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

Ph.D. Thesis

Next Generation Sequencing Revolution Challenges: Search, Assemble, and Validate Genomes

Candidate:
Francesco Vezzi

Supervisor:
Professor Alberto Policriti

Referee:
Professor Graziano Pesole
Professor Lars Arvestad

March 11, 2012

Author's Web Page: //users.dimi.uniud.it/~francesco.vezzi/

Author's e-mail: vezzi@appliedgenomics.org

Author's address:

Dipartimento di Matematica e Informatica
Università degli Studi di Udine
Via delle Scienze, 206
33100 Udine
Italia

To my parents Piera and Fulvio, my little sister Elia, and to my girlfriend Mary

Contents

List of Figures

List of Tables

List of Algorithms

Preface

> A revolution (from the Latin *revolutio*, "a turn around") is a fundamental change in power or organizational structures that takes place in a relatively short period of time.
>
> Wikipedia

World and human beings have witnessed a large number of *revolutions*. Revolutions are a constant denominator in improving and enhancing human rights and possibilities. Social revolutions like the *Protestant Revolt* (1517) or the *French Revolution* (1789) have reshaped the social hierarchies and overthrown political environments though unchangeable. Technology revolutions like the *Printing Revolution* (1440) and the *Industrial Revolution* (18th century) improved lives of thousand of hundreds of people thanks to the introduction of new techniques and instruments and the possibility to spread the new Knowledge. Some revolutions allowed people to express more freely than before (*Egyptian revolution*), some others utterly changed the way in which people communicate (*Web Revolution*).

Revolutions allow to improve and enhance our societies, flattening social differences, reducing physical distance among people and extending life expectations. Today, 10 years after the official entrance in the 21th century, we are witnessing another revolution that will reshape our vision of world, society and life: the *Personal Genomics Revolution*.

Personal genomics is the branch of *Genomics* responsible of the sequencing and analysis of the genome of an individual. Personal genomics is the final goal of a complex process that starts with the *sequencing* of an organisms, passes through the *assembling* and the *characterization* of the individual being sequenced, and proceeds with the comparison of different individuals and the determination of important traits and functions.

Personal Genomics will allow to answer a large number of questions about mankind and about the entire biosphere. The possibility to routinely sequence one's genome will allow not only to design personal drugs, but also to foresee future diseases that may be prevented with target treatments.

The outbreak of Next Generation Sequencing (NGS) Technologies at the beginning of 2005, allowed to realize in few years half of this revolution. We are now able to sequence and analyse a genome belonging to a living organism (a virus, a bacteria, a man) in a couple of weeks, if not days, and at a cost that will be soon affordable for public health systems. However, a second half of the revolution, maybe the most important one, is still missing: use produced data and extract from it all valuable information in a efficient way is still a major problem.

New sequencing technology distinguish themselves from old ones for a dramatic increase in data production. This large amount of data proposed to the Computer Science community new (and old) computational and algorithmical challenges. Problems believed solved became suddenly practically difficult (*e.g.*, string alignment problem), other problems close to approximate satisfiable solutions needed to be re-formulated and re-analysed due to new data (*e.g.*, *de novo* assembly problem), moreover new problems appeared has a consequence of the NGS outbreak (*e.g.*, assembly validation).

In order to fulfil the Personal Genomics goals we need to provide to the research community instruments (both theoretical and practical). We need to study and fully understand the open computational problems and to provide practically useful tools to continue the research. New problems are undermining basics Computer Science concepts: for example non asymptotic optimal

algorithms performs, in practice, better than optimal ones, moreover proved NP-complete problems can be practically solved by greedy procedures. Those are only two examples that suggests how complexity analysis lacks in fully describing algorithms and problem complexity.

The work in this Thesis aims at giving to the genomic community a contribution towards the fulfilment of the Personal Genomics Revolution. We faced, under different perspectives two of the most pressing and challenging problems of today's genomics: string alignment and de novo assembly.

The Personal Genomic Revolution started in 2005 and proceed at a fast pace. Our duty is to design tools that in the next years will be used to enhance mankind life. Working in the NGS-field allows to interact with a global community, whose open problems are rapidly changing and evolving. The giant leap towards Personal Genomics that happened and is still in progress in these days will redefine our concept of medicine and of life. The work done in this period, by the global community and partially by us, will contribute to realize a revolution that will improve humanity.

Introduction

The first genome has been sequenced in 1975 [158] and from this first success sequencing technologies have significantly improved, with a strong acceleration in the last few years. Today these technologies allow us to read (huge amounts of) contiguous DNA stretches (named *reads*) that are the key to reconstruct the genome sequence of new species, of an individual within a population, or to study the expression levels of single cell lines. Even though a number of different applications exploit sequencing data today, the "highest" sequencing goal is always the reconstruction of the complete genome sequence. The success in determining the first human genome sequence has encouraged many groups to tackle the problem of reconstructing the code-book of others species, including microbial, mammalian, and plant genomes.

The sequencing process has been a slow and relatively expensive procedure until few years ago. Recently, new sequencing methods, known under the name of *Next Generation Sequencing* (NGS) technologies, have emerged. In particular the commercially available NGS technologies include pyrosequencing (commercialized by 454), sequencing by synthesis (commercialized by Illumina) and sequencing by ligation (commercialized by ABI). Compared to traditional methods (in particular Sanger method), these technologies function with significantly lower production costs and much higher throughput. These advances have significantly reduced the cost of several applications having sequencing or *resequencing* as an intermediate step. In particular the possibility to sequence or resequence several individuals among a population became possible and affordable even by small research facilities.

Despite the differences among them, NGS technologies differ from old sequencing technologies for three fundamental peculiarities:

- low cost: the cost of resequecing a human individual dropped from more than 1 million US dollars to 10.000 US dollars;
- high throughput: data production passed from a couple of mega bases per day to tens of Giga bases per day;
- short sequences: produced sequences (*i.e.*, *reads*) are much shorter than previous ones.

The roots of this dissertation are in the recent research revolution that followed the NGS appearance usually known under the name of NGS-revolution (the main engine of the more general Personal Genomics revolution). In particular we are interested in the tight link between the NGS-revolution and Computer Science. From the pioneering genomics projects (*i.e.*, Human Genome Projects [85, 175]) the role of computer scientists has been of primary importance. In particular, in the last decade the new discipline of *bioinformatics* clearly emerged. Bioinformatics is the application of computer science and information technology to the field of biology and medicine. Bioinformatics deals with algorithms, databases and information systems, web technologies, artificial intelligence, information and computation theory, software engineering, data mining, image processing, modeling and simulation, signal processing, discrete mathematics, control and system theory, circuit theory, and statistics, for generating new knowledge of biology and medicine, and improving and discovering new models of computation (*e.g.*, DNA computing, neural computing).

In this thesis, we will focus our attention on genomics and *sequence analysis* in the NGS context. Genomics is the discipline in genetics concerning the study of the genomes of organisms. Every genomic project starts by sequencing an organism. This process produces a large amount of reads that are subsequently used in the following analysis. When an unknown (*i.e.*, never sequenced before) genome is sequenced, we are usually interested in reconstructing its DNA sequence. In this case we speak of *de novo assembly* project. The process of reconstructing a genome is called *assembling* and it is carried out by pipelines dubbed *assemblers*. If the sequenced individual has

an already assembled genome, we are usually interested in discovering the differences between the assembled data and the just sequenced organism. In this case we speak of *resequencing* project. The assembled genome is referred to as the *reference genome*. The first mandatory step in these projects is to *map* or to *align* sequenced reads on the reference genome. This step is accomplished using tools dubbed *aligners*.

Throughout this thesis, we will see how often standard complexity analysis fails to faithfully depict the proposed problems. The alignment problem is a well known and studied problem, with several optimal solutions. Unexpectedly, more often than not, non asymptotically optimal algorithms have, in practice, better performances than asymptotically optimal ones. The assembly problem is even more surprisingly: even if all the formulations proposed so far categorize the problem as NP-hard, tools able to practically solve the problem are widely accepted and used. As we will see, this is a consequence of several factors: (*i*) real architectures are more oriented towards some solutions than others (*i.e.*, memory locality); (*ii*) complexity analysis ignores significant constants which turn out fundamental in practical scenarios (*e.g.*, use 50 GB RAM or 100 GB RAM means more than just discarding a constant factor of 2); (*iii*) complexity studies often introduce over simplifications that alter the problem's nature; (*iv*) complexity studies concentrate on worst case scenarios that, however, can be rare in practice or absent in nature.

All the solutions proposed in this thesis aim at contributing to the progress of the bioinformatics and genomic fields. Some of the contributions represent brand new ideas that have been bundled in publicly available tools (this is the case of *rNA*, *GapFiller*, and *16merCounter*), some other are the result of combining third party software to produce new tools or pipelines (this is the case of *eRGA*), in some other cases already proposed techniques have been critically studied and revised (this is the case of *forensics features analysis*).

The NGS revolution technology allowed to sequence at an extremely low cost and at high coverage an unexpected number of new organisms. This is well represented throughout this thesis by the numerous datasets that will be used to test and compare already available and new solutions. Among others, in our work we analysed different Human datasets, numerous grapevine varieties, several poplar species, data belonging to conifers (*i.e.*, spruce) and a large number of bacterial genomes. Pushed by the need to evaluate software capabilities, we also produced and employed *in-vitro/simulated* datasets.

The NGS-revolution not only confined Sanger based sequencing to the history, but it also ruled the end of several software solutions believed not replaceable for decades. Next generation sequencers are able to produce in a couple of weeks the same amount of data produced for the Human Genome Projects in 10 years. From the beginning it was clear that all available instruments were not able to cope with this large amount of data. The picture was even—computationally—worse, due to the fact that NGS sequences were characterized by an intrinsic short length and by new and almost unknown errors schemas.

The unquestioned aligner before the NGS revolution was BLAST. BLAST was unable to manage short reads and to align them in an acceptable amount of time. In the new NGS world, hundreds of millions of reads are produce within days and should be aligned in days if not hours. At the same time in which NGS vendors were busy in a race to produce more data at a lower cost than the others, the Computer Science community was busy in a race to produce faster aligners able to align data at higher throughput and consuming always less resources. Almost all proposed solutions are based on *indexes* over the reference used to quickly filter large portions of it and speed up the search. Beside the theoretical optimal performances of Suffix-Tree like solutions, all practical algorithms rely on Suffix-Arrays variations and Hash Tables. In particular, solutions based on Burrows Wheeler transformation and FM indexes (Ferragina and Manzini) have became a *de facto* standard in the NGS alignment landscape. It is worth noting that Suffix-Arrays and Hash Tables behave theoretically worse than Suffix-Trees. However, Suffix-Trees require more (by a constant) memory's accesses due to bad memory locality. The constants involved become not negligible when aligning hundreds of thousands of sequences.

In addition to the technicalities of each solution, all available software must deal with errors, that are an inherently characteristic of every sequencing technology. Allowing errors in the align-

ment process is therefore mandatory. Inexact alignment is much slower than the exact one, so many tools employ heuristics to speed up alignment at the cost of loss of accuracy (*i.e.*, missing optimal alignments).

One of the main contributions presented in this work is the short string aligner rNA (randomized Numerical Aligner). rNA is an hash-based aligner specifically designed to align the large amount of data produced by NGS technologies, with particular attention to Illumina technology. rNA uses an *Hamming-aware* mismatch function that allows to identify reference's positions that are likely to be occurrences of the searched pattern at a given Hamming distance. The key factor that distinguishes rNA from the vast majority of others aligners is its capability to align at high sensitivity without significantly affecting performances. All developed heuristics do not hinder rNA's sensitivity and no optimal alignment is missed by this tool.

Discouraged by short reads length, but encouraged by technology improvements, many groups have started to use NGS data in order to reconstruct new genomes from scratch. *De novo* assembly is in general a difficult problem and is made even more challenging by short reads. Assembly problem has been studied under several prospectives and all formalization attempts demonstrated that it is an NP-hard problem. However, many assembler have been proposed with particular success, especially in the Sanger sequencing context. NGS-based assemblers are almost always based on de Bruijn Graphs, that on one hand made the problem practically tractable, but on the other hand introduce over simplifications that can hamper the resulting assembly. NGS-based assemblers results are, until now, not comparable to those achievable with Sanger-based assemblers. The possibility to trade reads' length with reads' coverage (*i.e.*, the amount of data being assembled) seems, for now, not worth the deal.

Under certain assumptions, *de novo* assembly can became an easier and more tractable problem. This is the case, for example, of the assembly of individuals in presence of a reference genome belonging to a biologically related organism. In this case the reference genome can be used as a guide to reconstruct the new one, avoiding many common *de novo* assembly problems. The assembly problem picture can became tractable also if we concentrate our efforts on a small portion of the sequence, instead of considering the all picture at once. This is the idea behind several attempts to sequence and subsequently assemble small DNA's portions (*e.g.*, fosmid/BAC pools).

In this thesis we will discuss eRGA (enhanced Reference Guided Assembly) a pipeline that aims at integrating *de novo* assembly and alignment. In presence of a closely related reference sequence, one can align the sequenced reads against the available reference and produce a *consensus* sequence. The draw back of this technique (usually named Reference Guided Assembly) is the fact that only similar/conserved regions are reconstructed. *De novo* assembly, on the other hand, may be unable to produce an acceptable quality assembly to be used to perform subsequent analysis. eRGA, with the help of a reference sequence, integrates reference guided and *de novo* assembly to overcame the limits of both, while retaining their advantages.

Usually, as a natural consequence of the sequencing process, reads are provided in pairs that are at a known (estimated) distance. Such distance is usually refereed as the *insert size*. The assembly process starts, one way or another, by computing overlaps between pairs of reads and using this information to build contiguous sequences (*contigs*). The process of building contigs is error prone (misassemblies) as a consequence of both sequencing errors and presence of repeated sequences in the genome. With the goal of partially solving *de novo* assembly problem we proposed to limit the problem to the assembly of the missing insert between pairs of reads. GapFiller (GF) is a local assembler whose aim is "only" to close the gap between paired reads. Contigs produced in this way are much shorter than those returned by standard *de novo* assemblers, however GF returns contigs that are "certified" to be correct as a consequence of the fact that insert has been closed reaching the mated read at the expected distance.

As a consequence of the NP intractability of the assembly problem, heuristics and greedy approaches are standard ingredients in almost all assemblers. ARACHNE (Broad Institute MIT) assembler binaries (one of the assemblers used to assemble the Human genome) had size greater than 3 GB: the Human genome size! Despite the classical remark that software used to assemble the

Human genome was at least as complex as the genome itself, it is clear that no one in reality exactly knows what choices and what heuristics are really applied by an assembler. As a consequence of this, assembly validation is an important step in every genome project. Moreover, validation steps must be done as part of all the assembly projects in order to be sure to work with reliable data. Validation and filtering steps are fundamental to reduce the problem's complexity: every procedure able to reduce the dataset complexity, suggest the right tool to use, and/or compare assemblers performances and assembly's correctness, is fundamental for the overall quality of the final goal.

To accomplish the (practically) important *de novo* assembly pre-steps of read filtering and read evaluation we implemented two useful tools able to carry out these tasks. As far as the read filtering concerns, we add to rNA a functionality that allows to efficiently filter reads and to discard those that are likely to contain errors or that are contaminated (*i.e.*, belonging to an organism different from the one being sequenced). For what concerns read evaluation, we implemented 16merCounter a tool able to quickly count 16-mers (*i.e.*, substrings of length 16) composing the reads being sequenced and assembled. This tool allows to estimate not only genome composition, but also to know how complex it is.

Gauging assemblers results or compare the performances of different assemblers is a problem probably as difficult as the assembly problem itself. In this thesis we proposed a critical assessment of the various techniques used to validate and compare assemblies and assemblers. As an innovative contribution we will show how a selected number of features can be used to validate and compare assemblers results and how some standardly accepted features do not faithfully represent correctness. The selected and studied features can be used to compare and evaluate assemblies and assemblers using previously proposed solutions. We believe that these specific results can be of utmost practical importance for the community, as they partially fill the gap of assembly evaluation: one of the most pressing problems of today's genomics.

Throughout all the thesis we will analyse problems characterized by a strange trade-off between theoretical complexity results and practical performances. In particular we will see how theoretical optimal algorithms and complexity results are, in some way, contradicted by the use of non optimal algorithm or by greedy techniques that practically solve the problems

Our tour in the NGS bioinformatic world starts with an introduction to basic biological concepts and to NGS technologies in Chapter 1. The aim of this Chapter is to introduce non biologist readers to the complex bioinformatics' world. We will focus our attention not only on basic DNA concepts, but also on NGS technologies.

Part I is dedicated to the alignment problem in the NGS-era. In Chapter 2 we will analyse algorithms and data structure proposed so far to efficiently align the huge amount of short sequences against a reference genome.

Chapter 3, instead, presents one of the main contributions of this thesis: a new public available aligner for NGS sequences dubbed rNA. The aim of the Chapter is not only to describe the basic concepts of the software, but also to show the evolution of a tool from a basic idea to a complete suite in the NGS *panta rhei*.

Part II is dedicated to the *de novo* assembly problem. Chapter 4 gives an overview of the assembly problem and of the available solutions: we will discuss complexity analysis (Section 4.1), describe available tools to solve the problem (Section 4.2), explain limits of available validation techniques (Section 4.3), and see results achievable with state of the art instruments and data (Section 4.4).

Chapter 5 describes two new contributions to solve *de novo* assembly problem under restricted hypothesis that naturally simplify the problem. In Section 5.1 we will present eRGA a pipeline able to combine alignment and *de novo* assemblly techniques to assemble genomes in presence of an already available reference sequence. In Section 5.2 we will introduce GapFiller, a local assembler that aims at producing relative short contigs that are certified to be correct.

In Chapter 6 we will present the contributions to the assembly validation problem. In particular we will see how the information stored in reads can be improved (Section 6.1.1) and how reads can be used to infer genome's properties or to evaluate experiments' quality (Section 6.1.2). Moreover, in Section 6.2 we will show how the use of multivariate techniques allows us to better understand

metrics used in assembly evaluation/validation and how these techniques allows us to improve available methods.

1

A "bit" of Biology

Herein we introduce some basic biological information. In particular, after presenting the principal genomics concepts (nucleotide, DNA, genome, etc.) we will focus our attention on the sequencing process, that is the focal process that allows us to "read" genomes. The process of *reading/sequencing* a genome is the first mandatory step to reconstruct, search, and analyse it.

In particular, this first chapter aims at explaining how, in the last three years, a revolution took place in the sequencing projects: at least three new technologies, sponsored by three large companies, reshaped our knowledge and capabilities in the genomic field. The Human Genome Project [85, 175] needed 10 years and more than 10 US Billions dollars to produce all necessary data. The same amount of data can be produced today in three weeks at a cost of 10.000 US dollars. Sequencing costs are constantly dropping-down while the sequencing throughput is improving at a rate of about fivefold per year. At the same time a new generation of sequencing technologies appeared promising, again, lower costs and higher throughput.

A basic understanding of how sequencing technologies work and how they evolved is mandatory to understand some of the computational problems that will be discussed in the following chapters.

In Section 1.1 we will introduce some basics DNA concepts, like its double helix structure. In Section 1.2 we will describe the sequencing process as the general process that allows us to read a DNA sequence. Moreover we will present some of the most important reasons pushing our need of sequencing. Finally in Section 1.3 we will focus on the characteristics and peculiarities of available sequencing technologies. In particular Section 1.3 is divided into three main topics: Old Sequencing technologies (1.3.1), Next or Second Sequencing Technologies (1.3.2) and finally Third or Future Sequencing Technologies (1.3.3).
Readers already familiar with the fundamentals of the DNA and with sequencing technologies may skip this Chapter.

1.1 DNA and the Codebook of Life

DNA, or *deoxyribonucleic acid*, is the hereditary material in humans and in all other organisms. Cells belonging to the same individual contain the same DNA sequence. DNA is commonly located in the cell nucleus (nuclear DNA), however non negligible quantities of DNA can also be found in the mitochondrion (mitochondrial DNA or mtDNA) and other organelles (*e.g.*, like the chloroplast in plants). DNA contains (and hides) the code that not only makes life possible, but makes each individual different from all others. This code can be represented as a long sequence (or chain) containing a message encoded in a four letter alphabet: Adenine (A), Guanine (G), Cytosine (C), and Thymine (T). These letters represent four chemical bases classified into two types: Adenine and Guanine are named *purines*, while Cytosine and Thymine are called *pyrimidines*. Each base is also attached to a sugar molecule and to a phosphate molecule. Base, sugar molecule, and phosphate together are called a nucleotide.

DNA exists in the famous form of a *double-helix* structure (see Figure 1.1) as discovered by the two Nobel Prize scientists Watson and Crick. In living organisms DNA does not usually exists as a single molecule, but instead as a pair of molecules that are held tightly together in a double-helix shape. In a double helix the nucleotides direction in one strand is opposite to their direction in

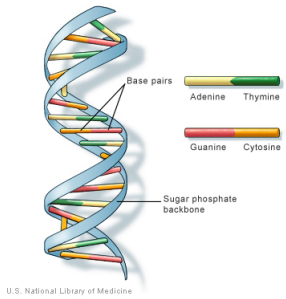


Figure 1.1: DNA is a double helix formed by base pairs attached to a sugar-phosphate backbone.

the other strand: the strands are *antiparallel*. The asymmetric ends of DNA strands are called the five prime ($5'$) and three prime ($3'$) ends.

Bases on opposite strands are bonded, but only two types of connections are possible: Adenine is coupled always to Thymine, while Cytosine is coupled always to Guanine. This situation is called *complementary base pairing*, and we will often use the term *base pair* (bp) to identify one of the two bases. The information contained on the two strands is the same, with the only difference that one is the *reverse complement* copy of the other. In other words, given a sequence belonging to one of the two strands, we have to revert it and to substitute As with Ts and Cs with Gs to obtain the sequence on the other strand.

An important property of DNA is its capacity to replicate, or make copies of itself. The bonds between bases in the two strands can be broken (by a mechanical force or by high temperature for example) and the two strands can be used to obtain two new copies of the original DNA sequence.

The replication process must be nearly perfect, otherwise errors in this phase can jeopardize essential living mechanisms of the individual. However, imperfect replications or insertions in the DNA of foreign genetic materials are essential for the evolution process especially in asexual organisms.

DNA contains most if not all the information needed to describe each aspect of a living organism, however some parts of this sequence are more important than others. In particular, genes are among the most important and studied segments of every genome. Genes contain open reading frames (*i.e.*, a portion of a DNA molecule that, when translated into amino acids, contains no stop codons) that can be *transcribed* and they are normally flanked by regulatory sequences such as promoters and enhancers, which control the transcription. More in particular, using the definition introduced in [136], a gene is a discrete genomic region whose transcription is regulated by one or more promoters and distal regulatory elements and which contains the information for the synthesis of functional proteins or non-coding RNAs, related by the sharing of a portion of genetic information at the level of the ultimate products (proteins or RNAs) .

