, Zheng Zhang, Swati S. Ranade, Eileen T. Dimalanta, Fiona C. Hyland, Tanya D. Sokolsky, Lei Zhang, Andrew Sheridan, Haoning Fu, Cynthia L. Hendrickson, Bin Li, Lev Kotler, Jeremy R. Stuart, Joel A. Malek, Jonathan M. Manning, Alena A. Antipova, Damon S. Perez, Michael P. Moore, Kathleen C. Hayashibara, Michael R. Lyons, Robert E. Beaudoin, Brittany E.

Coleman, Michael W. Laptewicz, Adam E. Sannicandro, Michael D. Rhodes, Rajesh K. Gottimukkala, Shan Yang, Vineet Bafna, Ali Bashir, Andrew MacBride, Can Alkan, Jeffrey M. Kidd, Evan E. Eichler, Martin G. Reese, Francisco M. De La Vega, and Alan P. Blanchard. Sequence and structural variation in a human genome uncovered by short-read, massively parallel ligation sequencing using two-base encoding. *Genome Research*, 19(9):1527–1541, 2009.

[112] Paul Medvedev, Konstantinos Georgiou, Gene Myers, and Michael Brudno. Computability of Models for Sequence Assembly. In Raffaele Giancarlo and Sridhar Hannenhalli, editors, *Algorithms in Bioinformatics*, volume 4645 of *Lecture Notes in Computer Science*, pages 289–301. Springer Berlin Heidelberg, 2007.

[113] Paul Medvedev, Son Pham, Mark Chaisson, Glenn Tesler, and Pavel Pevzner. Paired de Bruijn Graphs: A Novel Approach for Incorporating Mate Pair Information into Genome Assemblers. *Journal of Computational Biology*, 18(11):1625–1634, October 2011.

[114] Pall Melsted and Jonathan Pritchard. Efficient counting of k-mers in DNA sequences using a bloom filter. *BMC Bioinformatics*, 12(1):333, 2011.

[115] Jason R. Miller, Arthur L. Delcher, Sergey Koren, Eli Venter, Brian P. Walenz, Anushka Brownley, Justin Johnson, Kelvin Li, Clark Mobarry, and Granger Sutton. Aggressive assembly of pyrosequencing reads with mates. *Bioinformatics*, 24(24):2818–2824, 2008.

[116] Jason R Miller, Sergey Koren, and Granger Sutton. Assembly Algorithms for Next-Generation Sequencing Data. *Genomics*, 95(6):315–327, March 2010.

[117] Hamid Mohamadi, Benjamin P Vandervalk, Anthony Raymond, Shaun D Jackman, Justin Chu, Clay P Breshears, and Inanc Birol. DIDA: Distributed Indexing Dispatched Alignment. *PLoS ONE*, 10(4):e0126409, 04 2015.

[118] James C. Mullikin and Zemin Ning. The Phusion Assembler. *Genome Research*, 13(1):81–90, 2003.

[119] David J Munroe and Timothy J R Harris. Third-generation sequencing fireworks at Marco Island. *Nat Biotech*, 28(5):426–428, May 2010.

[120] Alysson R. Muotri, Maria C.N. Marchetto, Nicole G. Coufal, and Fred H. Gage. The necessary junk: new functions for transposable elements. *Human Molecular Genetics*, 16(R2):R159–R167, 2007.

[121] Eugene W. Myers. An $O(ND)$ difference algorithm and its variations. *Algorithmica*, 1(1-4):251–266, 1986.

[122] Eugene W. Myers. The fragment assembly string graph. *Bioinformatics*, 21(suppl 2):ii79–ii85, 2005.

[123] Eugene W. Myers, Granger G. Sutton, Art L. Delcher, Ian M. Dew, Dan P. Fasulo, Michael J. Flanigan, Saul A. Kravitz, Clark M. Mobarry, Knut H. J. Reinert,

Karin A. Remington, Eric L. Anson, Randall A. Bolanos, Hui-Hsien Chou, Catherine M. Jordan, Aaron L. Halpern, Stefano Lonardi, Ellen M. Beasley, Rhonda C. Brandon, Lin Chen, Patrick J. Dunn, Zhongwu Lai, Yong Liang, Deborah R. Nusskern, Ming Zhan, Qing Zhang, Xiangqun Zheng, Gerald M. Rubin, Mark D. Adams, and J. Craig Venter. A Whole-Genome Assembly of Drosophila. *Science*, 287(5461):2196–2204, 2000.

[124] Gene Myers. Efficient Local Alignment Discovery amongst Noisy Long Reads. In Dan Brown and Burkhard Morgenstern, editors, *Algorithms in Bioinformatics*, volume 8701 of *Lecture Notes in Computer Science*, pages 52–67. Springer Berlin Heidelberg, 2014.

[125] Francesca Nadalin, Francesco Vezzi, and Alberto Policriti. GapFiller: a *de novo* assembly approach to fill the gap within paired reads. *BMC Bioinformatics*, 13(Suppl 14):S8, 2012.

[126] N Nagarajan and M Pop. Parametric complexity of sequence assembly: Theory and applications to next generation sequencing. *Journal of Computational Biology*, 16:897–908, 2009.

[127] Giuseppe Narzisi and Bud Mishra. Scoring-and-Unfolding Trimmed Tree Assembler: Concepts, Constructs and Comparisons. *Bioinformatics (Oxford, England)*, 27(2):153–160, November 2010.

[128] Giuseppe Narzisi and Bud Mishra. Comparing De Novo Genome Assembly: The Long and Short of It. *PLoS ONE*, 6(4):e19175–, March 2011.

[129] Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443 – 453, 1970.

[130] Jurgen Nijkamp, Wynand Winterbach, Marcel van den Broek, Jean-Marc Daran, Marcel Reinders, and Dick de Ridder. Integrating genome assemblies with MAIA. *Bioinformatics (Oxford, England)*, 26(18):i433–i439, September 2010.

[131] Zemin Ning, Anthony J. Cox, and James C. Mullikin. SSAHA: A Fast Search Method for Large DNA Databases. *Genome Research*, 11(10):1725–1729, 2001.

[132] Laurent Noé, Marta Gîrdea, and Gregory Kucherov. Designing efficient spaced seeds for SOLiD read mapping. *Advances in Bioinformatics*, 2010:ID 708501, July 2010.

[133] M Nowrousian, JE Stajich, M Chu, I Engh, and E Espagne. De novo Assembly of a 40 Mb eukaryotic genome from short sequence reads: Sordaria macrospora, a model organism for fungal morphogenesis. *PLoS Genet*, 2010.

[134] Björn Nystedt, Nathaniel R. Street, Anna Wetterbom, Andrea Zuccolo, Yao-Cheng Lin, Douglas G. Scofield, Francesco Vezzi, Nicolas Delhomme, Stefania Giacomello, Andrey Alexeyenko, Riccardo Vicedomini, Kristoffer Sahlin, Ellen Sherwood, Malin Elfstrand, Lydia Gramzow, Kristina Holmberg, Jimmie Hällman,

Olivier Keech, Lisa Klasson, Maxim Koriabine, Melis Kucukoglu, Max Käller, Johannes Luthman, Fredrik Lysholm, Totte Niittylä, Åke Olson, Nemanja Rilakovic, Carol Ritland, Josep A. Rossello, Juliana Sena, Thomas Svensson, Carlos Talavera-López, Gunter Theiszen, Hannele Tuominen, Kevin Vanneste, Zhi-Qiang Wu, Bo Zhang, Philipp Zerbe, Lars Arvestad, Rishikesh Bhalerao, Joerg Bohlmann, Jean Bousquet, Rosario Garcia Gil, Torgeir R. Hvidsten, Pieter de Jong, John MacKay, Michele Morgante, Kermit Ritland, Björn Sundberg, Stacey Lee Thompson, Yves Van de Peer, Björn Andersson, Ove Nilsson, Pär K. Ingvarsson, Joakim Lundeberg, and Stefan Jansson. The Norway spruce genome sequence and conifer genome evolution. *Nature*, 497(7451):579–584, May 2013.

[135] Anna Pagh, Rasmus Pagh, and S. Srinivasa Rao. An Optimal Bloom Filter Replacement. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '05, pages 823–829, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics.