Genes are the basic units of hereditary information in all living organisms. All activities and behaviours of cells depends on genes. All organisms are characterized by many genes corresponding to different biological traits. Some of those traits are visible (eye color, height, etc.) some others are "hidden" like blood group or predisposition to diseases. When a gene is *active* (or *expressed*) its sequence is copied in a process dubbed *transcription*, producing the so call RNA (*Ribonucleic acid*). Transcription is the first of a series of activities characterizing the *gene expression mechanism*. After transcription, RNA is further processed to remove intronic regions to produce the so called *mature RNA*. Later, mature RNA is *exported* outside the cell nucleus (at least in eukaryotes) in order to *translate* RNA triplets into amino acids that will form the proteins.

It is important to notice that while all cells in an organism share the same DNA sequence, genes *expressed* in each cell can be (utterly) different. Moreover, even if two cells can produce the same protein, the *expression level* can be different (*i.e.*, one of the two produces a small amount of the protein in comparison to the other). Gene expression (and levels) depends on cell types,

developmental stage, response to environmental factors and many others.

1.1.1 Zygosity and Ploidity

The genotype is the genetic makeup of a cell, an organism, or an individual usually with reference to a specific character under consideration. One's genotype differs subtly from one's genomic sequence. A sequence is an absolute measure of base composition of an individual, or a representative of a species or group; a genotype typically implies a measurement of how an individual differs or is specialized within a group of individuals or a species.

DNA is divided into *chromosomes*. All complex living forms (more complex than bacteria) have more than one chromosome. *Ploidy* is the number of sets of chromosomes in a biological cell. An organism is said *haploid* if its cells have a single copy for each chromosome, *diploid* if each chromosome is present in two copies while *polyploid* if the number of copies is higher than two. For example humans, like almost all the animals, are diploid so they have two *homologous* copies of each of the 23 chromosomes, one from the mother and one from the father.

In diploid (and more in general in polyploid) organisms, some segments or *loci* of the same copy of chromosomes can be identical or different. A locus is the specific location of a gene or DNA sequence on a chromosome. A variant of the DNA sequence at a given locus is called an *allele*. If at a given locus the genotype consists of two identical alleles (*i.e.*, the two chromosome copies are identical) then the locus is said *homozygous*. Otherwise, if at a given locus the genotype contains two different alleles, then the locus is said *heterozygous*.

1.2 Sequencing the DNA

DNA sequencing is the process that allows us to *read* continuous stretches of bases from a DNA sequence. A common denominator between all available technologies is their capability to read only small sequences compared to the length of the genome being sequenced.

A *sequencing* process begins by physically breaking the DNA into millions of relatively small fragments. A single fragment cannot be easily read; for this reason, usually, each fragment is replicated several times. Replication allows the *sequencer* to read continuous stretches of DNA usually dubbed *reads*. DNA fragments being read can have various sizes, ranging from 30 base pairs (bp) to 100 Kbp. Almost all commercial available sequencers are able to read both fragments ends, producing in this way the so called *paired reads*. More precisely, reads are read in pairs at a known distance and orientation; the distance between the beginning of the first pair and the end of the second is usually called *insert size*. As we will see in Chapter 3 paired reads are of utmost importance in *de novo assembly* (*i.e.*, reconstructing the original DNA sequence). It is worth pointing out that there is a trade off between insert size and cost (time and money): short inserts are inherently easier and cheaper to produce than long ones.

1.2.1 Why Sequencing?

Sequencing is a difficult, time consuming and expensive procedure, so why are we so interested in sequencing? Sequencing is only the first step along the path to discover and unveil the book of life. The number of applications that follow the sequencing process is very long and here we can only list some of the most important.

Sequencing is essential to comprehensively characterize DNA sequence variation, to detect methylated regions of the genome, to study trancripts and to identify the degree to which mRNA tranctipts are being actively translated [175, 42, 5, 179].

De novo sequencing is the process of reconstructing a new genome sequence. The genome of every organism holds secrets that when unlocked yield a invaluable mechanistic information that would in some measure illuminate not only our own biology but that of the rest of the world as well. The complete genome sequence of an organism (often called reference genome sequence) allows us to study gene expression, genome evolution, cancer mechanisms, and many others aspects.

Obtaining a reference sequence is only the first step towards hundreds of new possibilities. Sequencing is of primary importance when a reference genome is available: in this case sequencing new individuals will allow us to study genetic diseases and to characterize differences among populations. Moreover, having a reference genome for a given species, we can sequence closely related species in order to study the differences among distant individuals.

The final goal of sequencing, however, still remains, at least for humans, the so called *personal genomics*. Variations in the genome are often unique among people. It is estimated that each human being carries about 200,000 single-base variants with respect to the reference genome. The possibility to obtain the genome sequence of an individual will possibly give us the opportunity to create target drugs and/or to prevent diseases.

1.3 Sequencing Technologies

The general technique of breaking DNA into several random fragments and read them after a replication phase has been introduced by Fred Sanger in 1975 [158] (the chain termination method) and in parallel by Maxam and Gilbert in 1977 [112] (a chemical sequencing method). Sanger sequencing ultimately prevailed given it was technically less complex and more amenable to being scaled up. The sequencing method introduced by Sanger, usually dubbed Sanger Method, has remained the mainstay of genome sequencing for nearly 30 years.

Recently, Sanger method has been supplanted by several next-generation sequencing technologies offering dramatic increases in cost-effective sequence throughput, albeit at the expense of read lengths. Starting from 2005 Illumina Solexa, Roche 454 and ABI Solid have been busy in a race to produce more and more sequences, at lower cost and in shorter time, consigning (at least apparently) Sanger method to the history. This major breakthrough is globally known under the name of "Next Generation Sequencing revolution", while the three new technologies are usually called Next Generation Sequencing Technologies. However, in the last months, the name Second Generation Sequencing Technologies is becoming usual, to distinguish the three already mentioned solutions from new technologies, promising again new data at a lower cost and at higher throughput. New emerging technology are often categorized as Single Molecule Sequencing Technology for their capacity to sequence a single molecule without the need of replication steps.

In the following we will describe the basics of major sequencing technologies. All sequencing technologies (even the "old" Sanger sequencing) rely on complex and advanced chemical reactions and/or high resolution optical devices, and/or state of the art nano-technology instruments. The aim of this paragraph is only to sketch the basic principles and to highlight strong and weak points of each technology.

1.3.1 First (Old) Generation Sequencing Technologies

The landmark publication of the late '70 by Sanger's group [158] and notably the development of the chain termination method by Sanger and colleagues [159] established the groundwork for the following decades of sequence-driven research.

The method is based on the DNA polymerase-dependent synthesis of a complementary DNA strand in the presence of natural 2′-deoxynucleotides (dNTPs) and 2′,3′- dide-oxynucleotides (ddNTPs) that serve as nonreversible synthesis terminators [158]. In order to obtain several copies of each fragment being sequenced, an *in vivo* amplification is performed. Usually this amplification is done through cloning into bacterial hosts.

Using their method, Sanger and colleagues in 1977 were able to sequence the phiX174 genome of size 5374 bp [158]. Five years later their method was used to sequence the bacteriophage λ genome consisting of 48501 bp [159]. The method was mainly manual and required an extensive human work. In the following years, Sanger method has been systematically improved and automatized and in 1995 the cost per base dropped to approximately 1 US$ per base pair. However, time and costs needed to sequence even a small bacterial genome (a couple of Mega base pairs) was still too high even for large sequencing centers. The real acceleration towards a method able to scale

on larger genomes was the introduction of the ABI Prism 3700 DNA Analyzer, which allowed the sequencing of approximatively 900,000 bp/day, making the cost per base falling to 0.1 US$ per base pair. Between 1998 and 2005 the costs approximatively dropped to 0.001 US$ per bp, given the possibility to sequence even Giga base pairs long genomes. Reads' length could reach 700 bp using the ABI3730 cl instruments, helping both the assembly and the alignment task.

It must be stressed that even with the progresses just mentioned, production-scale genome sequencing with Sanger technology is only possible at genome centers where there is a large availability of space, personnel, and equipment. As noticed in [27] to sequence at an high enough coverage a mammalian genome (*e.g.*, at 4× coverage) one needs a center with approximately 5,000 m^2, with 150 ABI3730 xl sequencers, a sample preparation area the size of a basketball court, and a 180 m^2 computing facility. Such a center still has to wait for one year before have all the data.

1.3.2 Second (Next) Generation Sequencing Technologies

The previous picture gives us a rough idea of the problems and of the difficulties of a sequencing project at the beginning of 2005. In particular cost and time needed to sequence an individual were a major stumbling block. With the completion of the Human Genome sequence [85, 175] an obvious question was "What's next?" There were (and there are) many answers to this question: we need the genome sequence of new species to better characterize the bio-diversity; we would like to resequence as many individuals in a population (*i.e.*, Human) as possible to characterize and understand genome variations and genetic diseases and to develop pharmacogenomic drugs.

All this targets were far from being possible at the beginning of 2005. As noticed by Margulies in [111] the cost of sequencing a human genome was estimated to lay between 10 US$ and 25 US$ Millions dollars. However the picture changed very fast after 454 sequencing technology appeared on the market [111]. This new technology represented a giant leap towards personal genomics thanks to its large throughput. The scenario became even more interesting in the next few months when other two technologies (Illumina/Solexa and ABI-Solid) appeared promising more data at a lower cost.

On one hand, Next (Second) Generation Sequencing technologies were characterized by large throughput and low costs, on the other hand, read's length posed a major problem. 454 was able to produce reads 200 bp long, Solid and Illumina were limited to reads shorter than 50bp. Moreover, new short reads produced with NGS-technologies were affected by error rates higher than Sanger ones.

Despite these problems, NGS technologies have practically substituted Sanger sequencing. In the following years a competition started with the goal of producing more and more reads, at a lower cost and/or at longer lengths. Illuminas latest instrument, HiSeq2000, is able to produce more than 60 Giga base pairs (Gbp) per day composed of 100 bp long reads. ABI Solid latest instrument (SOLID^{TM} System) can produce 300 Gbp in a single run, but it can generate reads of length at most 75 bp. Roche 454 (Genome Sequencer FLX) has the lowest throughput (1 Gbp per day) but is able to produce single reads of length 600-700 bp or paired reads of length 250 bp. Newer technologies like Ion Torrent, Pacific Bioscience and Oxford Nanopore are now emerging on the market with new instruments promising again new data at a lower cost.

454

One of the main problems and limitations of Sanger sequencing is the *in vivo* amplification of DNA fragments, usually achieved by cloning them into bacterial hosts. This process is affected by host-related biases, is lengthy, and is quite labor intensive. The first next-generation sequencing technology that appeared on the market in 2005 eliminated this problem thanks to a highly efficient *in vitro* DNA amplification method known under the name of emulsion PCR [171]. With this system individual DNA fragments-carrying steptavidin beads are captured into separate emulsion droplets. These droplets act as individual amplification reactors, producing $\sim 10^7$ clonal copies of a unique DNA template per bead (for more details refer to [111]). The material is subsequently transferred into a well of a picotiter. The templates are then analysed using a *pyrosiquencing*

reaction. The use of the picotiter plate allows hundreds of thousands of reactions to happen in parallel, greatly increasing sequencing throughput.
The pyrosequencing technique is a sequencing-by-synthesis approach that measures the release of inorganic pyrophosphate by chemiluminescence. In other words, every time a nucleotide complementary to the template strand is added into a well, the nucleotide is incorporated, thus producing a "light" detected by a CCD camera. The light signal is proportional to the number of bases being incorporated.
For this reason, the principal error of 454 reads consists of incorrectly estimations of the length of homo-polymeric sequences stretches (*i.e.*, continuous stretches of a single nucleotide).

Current state of the art 454 platform is able to produce single 600-700 bp long reads or 250 bp long paired reads. The throughput is approximately of 1 Gbp per day.

Illumina

Solexa company was a pioneer in proposing an utterly new sequencing method [13, 12]. The company was acquired in 2007 by Illumina Inc that started to commercialize the most widely used sequencer nowadays: the Ilumina/Solexa Genome Analyser (GA). At that time, the Illumina sequencer was characterized by an unbelievable throughput (1 Gbp in 2-3 days) and by a problematic short read length (35 bp). In three years Illumina was able to raise the throughput up to 100 times and read length up to 4 times. Recently, the new machine HiSeq2000 is able to produce up to 600 Gbp of data composed by reads of length 100bp in two weeks.

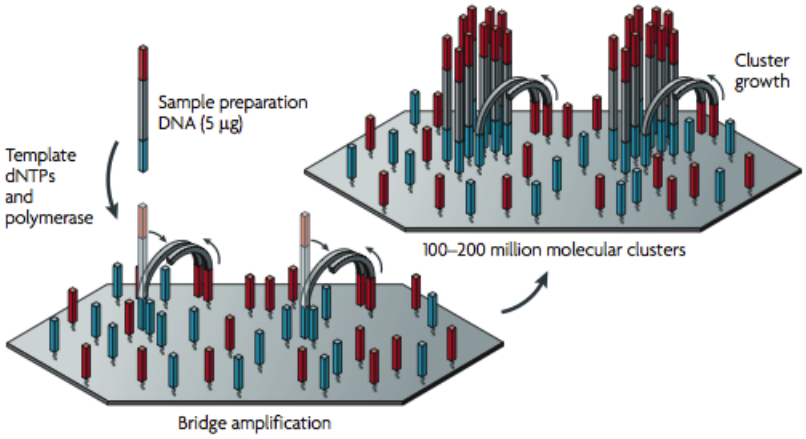


Figure 1.2: Ilumina/Solexa amplification process.

Besides the technical improvements the Illumina/Solexa approach has remained the same. The method achieves cloning-free DNA amplification by attaching single stranded DNA fragments to a solid surface (see Figure 1.2), dubbed *single-molecule array*, and conducting a solid-phase bridge amplification of single-molecule DNA templates. This process consists in attaching a single DNA molecule to a solid surface using an adapter. In this way, molecules bend over and hybridize to complementary adapters (*i.e.*, creating a bridge), forming the template for the synthesis of their complementary strand. In this way a ultra high sequencing *flow-cell* with hundreds of millions of clusters is produced, with each cluster containing thousands of copies of the same template.

Templates are then sequenced in a massively parallel fashion using a DNA sequencing-by-synthesis approach that employs reversible terminators with removable fluorescent moieties. At each cycle a base is read by adding the four nucleotides with an attached fluorescent dye to record the incorporated base. After every cycle the surface must be "cleaned" for the next phase. This procedure is often referred as *wash-and-scan* method.

Illumina method is not affected by the homo-polymeric stretches like 454. However, as a consequence of the wash-and-scan procedure, reading quality drops faster and Illumina is able to produce shorter reads compared to 454 ones, making some problems harder to solve (*i.e.*, *de novo* assembly). The most frequent error in Illumina reads are substitutions, usually present in read's tails as a consequence of the wash-and-scan approach.

Solid

Applied Biosystem has been the unquestioned leader company in the sequencing market for more than 30 years as they were the producers of almost all the sequencers based on Sanger-sequencing method. Despite their experience (or maybe due to their experience) they were unable to foresee Next Generation Sequencing revolution and therefore they were the last company to propose an NGS instrument.

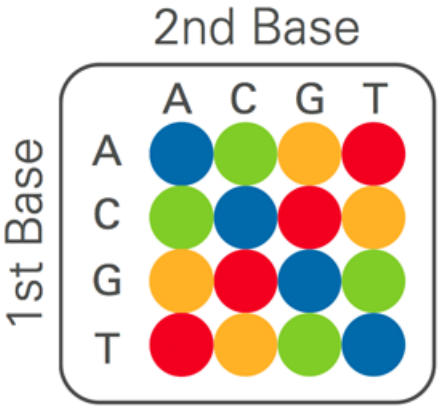


Figure 1.3: Solid reading schema. In Solid system a given color is compatible with four two-base combinations.

In ABI-SOLID system, sample fragments are attached into beads and cloned with emulsion PCR. Amplification products are transferred onto a glass surface where sequencing occurs by sequential rounds of hybridization and ligation with 16 dinucleotides (all possible combination of two consecutive nucleotides) combinations labelled by four different fluorescent dyes (see Figure 1.3). Each position is probed twice and the identity of the nucleotide is determined by analysing the color that results from two successive ligation reactions. However, in order to "translate" the reads from *color space* (the output format) to *letter space* at least the first base must be known (and in general it is). The advantage of such technique is the fact that sequencing errors and sequence polymorphisms (*i.e.*, real changes in the DNA being sequenced with respect to the reference) can be detected.

SOLID instruments are able to produce 300 Gbp of 75 bp long reads in a single run. However, a strong limitation of SOLID technology is the color space encoding, that obliges development of ad-hoc software. This fact, coupled with the short read length have limited the spread of this technology.

1.3.3 Third (Future) Generation Sequencing Technologies

As noticed in [122] the three main sequencer vendors (Roche, Illumina and ABI) are constantly improving their instruments, and year after year (sometimes month after month) reagents price and sequencing times are decreasing while throughput and precision are increasing. In less then three years NGS technologies have reshaped our visions and altered the previsions of the more optimistic prophets.

Even though improvements to second-generation sequencers continue to impress, there is a lot of rumour in the last period about the so called *third generation* DNA sequencing platforms. Their distinguishing factor between these new emerging technologies and the second generations are their capability to use single template molecules, lower cost per base, easy sample preparation and significantly faster run times analysis. Some of these new emerging technologies are able to (or at least aim to) produce long reads, with the goal to solve some *de novo* assembly open problem.

Major Second Generation Sequencing technologies rely on sequencing by synthesis approach that needs PCR to grow clusters of a given DNA template, a solid surface where to attach templates that are subsequently read by synthesis in a phased approach that requires synchronization and many washing steps. Third Generation Sequencing technologies, instead, interrogate a single DNA molecule: in this way no synchronization is required and issues related to the bias introduced by PCR amplification and de-phasing are overcome.

This new sequencing technologies promise (in some way "again") high throughput, fast turn-around times (sequencing bacterial genomes in minutes, not weeks), long reads' length to enhance *de novo* assembly and structural variations detection (especially in haplotypes), high consensus accuracy, small amounts of starting material (theoretically a single molecule) and low costs of sequencing machines and of sequencing process (having in mind the ambitious goal of sequencing a human genome with less than 1000 US$) [160].

In the following we will describe some of the Third Generation Sequencing Technologies already or soon available on the market.

IonTorrent

Ion Torrent technology is considered to sit between Second Generation Sequencing Technologies and Third Generation Sequencing Technologies [160]. Ion Torrent is based on semiconductor-based high-density array of microwells working as reaction chambers (see Figure 1.4) [155]. This technology eliminates the need for light, scanning and cameras. These facts not only dramatically simplify and accelerate the overall sequencing process but they also reduce both the overall footprint of the instruments and the sequencing costs.

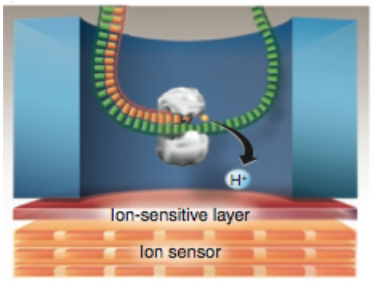


Figure 1.4: The Ion Torrent sequencing platform uses a semiconductor-based high-density array of microwell reaction chambers positioned above an *ion-sensitive layer* and an *ion-sensor*.

However, this technology is still a wash-and-scan system like all the available second generation sequencing technologies. Despite this, Ion Torrent is able to produce 200 bp reads in less than 2 hours with an instrument of the size of a typical microwave [122].

Recently, Ion Torrent sequencer demonstrated its speed and simplicity during an outbreak of pathogenic *E. coli* in Europe (in particular in Germany). In this occasion the Ion Torrent Instrument was used to sequence the bacteria in few days rather than in weeks [116].

Pacific Bioscience

Pacific Bioscience developed and commercialized a single-molecule real time sequencing approach able to directly observe a single molecule of DNA polymerase as it synthesizes a strand of DNA,

directly leveraging the speed and processivity of this enzyme [38].

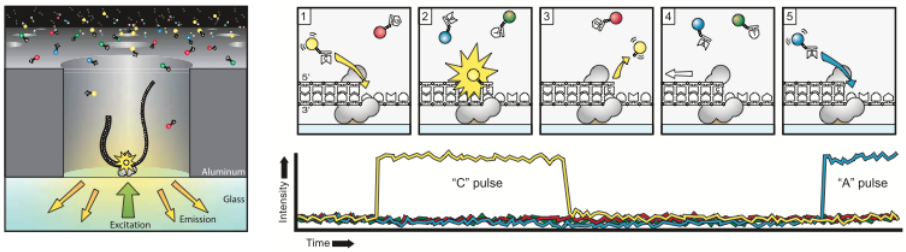


Figure 1.5: PacBio Single Molecule System. Principle of single-molecule, real-time DNA sequencing. A single molecule of DNA template-bound *Omega*29 DNA polymerase is immobilized at the bottom of a ZMW, which is illuminated from below by laser light.

Direct observation of DNA is greatly difficult as a consequence of the fact that a single DNA polymerase molecule is of the order of 10 nm in diameter. In order to observe DNA polymerase in real time (*i.e.* detecting the incorporation of a single nucleotide taken from a large pool of potential nucleotides during DNA synthesis) the *zero-mode waveguide* (ZMW) technology presented in [38] was used.

Pacific Bioscience reads length can reach 1 Kbp and an instrument run takes 15 minutes. However the error rate (substitutions, insertions, and deletions events) are quite frequent (around 15% . In order to prevent that, a feature of Pacific Bioscience instrument is the *redundant resequencing*. Redundant resequencing generates multiple independent reads of each template molecule, and then combine the information in a consensus, reaching in this way an accuracy exceeding 99.9% [122]. The other acclaimed feature proposed by Pacific Bioscience is the *strobe sequencing*. The aim of strobe sequencing is to obtain data that can be used in *de novo* assembly or in large variant detection: the Pacific Bioscience read length is essentially limited by the continuous and damaging illumination required to "read" the DNA. Strobe sequencing addresses this issue by periodically "turning off" the light (*i.e.*, the laser). While the laser is off, no sequence data is produced, but the reaction can continue without damaging the DNA. When the light is on again, another continuous stretch can be read. The amount of DNA that is not read can be inferred knowing the fragment speed. This procedure can continue until 1 Kbp of data is read but a much longer sequence of DNA is processed. In this way it is possible to obtain a set of sequences at a know distance and orientation.

Oxford Nanopore

Oxford Nanopore sequencing technology (together with other nanopore based technologies) relies on transit of a DNA molecule or its component bases (*i.e.*, nucleotides) through a hole and detecting the bases by their effect on an electric signal.

In particular, Oxfod Nanopore is commercializing a sequencing system based on three natural biological molecules [29, 59]. This technology employs an exonuclease/based "sequencing-by-deconstruction" [122]. The individual nucleotides are cleaved from the DNA strand: as each cleaved base traverse the nanopore, the current is distributed in a manner characteristic for each base, thus allowing the instrument to read the bases.

Like all single molecule based sequencers, also the Oxford Nanopore sequencer is characterized by long read lengths, small instrument size (due to the absence of optical cameras or lasers) and short sequencing times (due to the absence of PCR or labelled nucleotides). However, as consequence of the fact that the template molecule is digested during the sequencing (*i.e.*, single nucleotides are cleaved from the DNA strand), redundant sequencing is not possible.

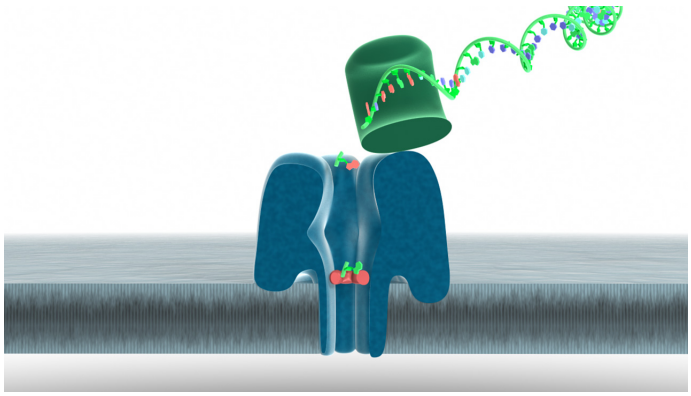


Figure 1.6: Oxoford Nanopore system. Using a processive enzyme to cleave individual nucleotides from a DNA strand and pass them through a protein nanopore.

I

The Alignment Problem

2

Short String Alignment Problem

One of the main applications of string matching is computational biology. A DNA sequence can be seen as a string over the alphabet $\Sigma = \{A, C, G, T\}$. Given a reference genome sequence, we are interested in searching (*aligning*) different sequences (*reads*) of various lengths. When aligning such reads against another DNA sequence, we must consider both errors due to the sequencer and intrinsic errors due to the variability between individuals of the same species. For these reasons, all the programs aligning reads against a reference sequence must deal (at least) with mismatches.

As a general rule, tools used to align Sanger reads (*i.e.* among the others BLAST [7]) are not suitable (*i.e.* are not efficient enough) to align next generation sequencers output due, essentially, to the sheer amount of data to handle. Therefore, aiming at keeping the pace with data production, new algorithms and data structures have been proposed in the last years.

In this chapter we are interested in describing state of the art algorithms and tools able to align the large amount of reads produced by Second Generation Sequencers. The race among Illumina, Solid, and ABI to produce always more reads has been coupled by a race among the Computer Science community to produce software able to analyse NGS data in a feasible amount of time. For this purpose old algorithms and data structures have been revised, while, more often, new algorithms and data structures have been designed.

The final picture is composed by a large variety of new and interesting approaches from which a small number of solutions have emerged. However, the fast pace at which technology is evolving suggests that all the solutions proposed until now (also the most successful ones) have to evolve and to adapt themselves. Moreover, the fast evolving environment makes possible that new algorithms and approaches could emerge as optimal solutions in the future.

A particular interesting observation is that often, theoretically slower algorithms and data structures are used in place of optimal ones. This scenario is a consequence of several practical considerations. As an example consider that complexity analysis discard constant factors that are fundamental in practical scenarios. Moreover, as we will see, some data structures are able to take advantage of real architectures (*e.g.* memory locality). The differences between theoretical optimal results and practical optimal results show how complexity analysis can fail in faithfully describe algorithm performances especially in real scenarios.

The Chapter is divided in the following parts: in Section 2.1 we will introduce some basics definitions that we will use throughout this Chapter and also in the next one. Section 2.2 wants to give an historical perspective of the string alignment problem. In Section 2.3 we will concentrate our attention on alignment algorithms and data structures for Next Generation Sequencing technologies, in particular the Section is divided into three main parts: Section 2.3.1 describes hash based aligners, Section 2.3.2 describes prefix based aligners while Section 2.3.3 describe aligners designed for distributed environments. Finally, in Section 2.4 we will draw some conclusion.

2.1 Definitions and Preliminaries

In order to ease the reading and comprehensibility of this Chapter (and also of the next one) we start presenting some preliminary concepts and some definitions that will turn out useful in what follows.

We will always work with strings over a finite alphabet Σ with $|\Sigma| = b$. In most of the cases we will work with $\Sigma_{DNA} = \{A, C, G, T\}$ (the DNA alphabet), however all definitions and algorithms presented therein are valid for all finite alphabets such that $b \geq 2$.

In the *Alignment Problem* we are given a pattern P and a string T and we are interested in finding all positions in T where an occurrence of P is present. The length of P is represented by m ($|P| = m$), while the length of T is represented by n ($|T| = n$). As a matter of fact $n \geq m$ and as we will see, usually a more interesting hypothesis is that $n >> m$. The symbol that occurs at position i of a string S is identified by $S[i]$, while $S[i..j] = S_{ij}$ are two alternative ways to identify the *subsequence* of S that starts in position i and ends in position j.

Definition 1 (Alignment Problem) *Given a text T and a pattern P all over the alphabet Σ, a real number $k \in \mathbb{R}$, and a* distance function *$d : \Sigma^* \times \Sigma^* \rightarrow \mathbb{R}$ the Alignment Problem is the problem of identifying the set of all text positions i such that there exist j such that $d(P, T[i, j]) \leq k$.*

If $k = 0$ we have the special case of *exact string matching* problem, while when $k > 0$ we have the more general *inexact/approximate string matching* problem. In this last case the two most widely used distance metrics are the *Hamming distance* [53] and the *edit distance* (or Levenshtein distance) [91].

The Hamming distance between two strings R and S such that $|R| = |S|$ is the number of mismatches between R and S.

Definition 2 (Hamming Distance) *Let us define* $\mathrm{neq}(c, d)$ *as a function that returns* 1 *if $c \neq d$, and* 0 *otherwise. The Hamming distance between R and S with $|R| = |S|$ is:*

$$d_H(R, S) =_{\mathrm{def}} \sum_{i=0}^{n-1} \mathrm{neq}(R[i], [i])$$

The Levenshtein distance between two strings R and S is the minimal number of edit operations (substitutions, insertions, and deletions) to transform one string into the other. The alignment between two strings at Levenshtein distance k can be represented as the sequence of edit operations $E = e_1, e_2, ..., e_t$ necessary to transform R into T or vice-versa.

Definition 3 (Levenshtein (Edit) Distance) *Given a function $w(e)$ that associate a real number to every possible edit operation (substitutions, insertions, and deletions), and the sequence E of edit operations to transform R into S, then the Levenshtain or Edit distance between R and S is:*

$$d_L(E) =_{\mathrm{def}} \sum_{i=0}^{|E|} w(e_i)$$

In many applications one is given a text T and a pattern P and is required to find the *best* occurrences of P at distance at most k (usually Hamming or Levenshtain distance). This problem is defined as the *best k-alignment problem*.

Definition 4 (Best k-Alignment Problem) *Given a text T and a pattern P all over the alphabet Σ, a real number $k \in \mathbb{R}$, and a* distance function *$d : \Sigma^* \times \Sigma^* \rightarrow \mathbb{R}$ the Best k-Alignment Problem is the problem of identifying the set of all text positions i such that there exist a j that $d(P, T[i, j]) \leq k$ and such that there are no other indexes i', j' in the text such that $d(P, T[i', j']) < d(P, T[i, j])$.*

2.2 String Matching History

String matching can be divided into two main areas: exact string matching and inexact string matching. When doing approximate string matching the most used distance metrics are the Hamming and the Levenshtain distances.

The first algorithms to solve the exact string matching problem are due to Knuth, Morris and Pratt [80], Boyer and Moore [19], running in time $\mathcal{O}(n+m)$ (n text and m pattern lengths, respectively), and Rabin and Karp [75], requiring time $\mathcal{O}(n+m)$ on average.

When a large number of patterns must be searched, these solutions do not perform well, as the text has to be scanned for each pattern. In such cases, it is convenient to build an index over the text, allowing to search a pattern in time proportional to its length, or over the patterns, in which case the reference is scanned only once. Usually, when the reference is fixed, or the total pattern length is larger than the reference, indexing the text is the preferred solution. For a detailed discussion on when, in biological applications, an index over the text is preferred to one over the patterns, and vice-versa, refer to [57]. The most popular among such indexes are Suffix-Trees [184, 10, 173] and Suffix-Arrays [109].

Approximate string matching at distance k under the edit metric is called the *k-difference problem*, while under the Hamming metric, it is called the *k-mismatch problem*. A simple algorithm for the k-difference problem is based on dynamic programming and it has a running time $\mathcal{O}(nm)$. Several efforts were made to improve this result. Abrahamson [4] shows that string matching with mismatches can be solved in time $\mathcal{O}(n\sqrt{m \log m})$. The fastest solutions for the k-mismatch problem relies heavily on the ability to search the Suffix-Tree of the text and of the pattern. Landau and Vishkin [83, 84] introduced a method running in time $\mathcal{O}(nk)$ that uses constant time lowest common ancestor queries on the Suffix-Trees of P and T (which is now known as "kangaroo hopping"). The algorithm of Galil and Giancarlo [43] attains the same complexity $\mathcal{O}(nk)$. A more recent paper [157] proposed a variation of FAAST [105] that has average running time $\mathcal{O}(n(\log m + k)/m)$ that was proved to be optimal for approximate string matching [28]. The asymptotic running time was improved in [8] to $\mathcal{O}(n\sqrt{k \log k})$, by a method based on counting and filtering, the Suffix-Tree with kangaroo hooping, and fast Fourier transforms, which may ultimately lead to a more sophisticated implementation.

The first algorithm that solved the k-mismatch problem with the construction of an index is due to Ukkonen and Jokinen [72]. The first solution with query time depending only on k and m was proposed by Ukkonen [173] using Suffix-Trees. More recently [63], the k-difference problem has been solved in time $\mathcal{O}(|\Sigma|^k m^k \max(k, \log n))$ where Σ is the alphabet, using compressed Suffix-Arrays [51].

2.3 Aligners and Alignment Techniques

The recent spread of Next (Second) Generation Sequencing (NGS) technologies caused the need to design new tools able to cope with the huge amount of (short) sequences produced. Tools designed for long Sanger-like sequences were not able to operate on such amounts of data in a feasible amount of time. Tools able to align reads against a reference sequence are usually dubbed *aligners*. Solutions specifically designed for Next Generation Sequencing data are called NGS-aligners.

Driven by the short read lengths and by specifics technical aspects (Illumina and Solid reads are not seriously affected by duplication errors) a large amount of NGS-aligners focused on the *best k-mismatch problem* (*i.e.* finding the *best* occurrence of the pattern, with at most k mismatches). As a consequence of the increasing length of the reads and of the necessity to align reads against biologically distance reference most of the NGS-aligners are now able to deal with small insertions/deletions events (*indels*).

The vast majority of aligners build an index over the text. However, some tools are available that index the reads. According to [93] we can cluster existing alignment algorithms into two main classes: algorithms based on *hash tables* and algorithms based on *suffix-based* data structures. Moreover, in our discussion we will add a third category, composed by distributed algorithms.

On the one hand, hash-based aligners build a dictionary of the reference and then use this dictionary to search the query sequences. On the other one, suffix-based methods rely on the construction of an Prefix/Suffix Trie structure (*i.e.* an ordered tree data structure) over the reference. Aligners normally follow a multistep procedure to accurately map sequences. During a

first *filtering* phase, heuristics techniques are used to quickly identify a small set of positions in the reference sequence where a read's best alignment is most likely to occur. Once this small subset of locations is computed, more accurate and often slower alignment procedures are used.

Usually a small portion of the read is searched through the data-structure (hash as well as Prefix/Suffix-Trie) in order to isolate regions candidate to be aligned. Only on these regions a more accurate alignment is performed. This procedure is usually named *seed-and-extend*. The seed (which usually corresponds to the first part of the read) is searched allowing only a few number of mismatches (some times the seed can be searched in exact way).

One of the first and extremely successful hash based aligner is BLAST [7]. BLAST, basically, searches in the reference perfect matches of the query of length 11. Once these exact seeds are identified, the search is refined by a Smith-Waterman-like alignment [168, 48]. Hash-based aligners for NGS refined and improved this basic seed-and-extend schema. Basic seeding has been substituted by *spaced seeds* (SOAP [98], MAQ [95] and ZOOM [102] to mention a few) and *q-grams* (SHRiMP [156, 32], and RazerS [183] among the others). Moreover, several improvements over the standard seed-and-extend schema concern the extend phase: in [156] a *vectorized* version of Smith-Waterman algorithm is used to gain speed from the SSE2 CPU instructions implemented in latest x86 CPUs.

Suffix-based aligners implements one of the many available indexes like Suffix-Trees [10], Suffix-Arrays [109] and FM-indexes [40]. In the NGS context, FM-indexes are the most widely used thanks to their principal characteristic: (theoretically optimal) compressibility. Several of the short read alignment programs (BOWTIE [89], BWA [93], and SOAP2 [99] among the others) are based on the Barrows-Wheeler transformation [21]. These methods usually use the FM-indexes that allow the efficient construction of a Suffix-Array with the further advantage that can be compressed. The FM-index retains the Suffix-Array's potential for rapid searches with the great advantage that the index often is smaller than the text. Suffix-based aligners use heuristic similar to the one implemented in the hash-based: the index is used to search for (almost) exact matches that will be used as anchor for further extensions.

All the aligners are able to fruitfully exploit multiple cores architectures. However, it is worth to mention that there is a particular class of aligners that could use many machines in parallel. These *distributed* aligners can run over clusters or clouds of computers using frameworks for distributed computation like MPI [50] and MapReduce [34].

2.3.1 Hash-Based Aligners

Hash-based aligners pre-process the reference text and/or the query reads to obtain a dictionary that allows to search a read r in (expected) time proportional to $O(|r|)$. Basic hash-based aligners simply search for exact occurrences of seeds inside the text. As showed in [23] and later in [107], seeding with non-consecutive matches improves sensitivity. A *spaced seed* is a seed of length l where matches are required in only k fixed positions. A spaced seed is usually represented by a $\{0,1\}$-string typically called *template*. The total number of 1's is usually named seed's *weight*. For example, the template '111010010100110111' requiring 11 matches at the '1' positions is 55% more sensitive than BLAST when aligning two sequences allowing 70% of similarity, which by default uses a seed with 11 consecutive matches.

Seeds are usually searched allowing a small number of mismatches. If the read r must be aligned with at most k mismatches against the reference sequence, then there is at least one (consecutive) substring of length $\left\lfloor \frac{|r|}{k+1} \right\rfloor$ that occurs without errors (this is a simple consequence of the *pigeon hole principle*). With this clue in mind one can build an hash table of all l-mers with $l = \left\lfloor \frac{|r|}{k+1} \right\rfloor$ and use a seed-and-extend strategy. RMAP [167] uses this simple strategy. The main drawback of this technique is that, for practical values of $|r|$ and k, the seed length is so small that too many false positive are produced in the first alignment step causing a long and inefficient extension phase. As a consequence, the vast majority of seed and spaced-seed approaches allow a small number of mismatches even in the seed. Moreover, the seed size is usually decided during the pre-processing stage. Allowing k mismatches in a seed means that $\binom{2k}{k}$ different templates are required (all the

layouts that allow at most k mismatches). This number is exponential in k an therefore the method can became quickly inefficient. In [102], the ZOOM aligner uses different spaced seeds at several designated reads' positions to find all possible occurrences of a read without losing occurrences. In particular, ZOOM aligner, given the read length m and the maximum allowed Hamming distance k, tries to design the minimum number of spaced seeds of weight w to achieve full sensitivity. Other aligners (like SOAP and MAQ) use as seed the fist l bases of the read (the most reliable part of Illumina data sequences) allowing a limited number (usually two) of mismatches. In order to align a 32 bp read, RMAP uses three templates of weight 10, MAQ requires six templates of weight 26, while ZOOM requires five seeds of length 14.

Seed and spaced-seed techniques do not allow indels within the seed. Seed-based algorithms usually postpone the indel search to the extension phase, aligning the remaining part of the read with a Smith-Waterman-like algorithm. SHRiMP [156], its successor [32], and RazerS [183] build an hash table that embeds the indels thanks to *q-grams*. The q-gram concept (string of length q) was introduced in [147]: this method generalizes the basic principle of the seed method applying again the pigeon hole principle. The key observation is that if a read r of length $|r|$ occurs in the reference text with at most k differences (both mismatches and indels), then at least $(|r| + 1 - (k + 1)q)$ of the q-grams in r occur in a window of size at most $|r|$ in the text. The main difference between seed and q-gram based methods is the fact that the former search one long template while the latter search for multiple short seeds in a restricted region. Anyway, both strategies are based on fast look-up tables.

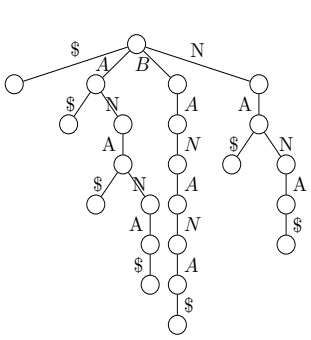
The idea to use multiple seeds, without employing q-grams, is used in SSAHA2 [132]: this technique is used to speed-up alignments of relatively long reads (e.g. 454 reads) where it is reasonable to require that two or more seeds fall in a small window of the reference.

2.3.2 Prefix/Suffix-Based Aligners

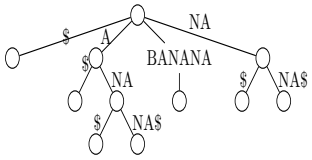
Algorithms that fall in this category are based on clever representation of a common data structure: Prefix/Suffix Trie. These representations (Suffix-Trees [184, 174], Directed Acyclic Word Graphs or DAWG [15], Compressed DAWG [16, 68], Suffix-Arrays [109, 74], enhanced Suffix-Arrays [3] and, FM-index [40]) have the advantage over hash-based algorithms that during the alignment phase identical copies of a substring in the reference need to be scanned only once. For example, in a Suffix-Tree, identical sub-strings collapse on a single path, while in a Suffix-Array they are stored in contiguous entries.

A *Suffix-Trie*, or simply a *Trie*, is a data structure that stores all the suffixes of a string. In particular a Trie for a text T could be constructed by simply inserting in a empty keyword-tree [52] all its suffixes. A Trie for a string T of length n over the alphabet Σ is a rooted tree with branching factor at most $|\Sigma|$ and with exactly n leaves numbered from 1 to n. In a Suffix-Trie each internal node represents a unique substring of T. For this reason each edge is labelled with a non empty character of T. Two edges leaving the same node cannot be labelled with the same character. For any leaf i, the concatenation of the edge labels on the path from the root to leaf i exactly spells the suffix of x starting at position i. A problem arises if a suffix j of T occurs also as a prefix of another suffix i. To avoid such a situation the special character $\$ \notin \Sigma$ is usually added at the end of T. In order to boost the search, Suffix-Tries are equipped with the so called *suffix-links*: given a suffix y, each node representing the string ay (with $a \in \Sigma$) has a suffix-link that leads to the node representing the suffix y. This way, the time needed to determine if a query r has an exact occurrence using a Trie is $O(|r|)$. The drawback of a Trie-like data structure is the fact that a string T needs space equal to $O(|T|^2)$ making this data structure useless for the vast majority of applications in bioinformatics.

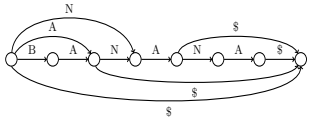
A *Suffix-Tree* is a more space efficient representation of a Trie. In order to preserve space, unary paths (*i.e.* stretches of nodes with out-degree one) are compressed into a single node. Only the first character of the unary path needs to be stored. It can be shown that such a tree has only $O(|T|)$ internal nodes. McGreight first [113] and Ukkonen [174] later showed that this structure can be constructed in time $O(|T|)$. Later improvements [121] showed the possibility to represent a Suffix-Tree in space proportional to $|T| \log_2(|T|) + O(|T|)$ bits. However, despite this, most



(a) Suffix Trie



(b) Suffix-Tree



(c) DAWG

	B	A	N	A	N	A	$
	0	1	2	3	4	5	6
0	6	$	B	A	N	A	N	A
1	2	A	$	B	A	N	A	N
2	3	A	N	A	$	B	A	N
3	1	A	N	A	N	A	$	B
4	0	B	A	N	A	N	A	$
5	4	N	A	$	B	A	N	A
6	2	N	A	N	A	$	B	A
		↓						↓
		SA						L

(d) SA and BWT

Figure 2.1: Suffix-based structure for the string "BANANA$": Figure 2.1(a) shows the Suffix-Trie, Figure 2.1(b) shows the Suffix-Tree, Figure 2.1(c) shows the DAWG generated through compression of the Suffix-Trie (compressing in the same way the Suffix-Tree would have generated a CDAWG), while Figure 2.1(d) shows the Suffix-Array and the BWT transformation.

space-efficient implementations of bioinformatics tools require 12-17 bytes per nucleotide, making Suffix-Tree impractical for indexing large genomes (e.g. the Human one).

Directed Acyclic Word Graphs (DAWG) and their compressed version (CDAWG) are deterministic automata able to recognize all the substrings of a string T. Similarly to what happens in Suffix-Trees, a node in a DAWG represents a substring of the text, but in this case each node is augmented with *failure* links (*i.e.* information to deal with the paths that are not present in the text). Leaves do not need to be distinguishable, therefore less space is necessary retaining, however, all the Suffix-Tree's abilities. Also with this reduction, known implementations are, again, not able to scale on large genomes.

The common disadvantages of previous solutions are the amount of memory required to actually represent the Trie-like data structures, and the lack of memory locality of graphs in general. Suffix-Arrays have been proposed with the ambition to solve both problems. A Suffix-Array for a string T is basically a sorted list of all the suffixes of T. As for Suffix-Trees, also in this case is useful to append a \$ to T. A Suffix-Array can be built in time $O(|T|)$ and can be used to search a pattern r in time $O(|r| \log |T|)$ using a simple dichotomic search. In its basic representation such a data structures requires 4 bytes per character (*i.e.* 4 bytes to represent a 32 bit pointer to the text). This advantage in space is limited by the presence of a $\log |T|$ factor in the search time. The Suffix-Array, however, can be coupled with the two other arrays containing the information about the *longest common prefix (lcp)* that is an implicit representation of the suffix-links. In this way a query time of $O(|r| + \log |T|)$ can be achieved without compromising the space efficiency. In [3] a clever representation of Suffix-Array and of lcp information is presented that allows a query time of $O(|r|)$ and uses 6.25 bytes per nucleotide.

It must be noticed that in practical situations the basic Suffix-Array version is faster than the most advanced Suffix-Tree implementation. This result is contradicted by complexity analysis that instead suggest that Suffix-Trees attain a speed that is a $\log n$ factor faster than Suffix-Arrays. This

situation is a consequence of two main subtle reasons: first (but less important) the memory locality of Suffix-Array allows to overcame the complexity gap, second, the $\log n$ factor of Suffix-Array is limited, in practical cases, to a small number (in the human genome case $\log |G_{human}| =\sim 22$). The classical complexity analysis do not consider constants factors, that in this case are of primary importance.

Ferragina and Manzini in [40] obtained a major improvement on memory with their FM-index, a data structure based on Burrows-Wheeler Transform (BWT) [21]. The BWT takes as input a text T and returns a reversible permutation of the text characters which gives a new string that is "easy to compress". In order to build the transformation for a text T a '$' is appended at the end of T (for the same reason already seen with Suffix-Trees and Suffix-Array), then a "conceptual" matrix $\mathcal{M}$ is created, whose rows are the lexicographically ordered cyclic shifts of T\$ (see Figure 2.1(d)). The *transformed* string L returned is the last column of $\mathcal{M}$. Ferragina and Manzini showed that there is a strong connection between the Suffix-Array of T and the string L. In particular, given the Suffix-Array S of T, then $L[i] =$ \$ if $S[i] = 0$, otherwise $L[i] = T[S[i] - 1]$. Hon in [56] showed how it is possible to build the BWT of the human genome using no more than 1 GB RAM.