[136] Jason Pell, Arend Hintze, Rosangela Canino-Koning, Adina Howe, James M. Tiedje, and C. Titus Brown. Scaling metagenome sequence assembly with probabilistic de Bruijn graphs. *Proceedings of the National Academy of Sciences*, 109(33):13272–13277, 2012.

[137] H Peltola, H Söderlund, and E Ukkonen. SEQAID: a DNA sequence assembling program based on a mathematical model. *Nucleic Acids Research*, 12(1 Pt 1):307–321, January 1984.

[138] Yu Peng, Henry C. M. Leung, S. M. Yiu, and Francis Y. L. Chin. IDBA-UD: a de novo assembler for single-cell and metagenomic sequencing data with highly uneven depth. *Bioinformatics*, 28(11):1420–1428, 2012.

[139] Pavel A. Pevzner and Haixu Tang. Fragment assembly with double-barreled data. *Bioinformatics*, 17(suppl 1):S225–S233, 2001.

[140] Pavel A Pevzner, Haixu Tang, and Michael S Waterman. An Eulerian path approach to DNA fragment assembly. *Proceedings of the National Academy of Sciences of the United States of America*, 98(17):9748–9753, June 2001.

[141] Adam M Phillippy, Michael C Schatz, and Mihai Pop. Genome assembly forensics: finding the elusive mis-assembly. *Genome biology*, 9(3):R55, January 2008.

[142] A. Polanski and M. Kimmel. Genomics. In *Bioinformatics*, pages 213–260. Springer Berlin Heidelberg, 2007.

[143] Alberto Policriti, Nicola Gigante, and Nicola Prezza. Average Linear Time and Compressed Space Construction of the Burrows-Wheeler Transform. In Adrian-Horia Dediu, Enrico Formenti, Carlos Martín-Vide, and Bianca Truthe, editors, *Language and Automata Theory and Applications*, volume 8977 of *Lecture Notes in Computer Science*, pages 587–598. Springer International Publishing, 2015.

[144] Alberto Policriti and Nicola Prezza. *Algorithms and Computation: 25th International Symposium, ISAAC 2014, Jeonju, Korea, December 15-17, 2014, Proceedings*, chapter Hashing and Indexing: Succinct Data Structures and Smoothed Analysis, pages 157–168. Springer International Publishing, Cham, 2014.

[145] Alberto Policriti and Nicola Prezza. Fast randomized approximate string matching with succinct hash data structures. *BMC Bioinformatics*, 16(Suppl 9):S4, 2015.

[146] Michael A Quail, Miriam Smith, Paul Coupland, Thomas D Otto, Simon R Harris, Thomas R Connor, Anna Bertoni, Harold P Swerdlow, and Yong Gu. A tale of three next generation sequencing platforms: comparison of Ion Torrent, Pacific Biosciences and Illumina MiSeq sequencers. *BMC Genomics*, 13:341–341, July 2012.

[147] Aaron R Quinlan, Donald A Stewart, Michael P Strömberg, and Gábor T Marth. Pyrobayes: an improved base caller for SNP discovery in pyrosequences. *Nat Meth*, 5(2):179–181, February 2008.

[148] Steven Ralph, Hye Chun, Natalia Kolosova, Dawn Cooper, Claire Oddy, Carol Ritland, Robert Kirkpatrick, Richard Moore, Sarah Barber, Robert Holt, Steven Jones, Marco Marra, Carl Douglas, Kermit Ritland, and Jorg Bohlmann. A conifer genomics resource of 200,000 spruce (*Picea* spp.) ESTs and 6,464 high-quality, sequence-finished full-length cDNAs for Sitka spruce (*Picea sitchensis*). *BMC Genomics*, 9(1):484, 2008.

[149] Josephine A. Reinhardt, David A. Baltrus, Marc T. Nishimura, William R. Jeck, Corbin D. Jones, and Jeffery L. Dangl. De novo assembly using low-coverage short read sequence data from the rice pathogen *Pseudomonas syringae* pv. *oryzae*. *Genome Research*, 19(2):294–305, 2009.