The BWT of T coupled with some other array can be used to efficiently query a pattern P in T using the *backward search* algorithm [40] in time $O(|P|)$. The backward search algorithm is based on the observation that a pattern P that occurs in T induces an interval in the Suffix-Array S.

In order to quickly compute this interval, two functions need to be pre-computed: $C(a)$, returning the number of symbols in T that are lexicographically smaller than $a \in \Sigma$, and $O(a, i)$, returning the number of occurrences of a in $L[0, i]$.

$\underline{R}(P)$ and $\overline{R}(P)$ return the minimum and maximum indexes, respectively, of the Suffix-Array S storing indexes corresponding to suffixes whose prefix is P. More formally:

- $\underline{R}(P) = min\{k : P \text{ is a prefix of } T[S[k]..|T| - 1]\}$
- $\overline{R}(P) = max\{k : P \text{ is a prefix of } T[S[k]..|T| - 1]\}$

Given a pattern P, the interval $[\underline{R}(P), \overline{R}(P)]$ is said Suffix-Array interval and it stores all the occurrences of P into T. This information tells us all the positions at which the pattern occurs within the text T.

In [40] it is proven that if P occurs in T as a substring then, for each character a,

- $\underline{R}(aP) = C(a) + O(a, O(\underline{R}(P) - 1) - 1) + 1$
- $\overline{R}(aP) = C(a) + O(a, O(\overline{R}(P)))$

and that $\underline{R}(aP) \leq \overline{R}(aP)$ if and only if aP is a substring of T. In this way, by iteratively computing $\underline{R}$ and $\overline{R}$ from the last character of P to the first, is possible to find all the occurrences of P in T in time $O(|P|)$.

Ferragina and Manzini showed how it is possible to compress L without significantly increasing the search time. Without compression an FM-index requires 2 bytes per character, but more sophisticated implementation require only 0.5 bytes per character, allowing to store the entire human genome in 2 GB.

All the described data structures are well-suited for exact string matching but do not scale well when (many) mismatches are allowed.Therefore, in real-life applications, a seed-and-extend technique is often used. Usually the first l bases of the read are searched throughout the prefix-like index in order to identify the positions that have to be extended.

FM-index and BW-transformation are the main data structure used by the most successful NGS-aligners (*i.e.* the most used tools). Among the most popular algorithms that use these data structures we should mention BWA [92, 93], BOWTIE [89], and SOAP2 [99].

2.3.3 Distributed Architectures

All "practically-oriented" aligners allow parallel execution to align the huge amount of data produced by NGS. This feature, coupled with the sophisticated algorithms presented above, makes

Name	Algorithm	indels	Author	Year
AGILE	Hash	YES	Misra S. *et al.*	2010
BWA-SW	FM	YES	Li H. and Durbin R.	2010
LASTZ	Hash	YES	unp	2010
BFAST	SA/Hash	YES	Homer N. *et al.*	2009
BOAT	Hash reads	YES	Zhao S. *et al.*	2009
CLC-bio	unknown	YES	commercial	
GASST	Hash	YES	Rizk G. and Lavenier	2010
SSAHA2	Hash	YES	Ning Z *et al.*	2001
ZOOM	Hash reads	YES	Lin H. et al	2008
BOWTIE	FM-index	NO	Langmead B. *et al.*	2009
BRAT	Hash	NO	Harris E. *et al.*	2010
BWA	FM-index	YES	Li H. and Durbin R.	2009
ELAND	Hash	NO	commercial	
Galign	Hash	YES	Shaham S.	2009
GEM	FM-index	YES	unpublished	
GenomeMapper	Hash	YES	Schneeberger K. *et al.*	2009
GSNAP	Hash	YES	Wu T. and Nacu S.	2010
KARMA		YES	unpublished	
MAQ	Hash reads	NO	Li H. *et al.*	2008
MOM	Hash	NO	Dohm J. *et al.*	2008
MrFAST	Hash	YES	Alkan C. *et al.*	2009
MrsFAST	Hash	YES	Hach F. *et al.*	2010
PASS	Hash	YES	Campagna D. *et al.*	2009
PatMaN	Index reads	YES	Prüfer K. *et al.*	2008
PerM	Hash	NO	Chen Y. *et al.*	2009
RazerS	Hash	YES	Weese D.	2009
RMAP	Hash reads	NO	Smith A. *et al.*	2008
rNA	Hash	YES	Policriti A. *et al.*	2008
segemehl	SA	YES	unpublished	
SeqMap	Hash	NO	Hui J. and Wong W.	2008
SHRiMP2	Hash	YES	Matei D. *et al.*	2011
SOAP	Hash	NO	Li R. *et al.*	2008
SOAP2	FM-index	YES	Li R. *et al.*	2009
CloudBurst	Hash/MapReduce	YES	Schatz M.	2009
CrossBow	FM-ind/MapReduce	NO	Langmead B. *et al.*	2009
GNUMAP	Hash/MPI	YES	Clement N.	2010
Myrialign	GPU based		unpublished	
mrNA	Hash/MPI	YES	Del Frabbro C. *et al.*	2011
NovoAlign	Hash/MPI	YES	Krawitz P. *et al.*	2010
pBWA	FM-ind/MPI	YES	unpublished	

Table 2.1: A (surely incomplete) list of available NGS aligners. For each aligner we specified the type of algorithm employed, the possibility to align with indels, authors and publication year. In red we listed aligners designed for "long" 454 reads, in green aligners able to align both long 454 reads and short Illumina/Solid reads, in brown and in blue aligners specifically designed for Illumina reads. Brown and Blue colors distinguish respectively multi-threaded and distributed solutions.

possible to align reads at a very high rate. Nevertheless, the increasing data production is moving several groups toward the implementation of distributed aligners (road already successfully followed by *de novo* assemblers [166]). Despite being a topic worth mentioning, these solutions

do not propose new algorithms and data structures but simply aim at increasing the alignment throughput. Therefore we have decided to give just a brief description of some of the available solutions.

Two commonly used frameworks for distributed computation are MPI [50] and MapReduce [34]. The former is an API (*Application Programming Interface*) specification that allows processes to communicate. The different processes can be on the same machines or on different machines (usually called *nodes*) connected through a communication channel (*e.g.*, Ethernet and Infiniband). MapReduce is a framework that allows and simplifies distribution of independent, and hence parallelizable, operations. It mainly operates in two steps: during the Map step a master node partitions the input into smaller subsets and distributes those to worker nodes that, after processing data, return the results to the master node. In the Reduce step the master collects the results and combines them (for a review see [162]).

pBWA [135] is a parallel implementation of the popular software BWA [92]. It was developed by modifying BWA source code with the OpenMPI C library. mrNA (unpublished) is the MPI version of rNA aligner [144]. mrNA is different from the vast majority of distributed aligners for the fact that not only the alignment phase is distributed, but also the indexing happens on different nodes.

Crossbow [88] is Hadopp-based [9] (Hadoop is the open source MapReduce's implementation) aligner and SNP-caller. Crossbow [88] uses Bowtie [89] to align reads and SOAPsnp [97] to find SNPs. Myrna [87] is designed for transcriptome differential expression analysis. Like Crossbow it uses the Hadoop interface. CloudBurst [161] is a parallel read-mapping algorithm and is able to obtain the same results of RMAP [167] but, thanks to Hadoop framework, it is able to align large data sets within a reasonable amount of time.

2.4 Conclusions

In this Chapter we described and analysed algorithms and data structures used to perform string alignment. In particular our attention has been focused on NGS-aligners and short sequence alignment. Recent years have seen huge efforts in the production of aligners as Table 2.1 wants to witness. However, despite the technicalities and the different heuristics proposed tools used a limited amount of algorithms. In particular, most of available NGS-aligners make extensive use of heuristics to speed up the search, at the cost of a lower precision. This is a consequence of the need to align in a short time the large amount of data produced by NGS-machines. Recently, a new generation of *distributed* aligners appeared, trying to use the network to overcame speed limits.

It is remarkable the fact that often not theoretically optimal solutions (like Hash Tables and Suffix-Arrays) perform better, at least in practice, than theoretically guarantee optimal solutions (like Suffix-Trees). This fact is recurrent in bioinformatics (Chapter 4 will provided another example), and it is a consequence of the fact that often complexity analysis do not take into account constants factors, that, in practice, are of primary importance in real scenarios.

3

rNA: Birth and Growth of a NGS Tool

String alignment is an easy (computationally) and well studied problem, however, it is at the center of increasing interest by the bioinformatics community. The huge amount of data produced by NGS technologies, and the promises to produce even more data in the next feature, boosted the need to design and implement tools able to align hundreds of millions of sequences as fast as possible.

In the NGS context, optimal solutions can became obsolete every time a sequencer is improved, that means, that every few months the best available software can became unusable. At the present time many solutions are under development, both based on algorithms/heuristics improvements and on distributed architectures.

The aim of this chapter is to present and analyse one of the main contribution of this thesis: *rNA*. The *randomized Numerical Aligner* has been designed for the Institute of Applied Genomics (IGA) in Udine but is now being used and tested by several other research groups around the globe. rNA is an *hash based* aligner designed, but not limited, to align Illumina reads.

We will describe rNA showing how this tools is born (the original idea) and how it evolved until now. Section 3.1 will explain the reasons that guided us to design and to implement a new aligner despite the many already available solutions (see Table 2.1). In Section 3.2 we will describe the Rabin and Karp *exact* matching algorithm and, in Section 3.3, we will subsequently see how this algorithm can be *extended* to deal also with *mismatches*. Section 3.4 will describe how the extended Rabin and Karp algorithm can be used as a core for a *short string aligner*. The large amount of data produced by NGS machines, and the need of fast but reliable analysis suggest us to implement a distributed version, dubbed mrNA, that will be presented in Section 3.5. Section 3.6 will show how rNA is able to compete with state of the art available solutions: we tested our tool against other well known and used aligners on simulated datasets (Section 3.6.2), on real datasets (Section 3.6.3) and in a distributed environment (Section 3.6.4). Finally, Section 3.7 will discuss the future works and extensions already planned on rNA.

3.1 Why a New (Short) Read Aligner?

Table 2.1 of Chapter 2 shows a long list of aligners probably far from be complete. It must be stressed that only a limited number of such tools are in practice used by the bioinformatics community. Many solutions have been designed for a particular purpose often no more useful (*i.e.* aligning reads of length 36 bp), many others have been published, but then have not been improved in order to keep the pace with the sequencers throughput (for example the MAQ aligner). Other solutions, despite being algorithmically interesting, do not output alignments in a standard format (like the well known SAM format) and hence are of no practical value.

Among the most popular and most used aligners we can identify BWA [93], BOWTIE [89], and SOAP2 [99]. These three software are used by a large community around the world, they are fast, they need a small amount of computational resources, often are coupled or integrated with

other software able to perform subsequent analysis like SNPs identification (Single Nucleotide Polymorphisms) and CNV analysis (Copy Number Variation). All these software require the same input format (*fastq* format), and (even if with some differences) output the alignments in the same standard format (SAM/BAM format).

The great number of already available solutions (see Table 2.1) may suggest us a couple of questions: "Why we need another aligner?" and "Why somebody has to spend time and resources to design, implement, test and distribute a tool to solve a problem that has already so many solutions?"

As we will see there are many stimulating reasons to design, implement, test, and distribute a new short string aligner:

- sequencing technology is improving at a speed higher than the fastest software development pipeline. In order to release to a wide public (global) a new version of a tool, this must be analysed and tested, even if a simple modification took place. As a consequence of this, tools available to download are not the latest available version and therefore a lab may end up in the annoying situation to *wait* for the next software release to proceed with analysis.

- In a constantly improving and experimenting environment specific problems can appear and only ad-hoc solutions can be used to solve them. The chances of successfully modifying a non-in-house tool are close to zero.

- Even though the most used tools comply with standard input and output formats, often they do not implement/respect all specifications. This is the case, for example, of the output format produced by SOAP and BOWTIE that is limited only to the mandatory fields, while often also non mandatory ones are useful.

- Hands-on-experience allows a group to understand the need of new features or heuristics that can be useful to the overall community. An in-house aligner gives the opportunity to implement and test these new ideas.

These motivations, and many others that we will see, guided us towards the implementation of a new short read aligner called rNA: *randomized Numerical Aligner*. rNA [144] uses a core alignment based on an extension of the Rabin and Karp string alignment algorithm [75] that allows the search of inexact occurrences of a pattern in time proportional to pattern and text length. rNA is a complete tool [178] (a NGS-aligner) able to read input files in the most widely used formats and output a complete and valid standard output format (*i.e.*, SAM/BAM).

rNA is used in almost all activities of the Institute of Applied Genomics (IGA) in Udine, where it has been developed in close interaction with Bioinformatics, Biostaticians and Biologists. Moreover, the stable rNA version is available for download at `http://iga-rna.sourceforge.net/` where, at the present time, it has been downloaded 133 times in 18 different countries.

3.2 Rabin and Karp Algorithm: From Strings To Numbers

Without loss of generality we can consider our alphabet composed by b characters/digits $\Sigma = \{0, 1, \ldots, b-1\}$, with $b \geqslant 2$. Let $X = X[0]X[1]\ldots X[n-1]$ and $Y = Y[0]Y[1]\ldots Y[n-1]$ be two strings over the alphabet Σ. The *Hamming distance* $d_H(X, Y)$ between X and Y is defined as the number of mismatches between X and Y (see Chapter 2).

Given numbers $0 < m \leqslant n$ and $0 \leqslant s \leqslant n - m$, we denote by $X_{(s)}$ the string $X_{(s)} =_{\text{def}} X[s]X[s+1]\ldots X[s+m-1]$. We denote the numerical radix-b representation of a string X of length n by $x =_{\text{def}} b^{n-1}X[0] + b^{n-2}X[1] + \cdots + bX[n-2] + X[n-1]$. Given a positive integer q, the number $\hat{x}$ stands for $x \bmod q$, and is called the *fingerprint* of the string X.

Given a text T and a pattern P both over the alphabet Σ, the exact string matching problem consists in finding all the position i in T such that $P = T_i$.

One of the simplest exact string matching algorithms—that also performs well in practice—is the Rabin and Karp randomized algorithm [75]. For every $s \in \{0 \ldots n - m\}$, the algorithm

encodes P and any $T_{(s)}$ by the radix-b numbers p and $t_{(s)}$, respectively, and replaces expensive string comparisons by constant-time suitable numerical comparisons (see Algorithm 1).

Algorithm 1: Rabin and Karp algorithm for exact string matching

Input: $T = T[0]T[1]\dots T[n-1]$, $P = P[0]P[1]\dots P[m-1]$, both over the alphabet $\Sigma = \{0, 1, \dots, b-1\}$ and q a well chosen number

Output: All positions s, where $0 \leqslant s \leqslant n-m$ and $d_H(P, T_{(s)}) = 0$.

```
1  SOLUTIONS ← ∅;
2  p̂ ← t̂ ← 0;
3  for i ← 0 to m − 1 do
4  |  p̂ ← (b · p̂ + P[i]) mod q;
5  |  t̂ ← (b · t̂ + T[i]) mod q;
6  if (p̂ = t̂) then
7  |  if d_H(P, T_(0)) = 0 then
8  |  |  SOLUTIONS ← SOLUTIONS ∪ {0};
9  for s ← 1 to n − m do
10 |  t̂ ← (b · (t̂ − h · T[s − 1]) + T[s + m − 1]) mod q;
11 |  if (p̂ = t̂) then
12 |  |  if d_H(P, T_(s)) = 0 then
13 |  |  |  SOLUTIONS ← SOLUTIONS ∪ {s};
14 return SOLUTIONS;
```

As usually m is larger than the length of a processor word, instead of storing p and $t_{(s)}$, one keeps the values $\hat{p} = p \bmod q$ and $\hat{t}_{(s)} = t_{(s)} \bmod q$. As an indication that P *may* occur with shift s in T , the algorithm now tests whether $\hat{p} = \hat{t}_{(s)}$ and, if so, it proceeds to a character-by-character comparison of P and $T_{(s)}$. Randomly choosing q to be a prime number in the interval $[2, mn^2]$, the test $\hat{p} = \hat{t}_{(s)}$ produces few false positives [75] (*i.e.*, it gives a positive answer in the case when $P \neq T_{(s)}$). Moreover, as $\hat{t}_{(s+1)}$ can be computed from $\hat{t}_{(s)}$ in constant time, the overall expected time complexity is $\mathcal{O}(n+m)$.

3.3 Extending Rabin and Karp Algorithm to Mismatches

Rabin and Karp algorithm cannot handle mismatches. A particular biologically interesting problem is the *k-mismatch problem* (see Chapter 2). Given a text T of length n and a pattern P of length m over the same finite alphabet Σ and an integer k we are interested in all the pairs $\langle s, d_H(P, T_{(s)})\rangle$, where $0 \leqslant s \leqslant n-m$ and $d_H(P, T_{(s)}) \leqslant k$.

The Rabin and Karp method has been employed in [123] to solve the k-mismatch problem. That approach is based on generating all the $\sum_{i=0}^{k} \binom{m}{i}(b-1)^i$ strings obtained from P with at most k mismatches. This method is limited by the exponential blow-up on m, it would be interesting be able to avoid this and design a method based on "verification" rather than on "generation".

The final goal is find a way to retain all the advantageous features of the Rabin and Karp algorithm (encoding strings by a radix-b number and storing values modulo an appropriate number q) adding the possibility to deal with mismatches. In particular we need a fast test such that it outputs true every time $d_H(P, T_{(s)}) \leqslant k$ and that produces *few* false positives (*i.e.* it outputs true but $d_H(P, T_{(s)}) \geq k$).

One can start by noting that when $k = 0$, then $\hat{p} = \hat{t}_{(s)}$ is equivalent to $(\hat{p} - \hat{t}_{(s)}) \bmod q = 0$. Starting from this observation we can define a set $\mathcal{Z}(k, q) \subseteq \{0, \dots, q-1\}$, such that whenever $d_H(P, T_{(s)}) \leqslant k$, then $(\hat{p} - \hat{t}_{(s)}) \bmod q \in \mathcal{Z}(k, q)$ holds. More formally, the set $\mathcal{Z}(k, q)$ is defined as follows.

Definition 5 *Given $m > 0$, $0 < k < m$ and $q > 0$, define $\mathcal{Z}(k, q)$ to be the set*

$$\mathcal{Z}(k,q) =_{\text{def}} \{(x-y) \bmod q \mid X, Y \in \Sigma^m,\ d_H(X,Y) \leqslant k\}.$$

We will sometimes refer to the elements of $\mathcal{Z}(k, q)$ as *witnesses*, as they testify that two strings *can* be at Hamming distance at most k. The algebraic difference between the numerical representations of two strings at a given Hamming distance is characterized in Lemma 1.

Lemma 1 *Given two strings X and Y of the same length m, for any $0 < k < m$ we have $d_H(X, Y) = k$* if and only if

$$\begin{aligned} x - y \in \{&(-1)^{u_1} t_1 b^{i_1} + \cdots + (-1)^{u_k} t_k b^{i_k} : u_1, \ldots, u_k \in \{0,1\}, \\ &t_1, \ldots, t_k \in \{1, \ldots, b-1\},\ i_1 > \cdots > i_k \in \{0, \ldots, m-1\}\}. \end{aligned}$$

Plainly, from Lemma 1, $\mathcal{Z}(k, q)$ can be expressed as

$$\begin{aligned} \mathcal{Z}(k,q) \quad = \quad & \{0\} \cup \{\left((-1)^{u_1} t_1 b^{i_1} + \cdots + (-1)^{u_j} t_j b^{i_j}\right) \bmod q : 0 < j \leqslant k \\ & u_1, \ldots, u_j \in \{0,1\},\ t_1, \ldots, t_j \in \{1, \ldots, b-1\}, \\ & i_1 > \cdots > i_j \in \{0, \ldots, m-1\}\}. \end{aligned}$$

An upper bound for the cardinality of $\mathcal{Z}(k, q)$ is $\min\{q, \sum_{j=0}^{k} \binom{m}{j}(2(b-1))^j\}$, as for each $0 \leqslant j \leqslant k$, there are $\binom{m}{j}$ ways to choose j pairwise distinct $i_1, \ldots, i_j$, and $(2(b-1))^j$ ways to choose $u_1, \ldots, u_j$ and $t_1, \ldots, t_j$.

In order for the test $(\hat{p} - \hat{t}_{(s)}) \bmod q \in \mathcal{Z}(k, q)$ to give few false positives, the size of $\mathcal{Z}(k, q)$ must be *small*, which, working modulo an arbitrary number q, may not be true. The main idea of our approach is to choose $q = b^w - 1$, where $w < m$ is a natural number large enough, according to a few complexity considerations.

Notice that, arithmetic modulo numbers of the form $2^w - 1$ (called Mersenne numbers) is used in various applications, like digital systems based on residue number system, or cryptography, therefore, efficient VLSI circuit architectures for addition and multiplication modulo $2^w - 1$ have been proposed over the years (see, *e.g.*, the discussion in [192], and the references therein). Notice also that, in general, the usage of q of the form $2^w - 1$ is *not* suggested when exact search is performed.

The following lemma shows that the choice $q = b^w - 1$ guarantees that $\mathcal{Z}(k, q)$ has a small cardinality.