[150] Michael Roberts, Wayne Hayes, Brian R. Hunt, Stephen M. Mount, and James A. Yorke. Reducing storage requirements for biological sequence comparison. *Bioinformatics*, 20(18):3363–3369, 2004.

[151] Mostafa Ronaghi, Samer Karamohamed, Bertil Pettersson, Mathias Uhlén, and Pål Nyrén. Real-Time DNA Sequencing Using Detection of Pyrophosphate Release. *Analytical Biochemistry*, 242(1):84 – 89, 1996.

[152] Jonathan M. Rothberg, Wolfgang Hinz, Todd M. Rearick, Jonathan Schultz, William Mileski, Mel Davey, John H. Leamon, Kim Johnson, Mark J. Milgrew, Matthew Edwards, Jeremy Hoon, Jan F. Simons, David Marran, Jason W. Myers, John F. Davidson, Annika Branting, John R. Nobile, Bernard P. Puc, David Light, Travis A. Clark, Martin Huber, Jeffrey T. Branciforte, Isaac B. Stoner, Simon E. Cawley, Michael Lyons, Yutao Fu, Nils Homer, Marina Sedova, Xin Miao, Brian Reed, Jeffrey Sabina, Erika Feierstein, Michelle Schorn, Mohammad Alanjary, Eileen Dimalanta, Devin Dressman, Rachel Kasinskas, Tanya Sokolsky, Jacqueline A. Fidanza, Eugeni Namsaraev, Kevin J. McKernan, Alan Williams, G. Thomas Roth, and James Bustillo. An integrated semiconductor device enabling non-optical genome sequencing. *Nature*, 475(7356):348–352, July 2011.

[153] Kristoffer Sahlin, Francesco Vezzi, Björn Nystedt, Joakim Lundeberg, and Lars Arvestad. BESST - Efficient scaffolding of large fragmented assemblies. *BMC Bioinformatics*, 15(1):281, 2014.

[154] S. L. Salzberg, a. M. Phillippy, a. V. Zimin, D. Puiu, T. Magoc, S. Koren, T. Treangen, M. C. Schatz, a. L. Delcher, M. Roberts, G. Marcais, M. Pop, and J. a. Yorke. GAGE: A critical evaluation of genome assemblies and assembly algorithms. *Genome Research*, December 2011.

[155] F. Sanger, G. M. Air, B. G. Barrell, N. L. Brown, A. R. Coulson, J. C. Fiddes, C. A. Hutchison, P. M. Slocombe, and M. Smith. Nucleotide sequence of bacteriophage φX174 DNA. *Nature*, 265(5596):687–695, February 1977.

[156] F. Sanger and A. R. Coulson. A rapid method for determining sequences in DNA by primed synthesis with DNA polymerase. *J Mol Biol*, 94(3):441–448, May 1975.

[157] Eric E. Schadt, Steve Turner, and Andrew Kasarskis. A window into third-generation sequencing. *Human Molecular Genetics*, 19(R2):R227–R240, 2010.

[158] Michael C Schatz, Arthur L Delcher, and Steven L Salzberg. Assembly of large genomes using second-generation sequencing. *Genome Research*, 20(9):1165–1173, September 2010.

[159] Jay Shendure and Hanlee Ji. Next-generation DNA sequencing. *Nat Biotech*, 26(10):1135–1145, October 2008.

[160] Jay Shendure, Robi D. Mitra, Chris Varma, and George M. Church. Advanced sequencing technologies: methods and goals. *Nat Rev Genet*, 5(5):335–344, May 2004.

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

[162] JT Simpson, K Wong, SD Jackman, and JE Schein. ABySS: A parallel assembler for short read sequence data. *Genome*, pages 1117–1123, 2009.

[163] Douglas R. Smith, Aaron R. Quinlan, Heather E. Peckham, Kathryn Makowsky, Wei Tao, Betty Woolf, Lei Shen, William F. Donahue, Nadeem Tusneem, Michael P. Stromberg, Donald A. Stewart, Lu Zhang, Swati S. Ranade, Jason B. Warner, Clarence C. Lee, Brittney E. Coleman, Zheng Zhang, Stephen F. McLaughlin, Joel A. Malek, Jon M. Sorenson, Alan P. Blanchard, Jarrod Chapman, David Hillman, Feng Chen, Daniel S. Rokhsar, Kevin J. McKernan, Thomas W. Jeffries, Gabor T. Marth, and Paul M. Richardson. Rapid whole-genome mutational profiling using next-generation sequencing technologies. *Genome Research*, 18(10):1638–1642, 2008.