Lemma 2 *Given $1 \leqslant w < m$,*

$$\begin{aligned} \mathcal{Z}(k, b^w - 1) \quad = \quad & \{0\} \cup \left\{\left((-1)^{u_1} t_1 b^{i_1} + \cdots + (-1)^{u_j} t_j b^{i_j}\right) \bmod (b^w - 1) : \right. \\ & 0 < j \leqslant k,\ u_1, \ldots, u_j \in \{0,1\},\ t_1, \ldots, t_j \in \{1, \ldots, b-1\}, \\ & \left. i_1 > \cdots > i_j \in \{0, \ldots, w-1\}\right\}. \end{aligned}$$

Proof 1 *To simplify notation in this proof, we let $\mathcal{Z}^*(k, b^w - 1)$ stand for the set on the right-hand side of the equality claimed above. Hence, we have to show that $\mathcal{Z}(k, b^w - 1) = \mathcal{Z}^*(k, b^w - 1)$.*

Since the modulo operation is linear, we have

$$\begin{aligned} & b^s \bmod (b^w - 1) = \\ & \quad = b^{w(s \text{ div } w) + s \bmod w} \bmod (b^w - 1) \\ & \quad = \left((b^w)^{s \text{ div } w}\, b^{s \bmod w}\right) \bmod (b^w - 1) \\ & \quad = \left(((b^w)^{s \text{ div } w} \bmod (b^w - 1))(b^{s \bmod w} \bmod (b^w - 1))\right) \bmod (b^w - 1) \\ & \quad = \left(((b^w \bmod (b^w - 1))^{s \text{ div } w} \bmod (b^w - 1))\, b^{s \bmod w}\right) \bmod (b^w - 1) \\ & \quad = b^{s \bmod w}. \end{aligned}$$

This entails that

$$\begin{aligned}\mathcal{Z}(k, b^w-1) &= \{0\} \cup \big\{\big((-1)^{u_1}t_1 b^{i_1} + \cdots + (-1)^{u_j}t_j b^{i_j}\big) \bmod (b^w-1) : \\ &\quad 0 < j \leqslant k,\ u_1,\ldots,u_j \in \{0,1\},\ t_1,\ldots,t_j \in \{1,\ldots,b-1\}, \\ &\quad i_1,\ldots,i_j \in \{0,\ldots,w-1\}\big\} \\ &=_{\text{def}} R(k).\end{aligned}$$

Clearly, $\mathcal{Z}^*(k, b^w-1) \subseteq R(k)$ *(notice that the difference between* $\mathcal{Z}^*(k, b^w-1)$ *and* $R(k)$ *is that the indices* $i_1,\ldots,i_j$ *are not required to be distinct in* $R(k)$*). To prove the opposite inclusion, we will proceed by induction on* $k < m$*. When* $k = 1$*, the claim is true. Assuming that the claim is true for* $k < m-1$*, we will show that it also holds for* $k+1$*.*

For the sake of clarity, and without loss of generality, we assume onwards that $b = 2$*. For any* $x \in R(k+1) \setminus R(k)$*, where* $x = \big((-1)^{u_1}2^{i_1} + \cdots + (-1)^{u_k}2^{i_k} + (-1)^{u_{k+1}}2^{i_{k+1}}\big) \bmod (2^w-1)$*, we have to show that* $x \in \mathcal{Z}^*(k+1, 2^w-1)$*. We have that* x *can be written as*

$$\Big(\big((-1)^{u_1}2^{i_1} + \cdots + (-1)^{u_k}2^{i_k}\big) \bmod (2^w-1) + \\ +(-1)^{u_{k+1}}2^{i_{k+1}} \bmod (2^w-1)\Big) \bmod (2^w-1).$$

From the inductive hypothesis, the first of the above two terms belongs to $\mathcal{Z}^*(k, 2^w-1)$*, and hence equal to some* $\big((-1)^{v_1}2^{h_1} + \cdots + (-1)^{v_j}2^{h_j}\big) \bmod (2^w-1)$*, where* $0 \leqslant j \leqslant k$*,* $v_1,\ldots,v_j \in \{0,1\}$*, and* $h_1 > \cdots > h_j \in \{0,\ldots,w-1\}$*.*
Moreover, $(-1)^{u_{k+1}}2^{i_{k+1}} \bmod (2^w-1) = (-1)^{u_{k+1}}2^{i_{k+1} \bmod w} \bmod (2^w-1)$*.*

If $(i_{k+1} \bmod w) \notin \{h_1,\ldots,h_j\}$*, then the claim is true. Otherwise, suppose that* $i_{k+1} \bmod w$ *equals some* h_J*, and that* x *becomes*

$$\big((-1)^{v_1}2^{h_1} + \cdots + (-1)^{v_{J-1}}2^{h_{J-1}} + ((-1)^{v_J} + (-1)^{u_{k+1}})2^{h_J} \\ +(-1)^{v_{J+1}}2^{h_{J+1}} + \cdots + (-1)^{v_j}2^{h_j}\big) \bmod (2^w-1).$$

If $u_{k+1} = 1 - v_J$*, then* $x \in \mathcal{Z}^*(k-1, 2^w-1) \subset \mathcal{Z}^*(k+1, 2^w-1)$ *and the claim is true. Otherwise, assume that* $u_{k+1} = v_J = 0$ *(the case* $u_{k+1} = v_J = 1$ *is entirely analogous). Then,* x *is*

$$\big((-1)^{v_1}2^{h_1} + \cdots + (-1)^{v_{J-1}}2^{h_{J-1}} + 2^{h_J+1} \\ +(-1)^{v_{J+1}}2^{h_{J+1}} + \cdots + (-1)^{v_j}2^{h_j}\big) \bmod (2^w-1),$$

which belongs to $R(k) = \mathcal{Z}^*(k, 2^w-1) \subset \mathcal{Z}^*(k+1, 2^w-1)$*, completing thus the proof.*

Hence, $|\mathcal{Z}(k, b^w-1)|$ is at most $\sum_{j=0}^{k}\binom{w}{j}(2(b-1))^j$, as for each $0 \leqslant j \leqslant k$, there are $\binom{w}{j}$ ways to choose j pairwise distinct $i_1,\ldots,i_j$, and $(2(b-1))^j$ ways to choose $u_1,\ldots,u_j$ and $t_1,\ldots,t_j$.

Onwards, we suppose to work modulo $q = b^w - 1$, without explicitly mentioning it. Observe also that, as a result of Lemma 2, the set $\mathcal{Z}(k, b^w-1)$ depends only on b, w and k.

3.3.1 An On-Line Algorithm for String Matching with k Mismatches

The generalized algorithm (shown as Algorithm 2) works in a similar manner as the Rabin and Karp algorithm [75] (see Algorithm 1). It starts by setting $q = b^w - 1$, $s = 0$, and by computing $\hat{p} = p \bmod q$ and $\hat{t}_{(0)} = t_{(0)} \bmod q$, using Horner's rule and bringing into play the linearity of the modulo operation. Then, for each $0 \leqslant s \leqslant n-m$ it checks whether $(\hat{p} - \hat{t}_{(s)}) \bmod q \in \mathcal{Z}(k, q)$. If yes, it performs a character-by-character comparison of P and $T_{(s)}$. When incrementing s, the value $\hat{t}_{(s)}$ can be computed in constant time, as follows. For all $0 \leqslant s < n-m$, we have $\hat{t}_{(s+1)} = b \cdot (t_{(s)} - b^{m-1}T[s]) + T[s+m]$. Working modulo q, this equation becomes $\hat{t}_{(s+1)} = \big(b \cdot (\hat{t}_{(s)} - (b^{m-1} \bmod q)T[s]) + T[s+m]\big) \bmod q$. If we let $h =_{\text{def}} b^{m-1} \bmod q = b^{(m-1) \bmod w}$, we get $\hat{t}_{(s+1)} = \big(b \cdot (\hat{t}_{(s)} - h \cdot T[s]) + T[s+m]\big) \bmod q$.

Algorithm 2: String matching with k mismatches

Input: $T = T[0]T[1]\dots T[n-1]$, $P = P[0]P[1]\dots P[m-1]$, both over the alphabet $\Sigma = \{0, 1, \dots, b-1\}$, number of mismatches k ($0 \leqslant k < m$) and word length w.
Output: All pairs $\langle s, d_H(P, T_{(s)})\rangle$, where $0 \leqslant s \leqslant n-m$ and $d_H(P, T_{(s)}) \leqslant k$.

```
1  q ← b^w − 1;
2  h ← b^(m−1 mod w);
3  Z ← GenerateZ(k, q);
4  Solutions ← ∅;
5  p̂ ← t̂ ← 0;
6  for i ← 0 to m − 1 do
7      p̂ ← (b · p̂ + P[i]) mod q;
8      t̂ ← (b · t̂ + T[i]) mod q;
9  if (p̂ − t̂) mod q ∈ Z then
10     if d_H(P, T_(0)) ⩽ k then
11         Solutions ← Solutions ∪ {⟨0, d_H(P, T_(0))⟩};
12 for s ← 1 to n − m do
13     t̂ ← (b · (t̂ − h · T[s − 1]) + T[s + m − 1]) mod q;
14     if (p̂ − t̂) mod q ∈ Z then
15         if d_H(P, T_(s)) ⩽ k then
16             Solutions ← Solutions ∪ {⟨s, d_H(P, T_(s))⟩};
17 return Solutions;
```

In Algorithm 2 we assume that procedure GenerateZ(k, q) generates the set $\mathcal{Z}(k, b^w - 1)$, as expressed in Lemma 2.

In order to evaluate the expected complexity of the string matching phase of Algorithm 2, we follow the formalism of [30, Ch. 32.2]. We have to compute the time $c(q)$ the test $(\hat{p} - \hat{t}_{(s)}) \bmod q \in \mathcal{Z}$ on lines 9 and 14 takes, and the average number of false positives produced by it. If we denote by $p(q)$ the probability that at a specific shift $0 \leqslant s \leqslant n-m$ this test will produce a false positive, we can estimate the number of false positives as $n \cdot p(q)$. Considering ν to be the number of occurrences of P in T with at most k mismatches, the expected complexity is

$$\mathcal{O}\big(n \cdot c(q) + (m \cdot \nu + m \cdot n \cdot p(q))\big).$$

In many applications ν is small (*i.e.*, $\mathcal{O}(1)$) and if we choose q such that $n \cdot p(q) \leqslant 1$, then the expected complexity becomes $\mathcal{O}(n \cdot c(q) + m)$. The only values of $t_{(s)}$ for which $(\hat{p} - \hat{t}_{(s)}) \bmod q \in \mathcal{Z}(k, q)$, but $d_H(P, T_{(s)}) > k$ are of the form $p + z + j \cdot q$, where $z \in \mathcal{Z}(k, q)$ and $0 \leqslant j \leqslant \lfloor b^m/q \rfloor$. As we have at most $\lfloor b^m/q \rfloor |\mathcal{Z}(k, q)|$ such values, and there are at most b^m possible values for $t_{(s)}$, the probability that at a specific shift s, the test $(\hat{p} - \hat{t}_{(s)}) \bmod q \in \mathcal{Z}(k, q)$ produces a false positive is $p(q) \leqslant \frac{|\mathcal{Z}(k,q)|}{q}$, under the assumption that the operation $\bmod(b^w - 1)$ uniformly distributes numbers in the interval $[0 \dots q-1]$ (for example when $b^w - 1$ is a prime number).

Therefore, to attain the desired time complexity, one has to choose $q = b^w - 1$ such that $b \cdot q$ fits into a processor word and such that $q \geqslant n|\mathcal{Z}(k, q)|$.

Working on a 32-bit processor, with strings over the alphabet $\{0, 1, 2, 3\}$, limits w to 15, therefore, if n or k are large enough, a flurry of false positives are due to appear. If we use a 64-bit architecture, w is limited to 31, and hence the number of false positives drastically decreases. These numbers are computed in Table 3.1.

We choose to implement the test $(\hat{p} - \hat{t}_{(s)}) \bmod q \in \mathcal{Z}(k, q)$ by generating the set $\mathcal{Z}(k, q)$ beforehand, in time $\mathcal{O}(|\mathcal{Z}(k, q)|)$. The data structure storing it can be an ordered array, with search complexity $c(q) = \mathcal{O}(\log |\mathcal{Z}(k, q)|)$. A data structure more appropriate for unsuccessful queries, as

	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$
f. p. on 32 bits	3.73	339	13079	279959	3662224	30549760
f. p. on 64 bits	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0	2.4

Table 3.1: False positive with different architectures. The average number of false positives returned by the heuristic test $(\hat{p} - \hat{t}_s) \bmod q \in \mathcal{Z}(k, 4^w - 1)$, when $\Sigma = \{0, 1, 2, 3\}$, n =4G, and $w = 15$ (32-bit architecture) and $w = 31$ (64-bit architecture).

we expect most of them to be, is a trie, with worst case search time $c(q) = \mathcal{O}(w)$. However, due to better memory locality, a hash table with collisions resolved by chaining is preferred. Under the assumption of simple uniform hashing and using $\mathcal{O}(\alpha)$ memory, the average search complexity becomes $c(q) = \mathcal{O}(1 + |\mathcal{Z}(k, q)|/\alpha)$.

If one agrees to use an additional amount $\mathcal{O}(q)$ of memory, then $\mathcal{Z}(k, q)$ can be simply stored as a direct-address table $\mathcal{Z}[0 \dots q-1]$, where $\mathcal{Z}[z] = 1$ iff $z \in \mathcal{Z}(k, q)$, and thus $c(q) = \mathcal{O}(1)$.

The previous considerations allows us to state the following theorem:

Theorem 1 *Algorithm 2 solves the k-mismatch problem; if* $q = b^w - 1 \geqslant n|\mathcal{Z}(k, q)|$*, and if* $c(q)$ *denotes the complexity of testing membership in* $\mathcal{Z}(k, q)$*, its expected search complexity is* $\mathcal{O}(n \cdot c(q) + m + |\mathcal{Z}(k, q)|)$*.*

3.4 A randomized Numerical Aligner: rNA

In Section 3.3 we showed how the Rabin and Karp method can be extended in order to handle mismatches. Algorithm 2 is able to find all occurrences of a pattern P of length m in a text T of length n in average time $O(m + n)$. The algorithm can be easily adapted to solve also the *best-k-mismatch* problem, the problem of finding all the *best* occurrences of pattern P in T with at most k mismatches.

Algorithm 2, however, performs poorly when multiple patterns have to been searched. As a matter of facts every time a pattern P is processed the text T has to be read. If our pattern is an Illumina read (100 bp) and our text is the Human reference genome (3.2 Gbp) is clear that Algorithm 2 cannot scale. Moreover, as k increases the performances of Algorithm 2 decrease.

Algorithm 2 presented in Section 3.3 has been used has the core engine of an *NGS-aligner* designed to align the large amount of short sequences produced by Next Generation Sequencers. We called our aligner rNA (randomized Numerical Aligner). rNA is an hash-based aligner (refer to Chapter 2 for a detailed discussion) that makes extensive use of the seed-and-extend technique in order to solve the *best-k-mismatch* problem. Notably, most used aligners like BWA, BOWTIE and SOAP make extensive use of heuristics that accelerate the alignment task but reduce the sensitivity: rNA solves the best-k-mismatch problem in an exact way and its performances are comparable with those of the most used tools.

3.4.1 An exact string aligner

We will start showing how the standard Rabin and Karp algorithm can be revised and modified in order to obtain a *exact string aligner*. We will assume that all the patterns are of the same length m.

An exact string aligner is given a text T, of length n, and a collection $\mathcal{P}$ of patterns, and is required to find all exact occurrences of P in T, for every $P \in \mathcal{P}$. A possible strategy is to compute before-hand the fingerprints of all the patterns in $\mathcal{P}$ and store them in an appropriate data structure, in which every $\hat{t}_{(s)}$ $(0 \leqslant s \leqslant n - m)$ is searched for. If a matching fingerprint value is found, the corresponding pattern is compared with $T_{(s)}$. This approach takes time $\mathcal{O}(m|\mathcal{P}| + n)$ if a hash table is used to store the fingerprints of the patterns (as done *e.g.* in [123]), and time $\mathcal{O}(m|\mathcal{P}| + n \log |\mathcal{P}|)$, if they are stored as an ordered array.

Build a data structure on the patterns as several drawbacks: patterns are usually aligned only once against the reference, that instead is used several times. Moreover it often happens the total length of the patterns (*i.e.* the sum of the length of all the patterns) greatly exceeds the text length.

For this reasons we pre-processed the text, building in time $\mathcal{O}(n)$ and space $\mathcal{O}(n)$ the following index, graphically represented in Figure 3.1:

$$\mathcal{T} = \{\langle \hat{t}_{(s)}, s\rangle : 0 \leqslant s \leqslant n-m\}.$$

The shifts s in T which *may* be exact occurrences of a $P \in \mathcal{P}$ correspond to those pairs $\langle \hat{p}, s\rangle \in \mathcal{T}$. The set $\mathcal{T}$ can be stored in a way similar to a hash by chaining. We use an array indexed by numbers from 0 to $q-1$, having, for all $0 \leqslant r \leqslant q-1$, $\mathcal{T}[r] = \{s_1, \ldots, s_l\}$ iff for all $1 \leqslant i \leqslant l$, $\hat{t}_{(s_i)} = r$. Note that when doing exact alignment, q can be chosen to be $\Theta(n)$, according to the complexity analysis of Section 3.3.1. This exact aligner has average time complexity $\mathcal{O}(n+m|\mathcal{P}|)$.

3.4.2 A k-mismatch string aligner

In order to construct a string aligner that solves the best k-mismatch problem, Algorithm 2 can be adapted to use the index $\mathcal{T}$ over the text, by reverting from 'verification' back to 'generation'. For every $P \in \mathcal{P}$, we are interested in finding all the shifts s in T which *may* be occurrences of P with at most k mismatches. They correspond to those pairs $\langle \hat{t}_{(s)}, s\rangle \in \mathcal{T}$ such that $(\hat{p} - \hat{t}_{(s)}) \bmod q \in \mathcal{Z}(k,q)$. Using linearity of the modulo operation, we thus iteratively search in $\mathcal{T}$ all numbers $(\hat{p}-z) \bmod q$, for every $z \in \mathcal{Z}(k,q)$. For all shifts s such that $\langle (\hat{p}-z) \bmod q, s\rangle \in \mathcal{T}$, we check that indeed $d_H(P, T_{(s)}) \leqslant k$. The average complexity of a search for a pattern is thus $\mathcal{O}(m+|\mathcal{Z}(k,q)|)$, amounting to a total complexity of $\mathcal{O}(n+(m+|\mathcal{Z}(k,q)|)|\mathcal{P}|)$.

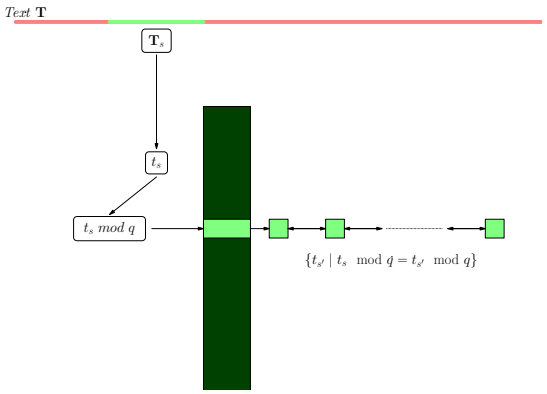


Figure 3.1: General Schema of the Hash Table constructed by rNA in order to store all the fingerprints of the text T.

However, the larger w is, the lower the probability of a false positive is, but the larger $|\mathcal{Z}(k,q)|$ gets, and vice-versa. We can remediate to this problem by a rather standard use (in this field) of the pigeon hole principle.

Definition 6 *Given a string $P = P[0]P[1]\ldots P[m-1]$ and a positive integer $1 \leqslant t \leqslant m$, for every $0 \leqslant i < t$, we denote by $P_{\lfloor m/t\rfloor}(i)$ its substring $P[i\lfloor m/t\rfloor]\ldots P[(i+1)\lfloor m/t\rfloor - 1]$ and call it the ith* block *of P.*

Note that the t blocks of a string P do not overlap, a crucial property for the following lemma to hold.

Lemma 3 *Let T be a text, $P = P[0]P[1]\ldots P[m-1]$ be a pattern, and t a positive integer, $1 \leqslant t \leqslant m$. If P occurs in T with at most k mismatches, then there is at least one block $P_{\lfloor m/t\rfloor}(i)$ of P that occurs in T with at most $\lfloor k/t\rfloor$ mismatches.*

Accordingly, instead of searching for an entire pattern P with at most k mismatches, we can perform t searches for all of the blocks of P, each with at most $\lfloor k/t \rfloor$ mismatches. Each occurrence of a block $P_{\lfloor m/t \rfloor}(i)$ $(0 \leqslant i < t)$ of P in T, with shift s, is an indication that P may occur in T with shift $s - i\lfloor m/t \rfloor$. As we are interested in finding the best occurrences of P in T, we will keep the smallest number of mismatches at which an occurrence of P has been found so far in a variable *best_k*. In this way, each block of the pattern is searched with at most $\lfloor best\_k/t \rfloor$ mismatches. The pseudo-code of the resulting procedure is given as Algorithm 3.

Algorithm 3: The randomized Numerical Aligner (**rNA**)

Input: Text $T = T[0]T[1]\dots T[n-1]$, a collection $\mathcal{P}$ of patterns of length m, all over the alphabet $\Sigma = \{0, 1, \dots, b-1\}$, number k of mismatches $(0 \leqslant k < m)$, the number t of blocks in which the patterns get divided $(1 \leqslant t \leqslant k+1)$, and word length w.

Output: For all $P \in \mathcal{P}$, all pairs $\langle s, d_H(P, T_{(s)})\rangle$, where $0 \leqslant s \leqslant n-m$, $d_H(P, T_{(s)}) \leqslant k$ and for all $0 \leqslant s' \leqslant n-m$, it holds that $d_H(P, T_{(s)}) \leqslant d_H(P, T_{(s')})$.

```
1  procedure SearchPattern(P)
2      for i ← 0 to t − 1 do
3          p̂_l(i) ← 0;
4          for j ← i · l to (i + 1) · l − 1 do
5              p̂_l(i) ← (b · p̂_l(i) + P[j]) mod q;
6      Solution ← ∅; best_k ← k;
7      exact_occurrence ← false; j ← 0;
8      while j < |Z(⌊best_k/t⌋, q)| do                    //for every witness Z[j]
9          i ← 0;
10         while i ⩽ t − 1 and (¬exact_occurrence) do          //for every block i
11             foreach s ∈ indexT[(p̂_l(i) − Z[j]) mod q] do      //for all shifts
12                 if s − i · l ⩾ 0 and d_H(P, T_(s−i·l)) ⩽ best_k then
13                     if d_H(P, T_(s−i·l)) < best_k then
14                         best_k ← d_H(P, T_(s−i·l));
15                         Solution ← ∅;
16                     Solution ← Solution ∪ {⟨s − i · l, best_k⟩};
17             if best_k = 0 then exact_occurrence ← true;
18         j ← j + 1;
19     print Solution;
20 q ← b^w − 1; l ← ⌊m/t⌋;                    //compute q and the block length t
21 indexT ←PreProcessText(T, b, l, q);
22 Z ←GenerateZ(k, q);
23 foreach P ∈ P do
24     SearchPattern(P);
```

Procedure PreProcessText builds the index over the text discussed in Section 3.4.1 by storing all fingerprints of length l of the text. We assume that procedure GenerateZ(k, q) returns an array containing the elements of the set $\mathcal{Z}(k, q)$, ordered in the following way: for all $0 \leqslant i < k$ the elements of $\mathcal{Z}(i, q)$ are placed before the elements of $\mathcal{Z}(i+1, q) \setminus \mathcal{Z}(i, q)$.

The procedure SearchPattern(P) starts by dividing the pattern in t blocks, each of length $l = \lfloor m/t \rfloor$. For each block $P_l(i)$ $(0 \leqslant i < t)$, its fingerprint $\hat{p}_l(i)$ is computed employing Horner's rule and the linearity of the modulo operation (lines 2 – 5). The variable *best_k* stores the smallest distance at which an occurrence of P has been found so far, while *exact_occurrence* indicates whether an *exact* occurrence has been found in the text.

For each index j $(0 \leqslant j < |\mathcal{Z}(\lfloor best\_k/t \rfloor, q)|)$, we iteratively search in the text every *block* $P_l(i)$ $(0 \leqslant i < t)$, with at most $\lfloor best\_k/t \rfloor$ mismatches (line 10). Every such shift s where the block i may occur is an indication that the *pattern* may occur at shift $s - i \cdot l$ with at most *best_k* mismatches (if, of course, $s - i \cdot l \geqslant 0$).

If this is indeed the case (line 12), we have to check whether the current occurrence is at distance

strictly smaller than *best_k* (line 13). If so, the variable *best_k* is updated with the current distance, and all the shifts s stored so far in the set SOLUTION are discarded. Anyhow, the current shift s together with *best_k* are added to SOLUTION. In other words, at every step of the computation, the set SOLUTION stores occurrences only at distance *best_k*.