[164] T F Smith and M S Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–7, March 1981.

[165] Daniel Sommer, Arthur Delcher, Steven Salzberg, and Mihai Pop. Minimus: a fast, lightweight genome assembler. *BMC Bioinformatics*, 8(1):64, 2007.

[166] Li Song, Liliana Florea, and Ben Langmead. Lighter: fast and memory-efficient sequencing error correction without counting. *Genome Biology*, 15(11):509, 2014.

[167] Luz Mayela Soto-Jimenez, Karel Estrada, and Alejandro Sanchez-Flores. GARM: Genome Assembly, Reconciliation and Merging Pipeline. *Current Topics in Medicinal Chemistry*, 14(3):418–424, 2014.

[168] Hayssam Soueidan, Florence Maurier, Alexis Groppi, Pascal Sirand-Pugnet, Florence Tardy, Christine Citti, Virginie Dupuy, and Macha Nikolski. Finishing bacterial genome assemblies with Mix. *BMC Bioinformatics*, 14(Suppl 15):S16, 2013.

[169] R Staden. A new computer method for the storage and manipulation of DNA gel reading data. *Nucleic Acids Research*, 8(16):3673–3694, August 1980.

[170] Paweł Stankiewicz and James R. Lupski. Structural Variation in the Human Genome and its Role in Disease. *Annual Review of Medicine*, 61(1):437–455, 2010. PMID: 20059347.

[171] David Stoddart, Andrew J. Heron, Ellina Mikhailova, Giovanni Maglia, and Hagan Bayley. Single-nucleotide discrimination in immobilized DNA oligonucleotides with a biological nanopore. *Proceedings of the National Academy of Sciences*, 106(19):7702–7707, 2009.

[172] Henrik Stranneheim, Max Käller, Tobias Allander, Björn Andersson, Lars Arvestad, and Joakim Lundeberg. Classification of DNA sequences using Bloom filters. *Bioinformatics*, 26(13):1595–1600, 2010.

[173] Granger G. Sutton, Owen White, Mark D. Adams, and Anthony R. Kerlevage. TIGR Assembler: A New Tool for Assembling Large Shotgun Sequencing Projects. *Genome Science and Technology*, 1(1):9–19, January 1995.

[174] Harold Swerdlow, Shaole Wu, Heather Harke, and Norman J. Dovichi. Capillary gel electrophoresis for DNA sequencing: Laser-induced fluorescence detection with the sheath flow cuvette. *Journal of Chromatography A*, 516(1):61 – 67, 1990.

[175] Adam Szalkowski, Christian Ledergerber, Philipp Krahenbuhl, and Christophe Dessimoz. SWPS3 - fast multi-threaded vectorized Smith-Waterman for IBM Cell/B.E. and x86/SSE2. *BMC Research Notes*, 1(1):107, 2008.

[176] J. Tarhio and E. Ukkonen. A Greedy Approximation Algorithm for Constructing Shortest Common Superstrings. *Theor. Comput. Sci.*, 57(1):131–145, April 1988.

[177] R. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[178] John F. Thompson and Kathleen E. Steinmann. *Single Molecule Sequencing with a HeliScope Genetic Analysis System*. John Wiley & Sons, Inc., 2010.

[179] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[180] Esko Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64(1 - 3):100 – 118, 1985. International Conference on Foundations of Computation Theory.

[181] J. Craig Venter, Mark D. Adams, Eugene W. Myers, Peter W. Li, Richard J. Mural, Granger G. Sutton, Hamilton O. Smith, Mark Yandell, Cheryl A. Evans, Robert A. Holt, Jeannine D. Gocayne, Peter Amanatides, Richard M. Ballew, Daniel H. Huson, Jennifer Russo Wortman, Qing Zhang, Chinnappa D. Kodira, Xiangqun H. Zheng, Lin Chen, Marian Skupski, Gangadharan Subramanian, Paul D. Thomas, Jinghui Zhang, George L. Gabor Miklos, Catherine Nelson, Samuel Broder, Andrew G. Clark, Joe Nadeau, Victor A. McKusick, Norton Zinder, Arnold J. Levine, Richard J. Roberts, Mel Simon, Carolyn Slayman, Michael Hunkapiller, Randall Bolanos, Arthur Delcher, Ian Dew, Daniel Fasulo, Michael Flanigan, Liliana Florea, Aaron Halpern, Sridhar Hannenhalli, Saul Kravitz, Samuel Levy, Clark Mobarry, Knut Reinert, Karin Remington, Jane Abu-Threideh, Ellen Beasley, Kendra Biddick, Vivien Bonazzi, Rhonda Brandon, Michele Cargill, Ishwar Chandramouliswaran, Rosane Charlab, Kabir Chaturvedi, Zuoming Deng, Valentina Di Francesco, Patrick Dunn, Karen Eilbeck, Carlos Evangelista, Andrei E. Gabrielian, Weiniu Gan, Wangmao Ge, Fangcheng Gong, Zhiping Gu, Ping Guan, Thomas J. Heiman, Maureen E. Higgins, Rui-Ru Ji, Zhaoxi Ke, Karen A. Ketchum, Zhongwu Lai, Yiding Lei, Zhenya Li, Jiayin Li, Yong Liang, Xiaoying Lin, Fu Lu, Gennady V. Merkulov, Natalia Milshina, Helen M. Moore, Ashwinikumar K Naik, Vaibhav A. Narayan, Beena Neelam, Deborah Nusskern, Douglas B. Rusch, Steven Salzberg, Wei Shao, Bixiong Shue, Jingtao Sun, Zhen Yuan Wang, Aihui Wang, Xin Wang, Jian Wang, Ming-Hui Wei, Ron Wides, Chunlin Xiao, Chunhua Yan, Alison Yao, Jane Ye, Ming Zhan, Weiqing Zhang, Hongyu Zhang, Qi Zhao, Liansheng Zheng, Fei Zhong, Wenyan Zhong, Shiaoping C. Zhu, Shaying Zhao, Dennis Gilbert, Suzanna Baumhueter, Gene Spier, Christine Carter, Anibal Cravchik, Trevor Woodage, Feroze Ali, Huijin An, Aderonke Awe, Danita Baldwin, Holly Baden, Mary Barnstead, Ian Barrow, Karen Beeson, Dana Busam, Amy Carver, Angela Center, Ming Lai Cheng, Liz Curry, Steve Danaher, Lionel Davenport, Raymond Desilets, Susanne Dietz, Kristina Dodson, Lisa Doup, Steven Ferriera, Neha Garg, Andres Gluecksmann, Brit Hart, Jason Haynes, Charles Haynes, Cheryl Heiner, Suzanne Hladun, Damon Hostin, Jarrett Houck, Timothy Howland, Chinyere Ibegwam, Jeffery Johnson, Francis Kalush, Lesley Kline, Shashi Koduru, Amy Love, Felecia Mann, David May, Steven McCawley, Tina McIntosh, Ivy McMullen, Mee Moy, Linda Moy, Brian Murphy, Keith Nelson, Cynthia Pfannkoch, Eric Pratts, Vinita Puri, Hina Qureshi, Matthew Reardon, Robert Rodriguez, Yu-Hui Rogers, Deanna Romblad, Bob Ruhfel, Richard Scott, Cynthia Sitter, Michelle Smallwood, Erin Stewart, Renee Strong, Ellen Suh, Reginald Thomas, Ni Ni Tint, Sukyee Tse, Claire Vech, Gary Wang, Jeremy Wetter, Sherita Williams, Monica Williams, Sandra Windsor, Emily Winn-Deen, Keriellen Wolfe, Jayshree Zaveri, Karena Zaveri, Josep F. Abril, Roderic GuigÃş, Michael J. Campbell, Kimmen V. Sjolander, Brian Karlak, Anish Kejariwal, Huaiyu Mi, Betty Lazareva, Thomas Hatton, Apurva Narechania, Karen Diemer, Anushya Muruganujan, Nan Guo, Shinji Sato, Vineet Bafna, Sorin Istrail, Ross Lippert, Russell Schwartz, Brian