Lastly, in line 17 we implement the following optimization: if the pattern occurs in an exact manner in the text, then the first block does as well. Since this block will indicate all exact occurrences, searching the remaining blocks of P brings no additional information. Therefore, we set *exact_occurrence* to true, stopping the search (this is true because *best_k* was changed to 0, hence the loop in line 8 is no longer executed).

3.4.3 Implementation Details

In Section 3.3 we showed how the Rabin and Karp algorithm for string matching can be extended to handle mismatches, and subsequently, in Section 3.4 we applied this idea to the implementation of a string aligner designed for NGS data.

Algorithm 3 is the first and most important step towards the design of a NGS aligner. However, when designing a complete software a large number of details must be considered: an NGS-aligner must be able to handle standard input formats (*fasta* and *fastq*) and output alignments in a standard format (SAM). Moreover, several performances issues must be taken into account: the aligner must efficiently handle texts of large size (human genome length is 3.2 Gbp), and it must be practically fast. Even more subtle matters must be solved: the aligner must be easy to install and to use in order to ease the software distribution.

Practical and Biological Problems

We will now list some of the most important practical and biological problems that must be faced while designing a short read aligner:

1. genome composition: until now we imaged a reference sequence as a single *long* text. In real scenarios we must keep in mind that the reference sequence is divided into *chromosomes* or into *scaffolds*, hence the input text consists of a database of *genomic* sequences $\mathcal{G} = \{T^1, T^2, \ldots, T^u\}$.

2. Resources requirements: the data structure being constructed must use a limited amount of resource (*i.e.* RAM). Therefore the implementation cannot rely on linked list like show in Figure 3.1.

3. Ambiguous bases: Σ, the finite alphabet, can be fixed to the four letter alphabet $\Sigma_{DNA} = \{A, C, G, T\}$. In practice both the text (called the *reference sequence*) and the reads can contain a certain number of ambiguous characters (caused by gaps in the assembly or by sequencing errors). Ambiguous bases are identify with the character N. Obviously every aligner must handle ambiguous bases.

4. Watson and Crick filament: as pointed out in Chapter 1 the DNA is a double-stranded molecule. Each DNA strand is connected to a complementary strand. Since the sequencers, in general, cannot indicate the strand from which each sequence has been read, given a read P, we must align both P and $\overline{P}$ (where $\overline{P}$ is the reverse complement of P). Moreover, reads are generally provided in pairs, at a known (estimated) distance and orientation.

5. Indels: it is often biologically interesting to align reads allowing not only mismatches, but also insertions and deletions (*indels*).

6. Multi-threading: NGS-aligners must allow multi-threading. Multi-threading is the basic parallel level that each aligner must implement in order to speed up read alignment.

7. Standard output format: first available NGS-sequencers were characterized by non-standard outputs format. The necessity to use alignment results for subsequent analysis oblige the community to find a common output format that all available NGS-aligners are supposed to implement.

Data Structures

Given the input genome $\mathcal{G} = \{T^1, T^2, \ldots, T^u\}$, rNA builds the string $T = T^1\$T^2\$\cdots\$T^u$, where \$ is a new character used as delimiter. Once a match is found inside T, its global coordinate is converted into a local coordinate inside a chromosome/scaffold, thanks to a lookup table.

The main data structure behind rNA is the hash table $indexT$. $indexT$ is implemented with two arrays, H and V. The former has length $q+1$ and contains pointers to V, while the latter has length equal to $|T|$ and contains pointers to the text. In position $H[r]$ we memorize the $rank$ of the fingerprint r, *i.e.*, the number of fingerprints less than r present in T. From position $V[H[i]]$ to position $V[H[i+1]-1]$ we store the shifts of T having fingerprint r. After having computed the fingerprint $\hat{p}$ of a read P, we perform the test in line 13 of Algorithm 3 for all these shifts.

Scanning the text two times, arrays H and V can be computed in-place, without any supplementary memory. Moreover, both them and T need to be in RAM during search phase. Hence, we need $4 \cdot q + 4 \cdot |T| + |T|$ bytes, if 4 bytes are used for each pointer, and each character of T is stored as one byte.

Ambiguous Bases and Read Filtering

The ambiguous bases problem is solved treating all N characters inside the reads as mismatches. During the fingerprint computation we simply generate a random character for each of them. In a similar way we treat ambiguous characters in the text: in the construction phase we randomly choose a non-ambiguous base, while in the alignment phase we treat them as mismatches.

Reads are usually provided in fastq format. A fastq file uses 4 lines per read. The first line is the header, which begins with the character @ and contains the read name. The second line is the read itself. The third line is a comment line while the last one is a string of the same length of the read which stores the quality of the read.

Low quality bases are likely to be reading mistakes and are usually concentrated at the beginning and at the end of the read (as a consequence of the chemical reactions used to read DNA). We developed a routine similar to the one implemented by the CLCbio Workbench [1] able to check the read's quality. We first trim the low quality bases at the beginning and at the end of the read. If after this process the remaining read has length and average quality higher than two predefined thresholds, the read is aligned, otherwise it is discarded.

Reverse Complement, Indels and Paired-end Mapping

In order to take care of the double stranded DNA nature, given the read P we first align P and soon after $\overline{P}$. Algorithm 3 must be slightly modified to compute the fingerprints of P and $\overline{P}$. Heuristics that allow to trim the search space can be used also on the reverse complemented read. It must be remembered that if P is found to occur with 0 mismatches, also $\overline{P}$ must be searched with 0 mismatches in order to find all the possible alignments.

It is of primary importance to align reads allowing indels. This is especially true when aligning reads belonging to a genome that is closely related to the reference genome. In such a case, we are particularly interested in discovering the presence of mutations like small insertions and small deletions (*i.e.* indels). If a read is not found at the requested Hamming distance, rNA scans again all the possible occurrences of the pattern in the reference (limiting the search only to the first seed) using this time a Smith-Waterman-like algorithm to locally align the read. It is clear that this procedure slows the algorithm.

Most of the sequencers are able to produce reads in pairs, by reading two sequences at a fixed distance and with a known orientation. Among the many advantages of this additional information,

let us only mention that it can be used to identify structural variations [90]. When aligning such a pair, rNA first returns the best occurrences for each read of the pair. If at least one of the two mates occurs in single copy, rNA sorts all the occurrences (possibly one) of the other read according to their position. At this point, a dichotomic search is performed to find a possible alignment of the two reads that satisfies both the distance and the orientation constraints.

Multi-threading and Output

Alignment is a highly parallelizable routine. Presently, rNA can be used on a multi-core machine: every CPU reads a chunk of n reads and aligns them against the reference. Every time a CPU finishes the alignment phase, it writes the result in the output file and reads the next chunk of reads. The chunk dimension is dependent on the specific machines being used.

Output is provided in the widely used SAM format [94], making rNA compatible with a large number of tools for post-processing alignments.

3.5 The Data Race Problem: mrNA

Recently, in [162], it was noticed that the "sequencing throughput has recently been improving at a rate of about fivefold per year, whereas computer performance generally follows '*Moore's Law*', doubling only every 18 or 24 months". We must add that the picture is even worse: Moore's Law is no more valid (see [79]) and now computer improvements are lower than in the last decades. Therefore, data production and software are two competitors in a speed race in which the former is the predictable winner. Software designers for NGS have to accomplish the difficult mission to overturn the forecast.

In general there are two ways to handle the previous situation. The first one relies on designing efficient algorithms that made extensive use of advanced sophisticated heuristics and *ad-hoc* data structures. The second approach is to use multiple computers and processors, implementing tools able to run on multiple CPUs (*e.g.*, multi-threading), either on tightly connected computers (*e.g.*, clusters) or on distant and shared machine over the network (*e.g.*, cloud computing).

Even though these two approaches must progress in parallel, the string alignment problem has known optimal solutions [168], so new and enhanced algorithms can only relatively speed up the computation. Moreover, due to the technical complexities involved at various stages of their design, most aligners (*e.g.* BWA, SOAP2, BOWTIE, and rNA itself) are limited to reference genomes of length 4 Gbp. While this limitation is in general accepted (human genome has length 3.2 Gbp) there are some situations in which this may became a significant stumbling block. First, several projects aiming at sequencing and assembling (*i.e.*, reconstructing) genomes of length greater than 4 Gbp (*e.g.*, the spruce genome project) have been launched. Moreover, in meta-genomic studies [186] it is important to simultaneously align against a large number of reference genomes (*e.g.*, *all* plants genomes) at the same time.

For these reasons (need of speed and necessity to align on a reference larger than 4 Ggb) we faced the problem of design a *distributed aligner*. The distributed version of rNA is implemented through MPI (mrNA) in order to use this instrument over a cluster of tightly connected machines.

mrNA uses the Master/Slave model to construct the distributed rNA-table, while it uses the Pipeline model (see Figure 3.2) in the alignment phase to avoid Master/Slave communication bottleneck. Moreover, the pipeline model allows us to store—for each read—the information concerning the current best alignment. As a consequence, if a node finds an occurrence of a read with $k' < k$ mismatches, then the following nodes will search only for occurrences with k' mismatches, reducing the overall time needed to align a read. It is worth noting that this mechanism is neither possible with the Master/Slave approach nor with naive grid distribution schemes in which every node works without receiving the necessary information from other nodes.

mrNA uses a synchronized message passing model to perform communication among nodes: the sender sends the message through a channel and waits until the message has been completely

and correctly acquired by the receiver. The receiver, on the other side, waits until the message has been completely received.

3.5.1 mrNA: rNA-table construction

mrNA needs n processes in order to split the reference into n pieces. Without loss of generality, we can image each process executed on a different node. The first node, the master, divides the whole reference into n similar-size chunks. $n-1$ chunks are sent by the master to the others $n-1$ nodes (slaves), while one chunk is kept locally. At this point the n nodes can independently compute and write to the disk n rNA-tables, one for each different reference genome's chunk. Once a node finished its job, it communicates the event to the master which has the duty to ensure that all the processes terminated successfully.

In this way the reference genome T is pre-processed and the n rNA-tables can be used to align reads using n nodes. When n nodes are used, the amount of memory required per each node is $4 \cdot q + 4 \cdot \frac{|T|}{n} + \frac{|T|}{n} = 4 \cdot q + 5 \cdot \frac{|T|}{n}$, where q is a number of the form $2^k - 1$.

3.5.2 mrNA: alignment

In the alignment phase n different processes are run in a distributed environment using the Pipeline model described in Figure 3.2. All the processes align the entire set of reads against a subset of the reference, moreover the first and the last processes have the task to read the input and write the output, respectively.

Every process has a unique associated number, taken between 0 and $n-1$, dubbed its *rank*. We will now describe and analyze the algorithm executed by a generic worker: Algorithm 4. The algorithm needs five parameters: *rank* is the process' rank, n is the total number of processes allocated for the computation, $input\_file$ is the file containing the reads, $output\_file$ is the file where output is written, and RT is the rNA-table (pre-loaded) that corresponds to the *rank* node.

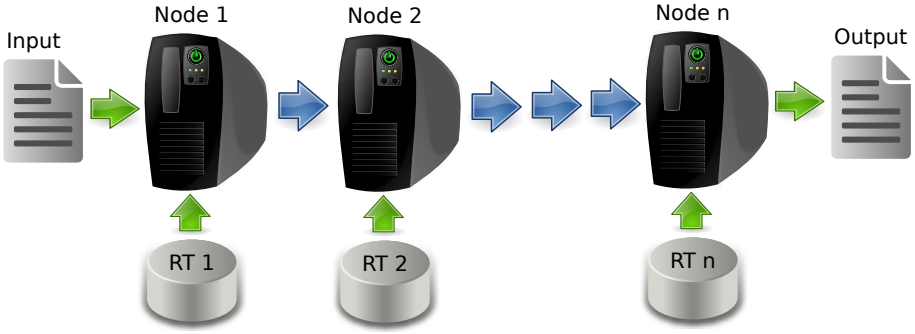


Figure 3.2: Pipeline model used in mrNA. Green vertical arrows represent I/O operation while blue horizontal arrows represent MPI channel communications. The n rNA-tables (RTs) are loaded on different nodes before the computation starts. Instead, the input is continuously read, processed, and sent to the next node. The last node, after the final alignment, writes the results to the output file.

The algorithm loops "forever" (Line 1), until an end condition is reached. The computation is divided in three phases. In the first phase the worker needs to acquire the information that will be computed. The first operation determines if the process is the "reader worker" ($rank = 0$, Line 2). If this is the case, the "reader" checks whether there are further reads to process (Line 3). In the former situation a sequence is read from the input file and stored in a variable r (Line 7), while in the latter case an "empty" signal is sent to the next node (Line 4) before the computation is stopped. The r variable is a complex object and stores the sequence being processed ($r.sequence$), the best alignment found so far if any ($r.alignment$), and the number of allowed mismatches ($r.k$). When a read is uploaded from the input it has no alignment and the search parameters are the

default ones (or the ones specified by the user). If the process is not the reader worker ($rank > 0$, Line 2), then it is a "generic worker" and it must receive data from the previous node (Line 11). If the empty read is received (Line 12) then the computation halts (Line 14) after the empty signal has been forwarded to the next node (Line 13).

In the second phase the worker computes the alignment. The worker uses the rNA algorithm (see [144]) to compute the best alignment for *r.sequence* against its reference's chunk allowing at most $r.k$ mismatches using the local rNA-table RT. The *align* function of Line 16 returns the alignment that has eventually found (t) and the number of mismatches used (∞ if no alignment has been found). If more than one best alignment is found then the algorithm chooses one randomly. In Line 17 the algorithm chooses the best alignment between the one stored in t and the one received by the previous node (*r.alignment*). If both alignments are *best alignments* one is randomly chosen (we guarantee that the read is chosen evenly among all possible alignments). In Line 18 the minimum number of mismatches used to compute the best alignment found so far is saved.

In the third and last phase the worker sends or writes the best alignment of r. If the node is the "writer worker" ($rank = n - 1$, Line 19) then the sequence together with its alignment are written to the output file (Line 20), otherwise (if $rank \neq n - 1$) the r object is sent to the next node (Line 22). Then the worker returns in Line 2, ready to process the next read (if any).

Algorithm 4: Generic worker algorithm

```
   Input: rank, n, input_file, output_file, RT
 1 while true do
       // First phase: acquire input
 2     if rank = 0 then // This is the reader
 3         if EOF(input_file) then
 4             send_to_node(rank + 1, ϵ);
 5             return ; // End the computation
 6         else // more data to read
 7             r.sequence ← read_from(input_file);
 8             r.alignment ← ϵ;
 9             r.k ← default_parameters();
10     else // Data received via network
11         r ← receive_from_node(rank − 1);
12         if r = ϵ then
               // Propagate end message
13             send_to_node(rank + 1, ϵ);
14             return ; // Stop the computation
15
       // Second phase: process input
16     (t, k′) ← align(RT, r.sequence, r.k);
17     r.alignment ← choose_best(r.alignment, t);
18     r.k ← min(r.k, k′);
       // Third phase: send/write r
19     if rank = n − 1 then // This is the writer
20         write_to_file(output_file, r);
21     else // Data must be sent to the next node
22         send_to_node(rank + 1, r);
```

3.6 Results

A mandatory step in order to present a new NGS-aligner is the demonstration that the new proposed aligner *behaves better* than other commonly used aligners. The words *"behaves better"* can have various meanings: a tool can *behaves better* than another because it aligns more reads, or because it *correctly* aligns more reads, or because it is simply faster, or because it uses less resources (*i.e.* RAM).

It is clear that when gauging aligner performances all these parameters must be considered and the evaluation must be done looking at the advantages and disadvantages of each tool.

For this reason we tested rNA both on simulated data (*Simulated Experiments*) and on real data (*Real Experiments*). The former have been used to gauge sensitivity (ability to align reads) and correctness (ability to align reads in the right position) at different levels of errors. The latter has been used to evaluate aligners in a real environment in order to judge speed/resource needed to align real reads.

3.6.1 A Race Between 5

We compared rNA with four widely used aligners. BWA [93], BOWTIE [89] and SOAP2 [99] are probably the three most used NGS-aligners in this moment. They are based on FM-index [41] and they differ from one and another for the heuristics used at several levels of the computation. In particular, for efficiency reasons, none of the previous solve the best-k-mismatch problem in a precise way, but they use heuristics that can generate sub-optimal alignments. BFAST [55] is a slightly different software: like rNA is an hash-based aligner and it is designed to be highly sensitive but it is particularly slow if compared to previous three.

The aim of this tests is to show how rNA has a sensitivity similar to BFAST but is able to achieve performances comparable to FM-index based aligners.

As noticed in Chapter 2, the number of available aligners is much larger then the 5 tested here. It is clear that a complete evaluation of al the available aligners is almost impossible and probably not even meaningful. As a matter of facts, only a small number of aligners are widely used by the global community: BWA, BOWTIE and SOAP2 are certainly among the most used aligners. Moreover aligners have several parameters that need to be "tuned" in order to obtain the best achievable result. This phase can be pretty expensive and imply a lot of manual work.

We distinguish between two alignment methods: mismatch alignment and mismatch/indels alignment. In the former only mismatches are allowed while, in the latter, also insertions/deletions events may occur in the alignment. rNA, BFAST, BWA, SOAP2 and BOWTIE allow mismatches, but only the first four allow also indels. The programs were ran on an 8 core Intel(R) Xeon(R) CPU E5420@2.50GHz with 32GB RAM machine. We always run the programs with 8 threads using in this way all the available processors.

Experiments set up

In order to achieve optimal performances with all the tools we tested them with different parameter. Despite being alignment time an important factor, we always preferred parameters that increase number of aligned reads at the price of lower performances to parameters that produce better performances at the price of a lower number of aligned reads.

For the simulated dataset we run programs with the following parameters:

- **rNA**: we specified the number of allowed mismatches with option `--errors`, we disabled the auto-trimming option (usually useful in practical situations) and we used 8 threads. The option `--indel` was used when indels were allowed. The result is stored in SAM format in the specified output file:

```
rNA --search --reference GENOME.rNA --query1 QUERY \
      --errors MIS  --no-auto-trim --threads 8 \
      --output QUERY.sam [--indels]
```

- **BFAST**: BFAST is divided into three distinct phases. In the first one we aligned the reads using all the 10 previously generated indexes. In the `localalign` phase we added the `-u` option when performing ungapped alignment. In the third and last phase we specified to keep all the alignments with the best score (`-a 4`) and to print the alignments in SAM format (`-O 1`):

```
bfast match -f GENOME -r QUERY -n 8 -A 0 \
       -i 1,2,3,4,5,6,7,8,9,10 > QUERY.match
bfast localalign -f GENOME -m QUERY.match -A 0 \
        -n 8 [-u] > QUERY.local
bfast postprocess -f GENOME -i QUERY.local -n 8 \
        -a 4 -O 1 > QUERY.sam
```

- **BOWTIE**: BOWTIE is probably the most cumbersome tool for what concerns options. It has two mutually exclusive alignments options. Authors suggest to use `-n` mode. For the sake of completeness we tried also the `-v` option observing slightly worse results. We noticed that `-e` option is of primary importance in order to align with mismatches. The default value (`-e 70`) allows to align only reads with 2 or at most 3 mismatches. By setting it to `180` we noticed a great improvement in aligning with mismatches. Option `-k 2` compels BOWTIE to print two best alignments when more than one is found. Recall that this is done in order to compute the alignment statistics. Options `--strata --best` force the aligner to output the best possible alignment(s) found.

```
bowtie -n 2 -e 180 -k 2 --best --strata --sam -p 8  -f \
        GENOME QUERY > QUERY.sam
```

- **BWA**: BWA alignment phase is divided into two different stages. The `-o` options was set to 0 when aligning only with mismatches and to 1 when aligning also with gaps. In this last case, the number of allowed differences between reads and reference (option `-n`) was set to the number of allowed mismatches plus maximum indel length (i.e., when aligning set $\mathcal{A}_g^i$ we allowed $i + 5$ differences):

```
bwa aln -n ERR -l 25 -t 8 -o [0|1]  GENOME QUERY > QUERY.sai
bwa samse GENOME QUERY.sai QUERY > QUERY.sam
```

- **SOAP2**: the parameter `-v` specifies the maximum number of allowed errors. The `-r 2` option specifies to return all the best alignments found. When gapped alignments were necessary we specified the `-g` option:

```
soap -a QUERY -D GENOME -o QUERY.soap -l 32 -v ERR -r 2 \
          -p 8 [-g 5]
soap2sam.pl QUERY.soap > QUERY.sam
```

The alignments commands used with the real dataset are slightly different from the ones used on the simulated dataset. The aim of this experiment was to show how rNA and the other aligners behave on a real dataset. In particular, we aligned the reads in paired read format. Some aligners (like BOWTIE) need to know the insert size: the reads used (SRX027713) have a nominal insert size of 288 bp with standard deviation of 26.6 bp. We decided to allow a larger insert in order to not discard the alignment of reads too far or too close. Therefore we set the minimum insert size to 50 and the maximum to 500 when requested.

- **rNA**: we specified the number of allowed mismatches with `--auto-errors` default option that allows at most one mismatch every 15 bases and we used 8 threads. We specified as input the two files containing the mates. The option `--indel` was used when indels were

allowed. We have tested also the use of `--no-auto-trim` options that disable automatic low quality extremities trimming (the option is discussed in the results). The result is stored in SAM format in the specified output file:

```
rNA --search --reference GENOME.rNA --output QUERY.sam \
    --query1 QUERY1 --query2 QUERY2 --auto-errors \
    --threads 8 [--no-auto-trim] [--indels]
```

- **BFAST**: BFAST is divided into three distinct phases. In the first one we aligned the reads using all of the 10 previously generated indexes. In the `localalign` phase we added the `-u` option when performing ungapped alignment. In the third and last phase we specified to choose uniquely the alignment with the best score (`-a 3`) and to print the alignments in SAM format (`-O 1`):

```
bfast match -f GENOME -r QUERY -n 8 -A 0 \
      -i 1,2,3,4,5,6,7,8,9,10 > QUERY.match
bfast localalign -f GENOME -m QUERY.match -A 0 \
      -n 8 [-u] > QUERY.local
bfast postprocess -f GENOME -i QUERY.local -n 8 \
      -a 3 -O 1 > QUERY.sam
```

- **BOWTIE**: BOWTIE requires the minimum and maximum insert size (options `-I 50 -X 500`). With options `-1` and `-2` we specified the first and second mate to be aligned. We require to output only one alignment (option `-k 1`). Like in the simulated experiments, also in this case the `-e` option is of fundamental importance, therefore we run BOWTIE with option `-e` set to `70` (default), `140`, and `180` and we reported the best results (option with largest number of aligned reads) that was

```
bowtie -1 QUERY1 -2 QUERY2 -n 2 -e 180 -l 50 -X 500 -k 1 \
       --sam -p 8 -q GENOME > QUERY.sam
```

- **BWA**: BWA alignment phase is divided into two different stages. The `-o` options was set to 0 when aligning only with mismatches and to 1 when aligning also with gaps. By default, the number of errors is computed by BWA on the fly.

```
bwa aln -t 8 -o [0|1] GENOME QUERY1 > QUERY1.sai
bwa aln -t 8 -o [0|1] GENOME QUERY2 > QUERY2.sai
bwa sampe GENOME QUERY1.sai QUERY2.sai \
    QUERY1 QUERY2 > QUERY.sam
```

- **SOAP2**: the parameter `-v` specifies the maximum number of allowed errors. When gapped alignments were necessary we specified the `-g` option:

```
soap -a QUERY1 -b QUERY2 -D GENOME -o QUERY.soap \
     -2 UNPAIRED.soap -p 8 [-g 5]
soap2sam.pl -p QUERY.soap > QUERY.sam
```

3.6.2 Precision and Accuracy: Simulated Experiments

In order to perform simulated experiments one needs a reference sequence and a read simulator (*i.e.* a tool able extract reads from the reference and introduce errors). We decide to use Chromosome 1 (in the following CHR) from the human genome assembly hg18 as reference. The first human chromosome has length approximately 250Mbp.