Walenz, Shibu Yooseph, David Allen, Anand Basu, James Baxendale, Louis Blick, Marcelo Caminha, John Carnes-Stine, Parris Caulk, Yen-Hui Chiang, My Coyne, Carl Dahlke, Anne Deslattes Mays, Maria Dombroski, Michael Donnelly, Dale Ely, Shiva Esparham, Carl Fosler, Harold Gire, Stephen Glanowski, Kenneth Glasser, Anna Glodek, Mark Gorokhov, Ken Graham, Barry Gropman, Michael Harris, Jeremy Heil, Scott Henderson, Jeffrey Hoover, Donald Jennings, Catherine Jordan, James Jordan, John Kasha, Leonid Kagan, Cheryl Kraft, Alexander Levitsky, Mark Lewis, Xiangjun Liu, John Lopez, Daniel Ma, William Majoros, Joe McDaniel, Sean Murphy, Matthew Newman, Trung Nguyen, Ngoc Nguyen, Marc Nodell, Sue Pan, Jim Peck, Marshall Peterson, William Rowe, Robert Sanders, John Scott, Michael Simpson, Thomas Smith, Arlan Sprague, Timothy Stockwell, Russell Turner, Eli Venter, Mei Wang, Meiyuan Wen, David Wu, Mitchell Wu, Ashley Xia, Ali Zandieh, and Xiaohong Zhu. The Sequence of the Human Genome. *Science*, 291(5507):1304–1351, 2001.

[182] Francesco Vezzi, Cristian Del Fabbro, Alexandru I. Tomescu, and Alberto Policriti. rNA: a fast and accurate short reads numerical aligner. *Bioinformatics*, 28(1):123–124, 2012.

[183] Francesco Vezzi, Giuseppe Narzisi, and Bud Mishra. Feature-by-Feature — Evaluating De Novo Sequence Assembly. *PLoS ONE*, 7(2):e31002, February 2012.

[184] Francesco Vezzi, Giuseppe Narzisi, and Bud Mishra. Reevaluating Assembly Evaluations with Feature Response Curves: GAGE and Assemblathons. *PLoS ONE*, 7(12):e52210, 12 2012.

[185] R. Vicedomini, F. Vezzi, S. Scalabrin, L. Arvestad, and A. Policriti. GAM-NGS: genomic assemblies merger for next generation sequencing. *BMC Bioinformatics*, 14(Suppl 7):S6, 2013.

[186] R. Vicedomini, F. Vezzi, S. Scalabrin, L. Arvestad, and A. Policriti. Hierarchical Assembly of Pools. In Francisco Ortuño and Ignacio Rojas, editors, *Bioinformatics and Biomedical Engineering*, volume 9044 of *Lecture Notes in Computer Science*, pages 207–218. Springer International Publishing, 2015.

[187] René L. Warren, Granger G. Sutton, Steven J. M. Jones, and Robert A. Holt. Assembling millions of short DNA sequences using SSAKE. *Bioinformatics*, 23(4):500–501, 2007.

[188] J. D. Watson and F. H. C. Crick. Molecular Structure of Nucleic Acids: A Structure for Deoxyribose Nucleic Acid. *Nature*, 171(4356):737–738, April 1953.

[189] Peter Weiner. Linear pattern matching algorithms. In *Switching and Automata Theory, 1973. SWAT '08. IEEE Conference Record of 14th Annual Symposium on*, pages 1–11, Oct 1973.

[190] Alejandro Wences and Michael Schatz. Metassembler: merging and optimizing de novo genome assemblies. *Genome Biology*, 16(1):207, 2015.

[191] David A. Wheeler, Maithreyan Srinivasan, Michael Egholm, Yufeng Shen, Lei Chen, Amy McGuire, Wen He, Yi-Ju Chen, Vinod Makhijani, G. Thomas Roth, Xavier Gomes, Karrie Tartaro, Faheem Niazi, Cynthia L. Turcotte, Gerard P. Irzyk, James R. Lupski, Craig Chinault, Xing-zhi Song, Yue Liu, Ye Yuan, Lynne Nazareth, Xiang Qin, Donna M. Muzny, Marcel Margulies, George M. Weinstock, Richard A. Gibbs, and Jonathan M. Rothberg. The complete genome of an individual by massively parallel DNA sequencing. *Nature*, 452(7189):872–876, April 2008.