We employed he following procedure to produce the simulated datasets:

- from the reference sequence CHR we extracted the set $\mathcal{CHR}$ composed by 1 million of error free reads of length 100bp;
- we generated 9 sets named $\mathcal{CHR}^0, \mathcal{CHR}^1, \ldots, \mathcal{CHR}^8$ such that the set $\mathcal{CHR}^i$ contains the same reads contained in $\mathcal{CHR}$ but with exactly i mismatches in i randomly chosen positions;
- we generated 9 sets named $\mathcal{CHR}_g^0, \mathcal{CHR}_g^1, \ldots, \mathcal{CHR}_g^8$ such that the set $\mathcal{CHR}_g^i$ contains the same reads contained in $\mathcal{CHR}$ but with exactly i mismatches in i randomly chosen positions and one contiguous indels (insertion or deletion) of size at most 5 bp in a randomly chosen position.

In total we produced 18 millions of simulated reads grouped in 18 sets characterized by different rates of polymorphism and insertion/deletion events.

In order to assess the aligners' ability to *correctly* place reads we aligned against the reference CHR the 9 sets of simulated reads without indels ($\mathcal{CHR}^0, \ldots\ \mathcal{CHR}^8$) using rNA, BFAST, BWA, BOWTIE, and SOAP2 and with indels ($\mathcal{CHR}_g^0, \ldots\ \mathcal{CHR}_g^8$) using rNA, BFAST, BWA, and SOAP2.

We defined a read *correctly* placed if it is uniquely aligned and the alignment starting position is between ± 5 bases (the same criteria used in [55]) far from the real read's position (which is obviously known). Such experiments are possible only in simulated environments but help in giving a clear picture of the aligner's capabilities. When aligning with gaps, we did not check that if the indel has been correctly reconstructed.

For each tool and for each experiment we reported the total number of aligned reads, the number of uniquely aligned reads, the number of reads aligned in multiple positions, the number of correctly placed reads, and the number of wrongly placed reads (*i.e.*, unique aligned reads placed in the wrong position).

In Table 3.2 and in Figure 3.3 we summarize the results of the simulated experiments on the first human chromosome (CHR). When only mismatches are present (Table 3.2, Fig. 3.3(a), and Fig. 3.3(c)) all the tools behave almost in the same way up to three mismatches. After this threshold the performances of the BW-based aligners constantly decreases. Their capabilities to align reads slowly decrease as the number of mismatches introduced in the reads increase. rNA and BFAST have similar performances when considering the total number of aligned reads and the number of correctly aligned reads. However, in absolute values, rNA has a lower number of wrongly aligned reads than BFAST.

When reads contained also indels (Table 3.3, Fig. 3.3(b), and Fig. 3.3(d)), we see that SOAP2 is the worst performing tool. BFAST performances remain similar to the mismatch only case. rNA and BWA behave similarly until 3 mismatches; after this threshold rNA shows a greater ability in aligning reads. As we will see in Section 3.6.3 the higher BFAST's sensitivity is reached at the cost of a lower alignment speed.

3.6.3 Throughput and Alignment: Real Experiment

Benchmarks based on simulated data allows us to show the theoretical performances of the tools, however, in the NGS context also the practical performances are of primary importance. A tool able to align with a precision of 99.9% is useless if it needs months to align the data produced in few days by an NGS sequencer.

We tested rNA and the other tools on the Human genome (version hg18, from UCSC). We downloaded 166,622,914 reads from the Short Reads Archive SRA (SRX027713). The results are shown in Tables 3.4 and 3.5. For each aligner we reported the time needed to build the index (indexing), the time needed to align the reads (align), the RAM peak while aligning, the query throughput (query/core/sec) and the percentage of aligned reads. BFAST is designed to align reads with a large number of mismatches and indels. For this reason we filtered the BFAST alignment results in order to keep only those alignments with at most 7 mismatches, while we did not limit the number of indels. However we reported into brackets the total percentage of reads aligned by BFAST.

rNA

ERR	TOT	SING	MULT	COR	WR
0	909711	883431	26280	883431	0
1	909710	883195	26515	882816	379
2	909706	882855	26851	882155	700
3	909701	882565	27136	881605	960
4	909709	882161	27548	880951	1210
5	909709	881704	28005	880330	1374
6	909711	881210	28501	879719	1491
7	909704	880742	28962	879028	1714
8	909709	880217	29492	878339	1878

BFAST

ERR	TOT	SING	MULT	COR	WR
0	899571	876965	22606	876965	0
1	905700	880801	24899	876669	4132
2	906346	880250	26096	876112	4138
3	906626	879782	26844	875783	3999
4	906733	879143	27590	875119	4024
5	906913	878620	28293	874526	4094
6	907070	878094	28976	873801	4293
7	907185	877725	29460	873118	4607
8	907290	877071	30219	872195	4876

BWA

ERR	TOT	SING	MULT	COR	WR
0	909691	883413	26278	883413	0
1	909691	883178	26513	882799	379
2	909691	882840	26851	882140	700
3	883928	857501	26427	856496	1005
4	828955	803901	25054	802587	1314
5	751488	728268	23220	726741	1527
6	660105	639238	20867	637529	1709
7	564237	546165	18072	544283	1882
8	472346	456657	15689	454761	1896

SOAP2

ERR	TOT	SING	MULT	COR	WR
0	907804	844859	62945	844858	1
1	907456	848661	58795	847257	1404
2	906467	851826	54641	849590	2236
3	884158	834488	49670	831725	2763
4	837678	794201	43477	791411	2790
5	772732	736068	36664	733480	2588
6	694197	665072	29125	662849	2223
7	609935	590234	19701	588584	1650
8	2224	1299	925	0	1299

BOWTIE

ERR	TOT	SING	MULT	COR	WR
0	907804	844859	62945	844858	1
1	907456	848661	58795	847257	1404
2	906467	851826	54641	849590	2236
3	884158	834488	49670	831725	2763
4	837678	794201	43477	791411	2790
5	772732	736068	36664	733480	2588
6	694197	665072	29125	662849	2223
7	609935	590234	19701	588584	1650
8	2224	1299	925	0	1299

Table 3.2: rNA evaluation varying number of mismathces. The five tables summarize the results of aligning 1M simulated reads against the sequence of the first human chromosome. We reported for each experiment and for each tool the number of aligned reads (TOT), the number of reads aligned in a single position (SINGLE), the number of reads aligned in multiple positions (MUL), the number of correctly placed reads (COR), and the number of wrongly aligned reads (WR).

rNA

ERR	TOT	SINGLE	MULT	COR	WR
0	800147	778080	22067	774679	3401
1	800435	724852	75583	721627	3225
2	799485	722400	77085	719535	2865
3	794611	717183	77428	713712	3471
4	784563	706932	77631	702778	4154
5	735875	659522	76353	654749	4773
6	717270	641058	76212	635569	5489
7	681156	605585	75571	599319	6266
8	637383	562145	75238	555282	6863

BFAST

ERR	TOT	SINGLE	MULT	COR	WR
0	899571	876965	22606	876965	0
1	906207	881370	24837	875005	6365
2	906557	880957	25600	874329	6628
3	906721	880703	26018	873786	6917
4	906848	880487	26361	873385	7102
5	906865	879938	26927	872363	7575
6	906885	879723	27162	871789	7934
7	906911	879207	27704	870790	8417
8	906974	878939	28035	869729	9210

BWA

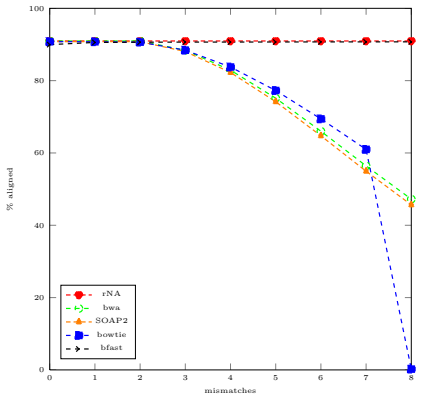
ERR	TOT	SINGLE	MULT	COR	WR
0	804297	780163	24134	778482	1681
1	789431	765568	23863	763366	2202
2	771851	748136	23715	745404	2732
3	743050	719935	23115	716618	3317
4	699824	677587	22237	673724	3863
5	645863	624983	20880	620533	4450
6	584536	564985	19551	559908	5077
7	520081	501927	18154	496368	5559
8	457068	440727	16341	434559	6168

SOAP2

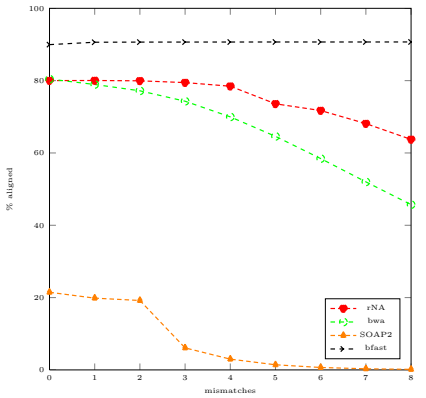
ERR	TOT	SINGLE	MULT	COR	WR
0	214671	207220	7451	204244	2976
1	198549	192624	5925	192379	245
2	191992	186212	5780	186020	192
3	60935	58759	2176	58542	217
4	29813	28715	1098	28582	133
5	14234	13668	566	13570	98
6	6668	6381	287	6316	65
7	3118	2963	155	2930	33
8	1482	1420	62	1395	25

Table 3.3: rNA evaluation varying number of mismathces and allowing one indel. The four tables summarize the results of aligning 1M simulated reads against the sequence of the first human chromosome. We reported for each experiment and for each tool the number of aligned reads (TOT), the number of reads aligned in a single position (SINGLE), the number of reads aligned in multiple positions (MUL), the number of correctly placed reads (COR), and the number of wrongly aligned reads (WR).

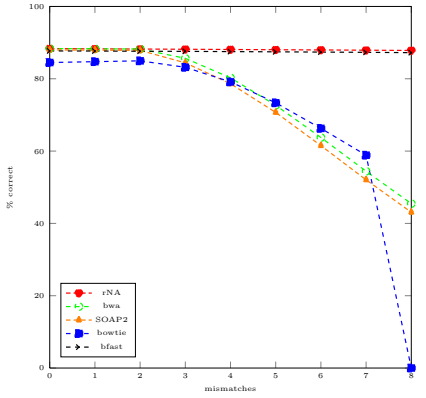
We run aligners as showed in Section 3.6.1. rNA was tested in two ways. The standard rNA's behaviour is to trim out the low quality bases before align a read. This behaviour is not present in all the other tools, that do not perform quality trimming. This situation was clearly unfair, so we run rNA also without this option: the rNA* column in Tables 3.4 and 3.5 refers to rNA ran with "`--no-auto-trim`" option. We have decided to use automatic trimming by default because, in our opinion, the result is more reliable.



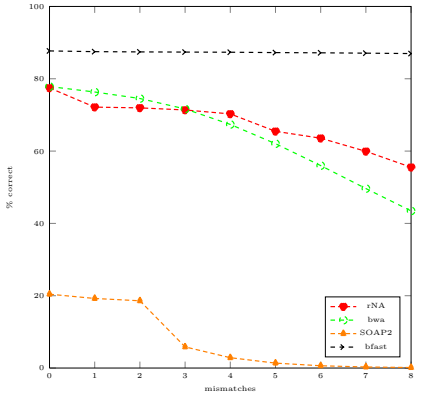
(a) Ratio between number of aligned reads and total reads allowing only mismatches.



(b) Ratio between number of aligned reads and total reads allowing mismatches and indels.



(c) Ratio between number of correctly aligned reads and total reads allowing only mismatches.



(d) Ratio between number of correctly aligned reads and total reads allowing mismatches and indels.

Figure 3.3: rNA evaluation on Human simulated datasets. The topmost plots show the percentage of aligned reads when varying the number of mismatches present in the reads (Fig. 3.3(a)) and when reads contain also indels (Fig. 3.3(b)). The other two figures show the percentage of correctly aligned reads when only mismatches are present (Fig. 3.3(c)) and when also indels are present (Fig. 3.3(d)).

In the first table we show the results obtained aligning reads *without* indels. BFAST is able to align 94% of the reads, but when counting only reads with up to 7 mismatches the total number of aligned reads is similar to the one of rNA and BWA. BFAST is extremely slower than all the other tools and it requires an unpractical amount of time to align the entire dataset. BWA and rNA are aligners able to align the huge amount of reads in acceptable time. On the one hand BWA uses less memory and requires ∼8 hours to align the enire dataset, on the other hand rNA is able to align an additional 3% of reads than BWA at the cost of using more memory and more time. SOAP2 has a throughput comparable to BWA, while BOWTIE is the fastest tool. As it can be noticed, these two tools align an amount of reads that is significantly (more than 10%) lower than rNA and BWA. It can appear strange that rNA used without the auto-trimming option (rNA* column) align more than rNA with this option on (rNA column). However this is explained by the fact that rNA with auto trimming option discards (*i.e.* do not align) all the the reads that have a low quality or that are shorter than a user predefined threshold (by default 25) after the trimming phase. The fact that reads have a low quality do not necessary mean that these reads contains a lot of errors (it is true that they are likely to contain a lot of errors). However, it is of doubtful

meaning align reads of low quality. Moreover, in subsequent analysis like SNP calling, it is often required high quality to "call" a SNP.

Similar results are summarized in Table 3.5 where the same dataset has been aligned allowing also indels. Again, BFAST is the tool able to align more data at the price of an unthinkable amount of time. rNA is again able to align more reads than BWA.

	rNA	rNA*	BFAST	BOWTIE	BWA	SOAP2
indexing (hh:mm:ss)	00:47:51		49:47:01	02:49:35	01:43:22	01:30:52
alignment (hh:mm:ss)	29:22:01	20:10:35	510:33:32	1:21:02	8:05:30	10:15:7
max used RAM	19.8GB	19.8GB	17.4GB	3.5GB	3.7GB	24.0GB
query/core/sec	197	286	11	4283	714	564
% aligned	77,15%	79.41%	74.12% (94.15%)	67.19%	76,25%	67.74%

Table 3.4: rNA evaluation on a real dataset (166,622,914 Illumina 100 bp reads) allowing mismatches only.

	rNA	rNA*	BFAST	BOWTIE	BWA	SOAP2
indexing (hh:mm:ss)	00:47:51		49:47:01	02:49:35	01:43:22	01:30:52
alignment (hh:mm:ss)	108:00:42	314:48:00	822:11:45	n.a.	14:6:15	10:37:28
max used RAM	19.8GB	19.8GB	17.4GB	n.a.	3.7GB	24.0GB
query/core/sec	53	18	7	n.a.	410	544
% aligned	78.21%	85.04%	84.24% (94.03%)	n.a.	76.91%	68.62%

Table 3.5: rNA evaluation on a real dataset (166,622,914 Illumina 100 bp reads) allowing mismatches and indels.

3.6.4 Aligning Over the Network: mrNA Performances

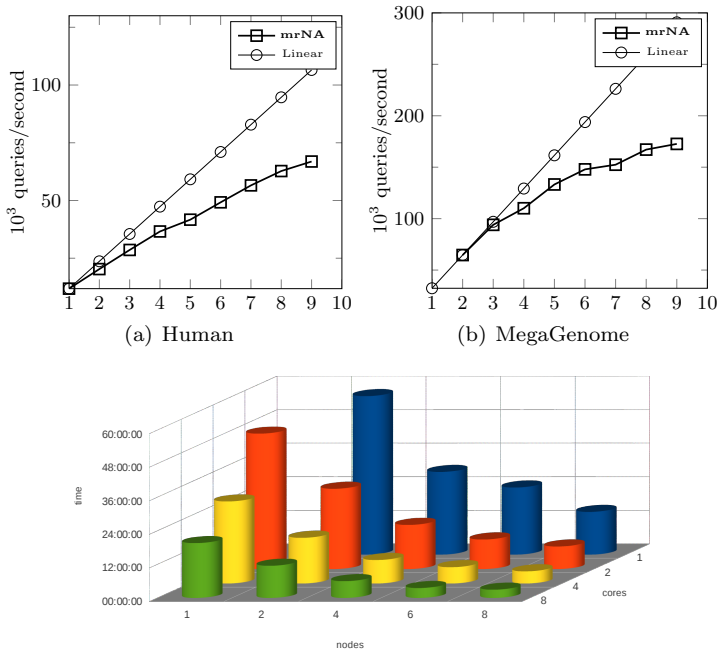
We also extensively evaluated the performances of mrNA, the distributed version of rNA. In order to test mrNA's performances we performed two large tests. The first one consisted in aligning a set of reads against the human genome, while the second consisted in aligning another set of reads against 12 plants genomes merged into a single reference that we call *MegaGenome*. Results are presented in figure 3.4. Moreover, we investigated the performances of the distributed implementation by studying how performances are affected by the variation of the number of nodes and threads per nodes in mrNA.

We aligned against the Human Genome reference a set of 3,257,108 reads of length 100 bp belonging to a Korean individual downloaded from the Short Read Archive (SRX011536). Against the MegaGenome we aligned 33,675,544 sequences of length 100 bp belonging to a grapevine variety (Sangiovese) produced at IGA laboratory.

In Figure 3.4 we can appreciate the performance of mrNA on the two datasets when varying the number of nodes and leaving unchanged the number of threads (8 per node) and the number of allowed mismatches (7 per read). For example, in order to align the reads against the Human Genome using 4 nodes, the reference has been divided into chunks of $\sim$ 800 Mbp and each node searched independently in its chunk using 8 threads.

In Figure 3.4(a) we can appreciate the results on the Human Genome. mrNA performances increase with the number of nodes. The performances of the algorithm are close to the theoretical best performance. Aligning the dataset against the human genome with only one node requires 46 minutes, while with 9 nodes it takes only 8 minutes. Similar results can be seen in Figure 3.4(b). It is worth stressing again the fact that the most popular aligners for short sequences (SOAP2 and BWA) are not able to align a set of reads against a genome of size greater than 4 Gbp. The aim of these experiments was to show how mrNA is able to work on reference genomes of every size. To the best of our knowledge, mrNA is the only available aligner for NGS able to align against genomes of size larger than 4 Gbp.

In Figure 3.4(c) we reported the results of aligning 166,622,914 reads of length 101 bp downloaded from the Short Read Archive (SRX027713) against the Human reference genome varying the number of cores and the number of threads per core, and allowing again at most 7 mismatches. We omit from the histogram the time of the experiment with one node and one thread for graphical



(a) Human (b) MegaGenome

(c) Nodes vs Cores

Figure 3.4: mrNA performances. In Figures 3.4(a) and 3.4(b) we summarise the results of running mrNA on a cluster using up to 9 nodes (X-axis). We run mrNA using one process per node and 8 threads per process. Figure 3.4(a) shows mrNA's performances on the human genome (hg18). Figure 3.4(b) shows mrNA's performances on the MegaGenome of length 6 Gbp (union of 11 plant genomes). Figure 3.4(c) shows mrNA executions time (Y-axis, hh:mm:ss) varying the number of nodes (X-axis) and threads per node (Z-axis).

reasons. The histogram in Figure 3.4(c) allows us to notice how running mrNA on 1 node with 8 threads (*i.e.* the standard way in which rNA is used) is slower than running it on 8 nodes with one thread per node. This is a direct consequence of the fact that each hash table contains less values and therefore there are fewer extension to make (*i.e.*, fewer false positives). Another explanation of such nice behavior is the effectiveness of the heuristic to communicate the current best solution between adjacent nodes.

3.7 Future Work and Conclusions

Throughout this Chapter we described rNA a short read aligner. Our aim was not only to explain the theory and the technicalities behind such a tool, but also to show how an NGS-aligner needs constant improvements and enhancements. rNA can be improved in several ways. The *seed-and-extend* approach can be substituted by a *q-gram* approach (see Chapter 2 for details). This enhancement will not cause large changes in rNA core algorithm (extended version of Rabin and Karp algorithm) but it will reduce the number of false positives and it will allow faster running times, especially when aligning with indels. Another possible way to boost rNA's performances is to make extensive use of the SSE instruction sets provided by state of the art CPUs. Processors are able to execute in almost constant time some operations that can boost alignment phase, with and without indels.

As noticed at the beginning of this Chapter, aligners needs to be revised and sometimes, redesigned, every six month in order to keep the pace with data production. In this Chapter we deeply analyse the birth, the growth and the evolution of a NGS aligner. In the last section we saw that rNA is an aligner able to compete with widely used solutions. In particular, with reference to others tools, rNA offers a greater sensitivity at the cost of a feasible increment in resources (time and space). The fact that rNA has been downloaded and used by several groups, and the fact that we constantly receive positives feedbacks from the community is a clear sign that rNA can have a future.

It is worth noting, at this point, how the most time consuming phase of the rNA-project has been the tool standardization. In other words the task to create a package easy to use, easy to install, able to handle standard input and output formats has been the most time consuming phase. However, the emotions and the feelings that we experienced distributing to the bioinformatics community a complete tool, used by several groups around the world, are worth the effort.

We already pointed out that compare and evaluate different aligners is not an easy task. First of all is not possible to compare all available software due to their large number, moreover each software is highly dependent on the parameters used that must be carefully chosen. We adopt the strategy to compare rNA against the most widely used aligners (BWA, BOWTIE and SOAP2) and against an highly sensitive one (BFAST). We showed all parameters used and we explained all decisions we took. However several other experiments could have been done: for example we did not test how precision and accuracy vary when aligning paired reads, or how performances improve or decrease when aligning real quality filtered reads. We believe that in order to allow a fair comparison among the available aligners a standard repository against which all software can be tested must be created. Similar repositories are available for *de novo* assembly tools (*i.e.*, assemblathon). Such repositories allow the global community to have a clear idea of performances achievable with different tools.

II

The Assembly Problem

4

De Novo Assembly

De novo whole-genome sequence assembly (WGSA) is the task of reconstructing the genome sequence from a large number of short sequences (i.e., *reads*) with no additional locational information or knowledge of the underlying genome's structure. *De novo* assembly is, probably, the most challenging and studied problem of current genomics.

Significant efforts have been done to formalize and study from a theoretical point of view *de novo* assembly problem. In [127] Pop and Nagarajan showed that different assembly formulations are NP-complete but that under certain assumptions the problem becomes easy to solve. The main message resulting from their analysis is that the reduction of the *de novo* assembly problem to the Shortest Common Super-String Problem (SCSP) is definitely not realistic and, probably, not even useful.