[192] Thomas D. Wu and Colin K. Watanabe. GMAP: a genomic mapping and alignment program for mRNA and EST sequences. *Bioinformatics*, 21(9):1859–1875, 2005.

[193] Guohui Yao, Liang Ye, Hongyu Gao, Patrick Minx, Wesley C. Warren, and George M. Weinstock. Graph accordance of next-generation sequence assemblies. *Bioinformatics*, 2011.

[194] C. Ye, C. Hill, J. Ruan, Zhanshan, and Ma. DBG2OLC: Efficient Assembly of Large Genomes Using the Compressed Overlap Graph. *ArXiv e-prints*, October 2014.

[195] Chengxi Ye, Zhanshan Ma, Charles Cannon, Mihai Pop, and Douglas Yu. Exploiting sparseness in de novo genome assembly. *BMC Bioinformatics*, 13(Suppl 6):S1, 2012.

[196] Daniel R Zerbino and Ewan Birney. Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome research*, 18(5):821–9, May 2008.

[197] Daniel R. Zerbino, Gayle K. McEwen, Elliott H. Margulies, and Ewan Birney. Pebble and Rock Band: Heuristic Resolution of Repeats and Scaffolding in the Velvet Short-Read *de Novo* Assembler. *PLoS ONE*, 4(12):e8407, 12 2009.

[198] Guofan Zhang, Xiaodong Fang, Ximing Guo, Li Li, Ruibang Luo, Fei Xu, Pengcheng Yang, Linlin Zhang, Xiaotong Wang, Haigang Qi, Zhiqiang Xiong, Huayong Que, Yinlong Xie, Peter W. H. Holland, Jordi Paps, Yabing Zhu, Fucun Wu, Yuanxin Chen, Jiafeng Wang, Chunfang Peng, Jie Meng, Lan Yang, Jun Liu, Bo Wen, Na Zhang, Zhiyong Huang, Qihui Zhu, Yue Feng, Andrew Mount, Dennis Hedgecock, Zhe Xu, Yunjie Liu, Tomislav Domazet-Loso, Yishuai Du, Xiaoqing Sun, Shoudu Zhang, Binghang Liu, Peizhou Cheng, Xuanting Jiang, Juan Li, Dingding Fan, Wei Wang, Wenjing Fu, Tong Wang, Bo Wang, Jibiao Zhang, Zhiyu Peng, Yingxiang Li, Na Li, Jinpeng Wang, Maoshan Chen, Yan He, Fengji Tan, Xiaorui Song, Qiumei Zheng, Ronglian Huang, Hailong Yang, Xuedi Du, Li Chen, Mei Yang, Patrick M. Gaffney, Shan Wang, Longhai Luo, Zhicai She, Yao Ming, Wen Huang, Shu Zhang, Baoyu Huang, Yong Zhang, Tao Qu, Peixiang Ni, Guoying Miao, Junyi Wang, Qiang Wang, Christian E. W. Steinberg, Haiyan Wang, Ning Li, Lumin Qian, Guojie Zhang, Yingrui Li, Huanming Yang, Xiao Liu, Jian Wang, Ye Yin, and Jun Wang. The oyster genome reveals stress adaptation and complexity of shell formation. *Nature*, 490(7418):49–54, October 2012.

[199] Shiguo Zhou, Michael Bechner, Michael Place, Chris Churas, Louise Pape, Sally Leong, Rod Runnheim, Dan Forrest, Steve Goldstein, Miron Livny, and David Schwartz. Validation of rice genome sequence by optical mapping. *BMC Genomics*, 8(1):278, 2007.

[200] Aleksey V Zimin, Douglas R Smith, Granger Sutton, and James a Yorke. Assembly reconciliation. *Bioinformatics (Oxford, England)*, 24(1):42–5, January 2008.