Many tools have been proposed to solve the Assembly Problem. Some of them demonstrated practically good results in particular in the Sanger sequencing context. However, as noticed in [134], all assembly tools are based on a small number of algorithms and differ from each other in the details of how they deal with errors, inconsistencies, and ambiguities.

Most of the published papers describing individual tools include a comparison with other already published assemblers, usually showing an improvement of their results. These considerations, together with the large number of available solutions and available tools, demonstrate on one hand how much this area is brisk, and, on the other hand, that there is not a widely accepted tool/solution. Moreover, there is not a clear way to compare different assemblers and assemblies, indeed standard statistics like contigs number and average contig length have recently been criticized [130]. Recently, doubts on completeness and correctness of NGS-based have been raised [6]. Assembly validation problem is becoming every day more pressing as a consequence of the large number of running projects.

The aim of this Chapter is to present and analyse *de novo* assembly from a theoretical and practical point of view. Section 4.1 will be focused on the computational methods proposed so far, with particular attention to their complexity. In Section 4.2 we will show how the supposedly intractable *de novo* assembly problem is, in practice, solved by a large number of tools. In Section 4.3 we will explain how the results produced by *de novo* assembly tools can be analysed and evaluated. Lastly, in Section 4.4 we will see how in practice assemblers behave on real data.

It is worth stressing the counter-intuitive nature of the *de novo* assembly problem: even though Assembly Problem is formulated as an NP-complete problem, tools based on heuristics and greedy strategies have so far been able to solve the problem in a satisfactory way. This can be either a consequence of the reductions needed to formalize the problem or of the fact that computational analysis always work with worst case scenarios that are rare (or not present) in real biological data.

4.1 Computational Problems of *De Novo* Assembly

In Shotgun Sequencing Method (SSM), fragments (*i.e.*, *reads*) are randomly sampled and read throughout the genome, using one of the various methods presented in Chapter 1. The genome sequence reconstruction is accomplish *assembling* the fragments according to their overlaps. A

mandatory condition to compute reliable overlaps, is the fact that every genome's position must occur in more than one read. The genome is therefore oversampled. We define the ratio between the total length of the reads and the (expected) genome length as *coverage*. More formally, if $\mathcal{R} = \{r_1, r_2, \ldots, r_n\}$ is the set of reads sampled from the genome G of length $|G|$, then we say that G has been sequenced with coverage C:

$$C = \frac{\sum_{i=1}^{n} |r_i|}{|G|}$$

We will often use the notation $C\times$ to indicate that the genome has been sequenced with coverage C.

The assembly problem (AP) is the problem to reconstruct the genome G starting from the set of reads $\mathcal{R}$:

Definition 7 (Assembly Problem (AP)) *Given the set $\mathcal{R} = \{r_1, r_2, \ldots, r_n\}$ of reads sequenced from the unknown genome G, reconstruct G.*

The basic strategy to solve the Assembly Problem is to compute overlaps between reads and to use this information to reconstruct the genome sequence, in a way similar to a jigsaw puzzle. This strategy is based on the assumption that similar sequences (*i.e.*, overlapping reads) are likely to belong to the same genomic region. Even though this procedure may seem straightforward, there are several points that make it practically difficult. First, reads may not assemble due to incomplete coverage of the original sequence (*i.e.*, regions that are *biologically* difficult to sequence). Second, all sequencing technologies are affected by sequencing errors (mismatches, insertions, and deletions) that make the overlap computation much harder. Third, overlaps between reads can occur by chance and not as a consequence of the fact that reads were sequenced from the same region. In projects involving hundreds of thousands of reads, spurious overlaps are not negligible. Fourth, DNA is double-stranded, and a particular fragment may have come either from one strand or from the other. Finally, the Assembly Problem worst point is the presence of (exact/inexact) repeats. Genomes sequences contain nearly identical repeated structures whose length can vary a lot. As shown in Figure 4.1 repeats cause several problems: the Figure shows how overlaps do not allow us to univocally assemble the original sequence.

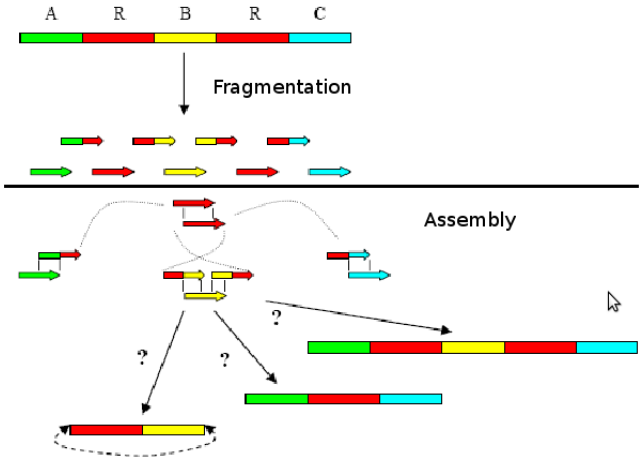


Figure 4.1: Repeats and *De Novo* Assembly. A possible *de novo* assembly scenario: the repeated sequence R (red) cause multiple scenarios that cannot be resolved using only overlap information.

It should be mentioned that several sequencing technologies also allow for the production of *paired reads* (or *mate pairs*) where reads are generated in pairs at a known distance (*i.e.*, with

a known *insert size*) and orientation between them. It is clear that this information is useful to resolve situations similar to those represented in Figure 4.1. Moreover, the presence of paired reads at correct (insert size) distance can be used to assess the correctness of the assembly. However, NGS technologies are characterized by relative short inserts, that can only partially help to resolve ambiguities.

The Assembly Problem is solved by software dubbed *assemblers*. Assemblers explicitly or implicitly represent reads and overlaps through a graph. The advantage of a graph structure is the possibility to reduce the assembly problem to other known problems and, therefore, study and analyse its complexity and eventually understand the limits of current approaches.

Different graph representations lead to different assembly models that are subsequently studied with the purpose of understanding assembly problem complexity. As a matter of facts, several *de novo* assembly formulations have been shown to be NP-hard: three among the most popular ones are the Shortest Common Superstring Problem (SCSP) [44], the Overlap/String Graph (OLG) [76] and the de Bruijn Graph (DBG) [127].

Even though most popular formulations have been proved to be NP-hard, a large number of tools (*assemblers*) have been successfully used to solve the Assembly Problem. The hardness results have driven the development of heuristic solutions several of which turned out to be quite successful [125, 11]. The success of such heuristics approaches suggests that the hardness represents the worst case scenario that rarely appears in real datasets [127]. However, the appearance and the spread of NGS technologies have reopened the discussion about Assembly Problem complexity. Particular attention have been paid to the computational impact of short sequences on the Assembly Problem.

4.1.1 Shortest Common Superstring Problem (SCSP)

One of the earliest approaches modelled the Assembly Problem as the task of finding the *Shortest Common Superstring* (SCS) of the reads based on a parsimony assumption. More formally, the SCS problem is defined in the following way:

Definition 8 (Shortest Common Superstring Problem (SCSP)) *Given* $\mathcal{R} = \{r_1, r_2, \ldots r_n\}$ *a set of sequences (i.e., reads) find the shortest string* R *such that* $\forall r_i \in \mathcal{R}$ r_i *is a subsequence of* R.

SCSP is a well known NP-complete problem [44], however it is widely accepted that this formulation does not correctly represent Assembly Problem as it fails to encode several of the points discussed at the beginning of Section 4.1. SCSP does not model the double stranded DNA nature, it does not take into account errors in the reads and, more importantly, it fails in reconstructing repeats.

The last point is particularly important: there is no biological reason to reconstruct the shortest common superstring of the sequenced reads. A strategy that follows this schema will lead to many wrongly reconstructed regions (*i.e.*, *mis-assemblies*). This is a direct consequence of the fact that in a shortest superstring repeats are collapsed, while we are interested in reconstructing the original DNA sequence. However, this NP-hard result has been often used to justify heuristics and greedy strategies employed to solve the Assembly Problem.

4.1.2 Overlap and String Graphs

Let v and w be two reads over the DNA alphabet Σ_{DNA}, let us indicate with $v[i..j]$ the substring of v that starts at i and ends in j. x is said to *overlap* y if there exists a maximal non-empty suffix of x of length o ($x[|x|-o, |x|-1]$) that matches a prefix of y of length o ($y[0, o-1]$). Let us denote with $ov(x, y) = o$ the overlap length between x and y.

Definition 9 (Overlap Graph) *Let* $\mathcal{R} = \{r_1, r_2, \ldots r_n\}$ *be the set of sequenced reads from the organism* G. *The Overlap Graph with minimum overlap threshold* k *is a bidirected, weighted graph* $OG^k(\mathcal{R}) = (V, E)$ *defined as:*

- $V = \{r_1, r_2, \ldots r_n\}$, *each read corresponds to a vertex;*

- $E = \{(r_i, r_j) | ov(r_i, r_j) \geq k, r_i, r_j \in \mathcal{R}\}$;
- $w(r_i, r_j) = |r_j| - o(r_i, r_j)$;

Given a path in an overlap graph, we can associate to such a path a *string-path*. A string-path is defined as the concatenation of the strings corresponding to the nodes on the path, where only one copy of the overlap string is kept. For instance, given the nodes $x = AAACCTTT$, $y = CTTTAGAGAG$, and $z = AGAGTATAG$ and the path $x \rightarrow y \rightarrow z$, then the corresponding string-path is $AAACAGAGAGTATAG$.

The overlap graph contains all the information about overlaps between reads and can be used to solve the Assembly Problem. As a matter of facts, the Overlap Graph $OG^k(\mathcal{R})$ is never used: as noticed by Myers in [124] the Overlap Graph can be sensibly reduced by a sequence of graph transformations aimed at discarding useless and/or redundant nodes and edges. In particular it is possible to remove *contained reads* and *transitively inferable edges*: in the former case, a read contained in another one does not give any valuable contribution to the assembly process (the containing read is more informative) and therefore can be discarded; in the latter case we have that reads y and z overlap x, and z overlaps y. In this situation the overlap of z to x can be *inferred* from the others overlaps and can therefore be removed from the graph.

The String Graph $SG^k(\mathcal{R})$ is obtained from the overlap graph $OG^k(\mathcal{R})$ by removing contained edges and transitively inferable edges. The String Graph can be computed using the algorithm proposed by Myers in [124] in polynomial time.

Once the String Graph $SG^k(\mathcal{R})$ is available, the idea is to visit it and obtain in such a way a solution to the Assembly Problem. In [124] and in [127] the Assembly Problem is solved by finding a generalized Hamiltonian Path (*i.e.*, find a path of minimal length such that every node is visited at least once) in $SG^k(\mathcal{R})$.

Overlap and String Graph formulations better represent Assembly Problem. The double stranded DNA's nature is taken into account while computing all the possible overlaps and keeping track of the orientations thanks to bidirected edges. Also errors in reads can be handled allowing errors (mismatches, insertion, and deletions) in the overlapping computation (at the price of a highest computing time). Moreover, these frameworks allow to represent the repeated nature of genomes: for example Myers in [124] proposed to mark edges as required, exact and optional meaning that they should be visited at least one time, exactly one time or they can either be visited or not.

Finding a minimum length generalized Hamiltonian Path in a string graph is shown to be NP-complete by Nagarajan and Pop in [127].

4.1.3 De Bruijn Graphs

Another commonly used graph approach is known under the name of de Bruijn Graph. This framework was first suggested in Sanger sequencing [66], based on a proposal for assembling using the old Sequencing By Hybridization technique [137]. However, it has become commonly used after NGS appearance thanks to the work done by Pavzner et al. [142].

De Bruijn Graphs [33] have been introduced for the first time by the Dutch mathematician Nicolaas Govert de Bruijn (born 9 July 1918). In graph theory a n-dimensional de Bruijn Graph of m symbols is a directed graph representing overlaps between sequences of symbols. In general a de Bruijn graph of degree n over an alphabet of m symbols is formed by m^n nodes. The set of nodes is composed by all the possible strings of length n over the m alphabet symbols ($V = \Sigma^n$), while an edge connects two nodes x and y if and only if the the $n-1$ suffix of x exactly overlaps the $n-1$ prefix of y.

In the context of genome assembly a slightly different (and simplified) version of the de Bruijn Graph is used:

Definition 10 (De Bruijn Graph) *Let $\mathcal{R} = \{r_1, r_2, \ldots r_n\}$ be the set of sequenced reads from the organism G. The de Bruijn Graph of order k is a directed graph $BG^k(\mathcal{R}) = (V, E)$ defined as:*

- $V = \{r[j, \ldots, j+k-1] \mid \exists r \in \mathcal{R} \wedge |r| \geq k \wedge j \in \{0, \ldots |r_i| - k\}\}$
- $E = \{(p_i, p_j) \in V \times V \mid p_i[1, k-1] = p_j[0, k-2]\}$

In the DBG representation a vertex represents a k-mer, while an edge represents a $(k+1)$-mer. All k-mers belong to the sequenced reads in $\mathcal{R}$ and, therefore, to the sequenced genome G (in reality, many reads might contain errors, we will see in Section 4.2 how these situations are handled). Within this framework, the Assembly Problem can be formulated as the problem of finding a path that visits all the edges exactly once. This problem is named the *Eulerian Path Problem*, a well known problem solvable in *polynomial* time.

Actually, the formulation just stated hides various problems: first of all a de Bruijn Graph can have more than one Eulerian Path (see [78]) and it is not clear which is the right one, secondly repeats still cause mis-assemblies and last but not least, the Eulerian Path generated can contain subwalks *not supported* by any read in $\mathcal{R}$.

Many of these problems, in particular the last one, are circumvented with the notion of *read-paths* or *read-walks*: a read $r \in \mathcal{R}$ corresponds to paths/walks in $BG^k(\mathcal{R})$ via the function $w(r) = r[1, k] \rightarrow r[2, k+1] \rightarrow \cdots \rightarrow r[|r| - k + 1, |r|]$. With this notion in mind, the Assembly Problem can be now formulated as the problem of finding a walk (in general dubbed *superwalk* or *superpath* [114, 127]) that uses each of the read-paths at least once. In this way the generated sequence will contain each read.

Finding a *superwalk* is an NP-complete problem, this can be demonstrated by reducing SCSP to it [127].

4.2 *De Novo* Assembly Strategies

The discussion and the results showed in Section 4.1 depict a dusky picture of *de novo* assembly. All proposed formulation are NP-complete. This situation, however, does not create a stumbling block: a large number of *assemblers* are available to effectively assemble genomes. Assemblers are based on a large number of heuristics and greedy strategies and many of these tools have proved their capabilities in a large number of projects ([175, 111, 60]).

Theoretically, the assembler's output should consists of a number of sequences equal to the number of chromosomes of the sequenced organism. As a matter of facts, most of assemblies consist of a larger number of contiguous sequences dubbed *contigs*. When possible, contigs are grouped in *scaffolds*, that can be represented as ordered lists of contigs in which the distance and the orientation within different contigs is known.

The large volume of ongoing research in the field of sequence assembly makes it difficult to keep the pace with all the different available techniques and their implementations (see Table 4.1). A common feature of existing assemblers is that they represent the reads by using (implicitly or explicitly) some types of graph data structure. In [119] *de novo* assemblers are divided into three main categories based on the core algorithm used: Greedy Graph Approach (Greedy), Overlap/Layout/Consensus (OLC) and de Bruijn Graph (DBG).

The Greedy assemblers, usually called *Seed-and-Extend* assemblers, apply one basic operation: given any read or contig, extend it by adding more reads or contigs [119]. This basic operation is repeated until no further extension is possible. At each extension step the highest scoring overlap is used to proceed. Regardless of all heuristics and refined implementations of such a technique, greedy assemblers fail in reconstructing even short repeats (*i.e.*, repeats longer then read length) and are not able to manage the large amount of reads to be dealt with while assembling a plant or a mammalian genome.

The OLC approach has demonstrated its capabilities in the Sanger Sequecing Projects. Assemblers like ARACHNE [11] and PCAP [61] implement this strategy. OLC assemblers represent the reads in an *overlap/string graphs* [124] in which reads are nodes while an overlap between two read r_i and r_j of length k is represented by an edge of weight $w(r_i, r_j) = r_j - k$. As suggested by their name, OLC assemblers go through three distinct phases: during the Overlap phase an all-against-all comparison between reads is performed with the purpose of building the Overlap/String

Name	Algorithm	Author	Year
Arachne WGA	OLC	Batzoglou, S. *et al.*	2002 / 2003
Celera WGA Assembler	OLC	Myers, G. *et al.*	2004 / 2008
Minimus (AMOS)	OLC	Sommer, D.D. *et al.*	2007
Newbler	OLC	454/Roche	2009
EDENA	OLC	Hernandez D., *et al.*	2008
FM-assebler	OLC	Durbin R., et al	2010
MIRA, miraEST	OLC	Chevreux, B.	1998 / 2008
SGA	OLC	Simpson. J.T. *et al.*	2010
PE-Assembler	OLC	Pramila, N.A. *et al.*	2010
TIGR	Greedy	TIGR	1995 / 2003
Phusion	Greedy	Mullikin JC, *et al.*	2003
Phrap	Greedy	Green, P.	2002 / 2008
CAP3, PCAP	Greedy	Huang, X. *et al.*	1999 / 2005
SHARCGS	Prefix-Tree	Dohm *et al.*	2007
SSAKE	Prefix-Tree	Warren, R. *et al.*	2007
VCAKE	Prefix-Tree	Jeck, W. *et al.*	2007
QSRA	Prefix-Tree	Douglas W. *et al.*	2009
Euler	DBG	Pevzner, P. *et al.*	2001 / 2006
Euler-SR	DBG	Chaisson, MJ. *et al.*	2008
Velvet	DBG	Zerbino, D. *et al.*	2007 / 2009
ALLPATHS	DBG	Butler, J. *et al.*	2008
Ray	DBG	Boisvert, S *et al.*	2010
ABySS	DBG	Simpson, J. *et al.*	2008 / 2009
SOAPdenovo	DBG	Ruiqiang Li, *et al.*	2009
Meraculous	DBG	Chapman, J. *et al.*	2011
SUTTA	B&B	Narzisi G., *et al.*	2009/2010
Sequencher	-	Gene Codes Corporation	2007
SeqMan NGen	-	DNASTAR	2008
Staden gap4 package	-	Staden *et al.*	1991 / 2008
NextGENe	-	Softgenetics	2008
CLC Genomics Workbench	-	CLC bio	2008 / 2009
CodonCode Aligner	-	CodonCode Corporation	2003 / 2009

Table 4.1: The first four categories are divided on the base of the algorithm used (Overlap Layout Consensus, Greedy, Prefix-Tree, and de Bruijn Graph). Prefix-Tree algorithm are greedy assembler. SUTTA deserves a category thus is the only assembler employing a Branch-and-Bound approach. The last category is a non complete list of proprietary solutions.

Graph. As a matter of fact, a full all-against-all comparison is always avoided and fast and efficient approximated algorithms have been proposed to speed-up this phase. The Overlap/String Graph is then used to generate a Layout from which, during the Consesus phase, a Multiple Sequence Alignment (MSA) is performed and the output is generated. Even though tools based on this approach have demonstrated their capabilities with Sanger Sequencing data, they are not easily extandable to handle short reads for two main reasons. First, the use of short reads forces the minimum overlap between reads to be so small that the number of overlaps occurring by chance becomes too high. Second, as a consequence of the extremely high amount of reads, the Overlap phase becomes an overwhelming computational bottleneck. However, encouraged by the increasing read length some OLC based assembler able to scale on NGS-data appeared (SGA [165]).

DBG assemblers are the most successful type of assemblers for short read sequences. A de Bruijn Graphs is a graph in which the nodes are k-mers and there is an edge connecting two nodes a and b if and only if the (k-1)-suffix of the k-mer a is identical to the $(k-1)$-prefix of the k-mer b. Given a set of reads, the de Bruijn Graph is constructed by dividing all the reads in all possible k-mers, associating k-mers to nodes, and then connecting nodes. This construction has the double advantage that no overlap has to be computed and the amount of memory needed is proportional to the number of distinct k-mers and not to the number of distinct reads (the number of distinct k-mers belonging to the reads directly depends on the number of distinct k-mers of the genome being sequenced).

4.2.1 Greedy Assemblers: Seed-and-Extend

Greedy assemblers repeatedly pick up a *seed* (it can be either a read or a previously assembled contig) and *extend* it using other reads. This procedure is done through the computation of all, or almost all, the overlaps between the seed's tips and all the available reads. The reads used for the extension are those with the highest alignment score. It is clear that the Seed-and-Extend assemblers' key feature is the capability to quickly compute all the alignment scores. Usually this goal is achieved using hashing schemata or quick look-up tables to obtain at least all the perfect matches. Most of the solutions described in the following sections are variants of the general schemata. Often solutions differ from one another only for the implemented heuristic. The main drawback of Seed-and-Extend-based assemblers is their incapability to distinguish and correctly assemble repetitive regions. Each seed is independent from the others and therefore no global information is available. Despite this problem, several seed-and-extend assemblers have been proposed. A common heuristic is to use such solutions to obtain long reads (*i.e.*, Sanger-like sequences) and use the sequences produced in this way as input to a Sanger-based assembler.

SSAKE.

SSAKE [182] was the first short-read assembler proposed. It is designed for Illumina reads but, more recently, it has been adapted to use also long Sanger reads. SSAKE first step is the population/creation of an hash table. Such hash table has as keys the input sequences, and as values the multiplicity of each sequence. At the same time a tree is used to memorize the first eleven bases of each read. Once all sequences are read and stored, the reads are sorted by decreasing number of occurrences. This information is used to understand the read coverage and to identify reads containing low copy sequences that are candidate to contain errors. Each unassembled read is used to start an assembly. SSAKE uses the prefix tree to progressively compute perfect alignments of length k.

The extension phase halts when there are no more reads to extend or when a k-mer matches the 5' end of more than one sequence read. This is done with the aim of minimizing sequence misassemblies. A more flexible halting strategy consists in stopping the extension when the retrieved k-mer is smaller than a user-set minimum word length. Recently [181] SSAKE has been extended to use paired read information, Sanger reads and imperfectly matching reads.

SHARCGS.

SHARCGS [35] is a DNA assembly program designed for *de novo* assembly of $25-40$-length input fragments and deep sequence coverage. The assembly strategy is similar to the one described for SSAKE with two more features: a pre- and a post- processing phase.

In the pre-processing phase, SHARCGS discards reads that are likely to contain errors. These reads are identified by requiring a minimum number of exact matches in other reads and requiring a minimum quality value, if available. SHARCGS performs three times this filtering phase, requiring each time a different stringency setting. This strategy allows to generate three different filtered sets. Then in a SSAKE-like way it assembles every set independently. The post-processing phase consists in merging the contigs obtained by running the algorithm with weak, medium, and strong filter parameters settings.

VCAKE.

The aim of VCAKE [70] is to assemble millions of small nucleotide reads even in the presence of sequencing errors. The main improvement proposed by VCAKE is the ability to deal with imperfect matches during contig extension. In particular VCAKE uses the same prefix tree implemented by SSAKE but it allows one mismatch during the extension phase.

VCAKE has been further used in two hybrid pipelines: in [149] VCAKE and Newbler are used together for assembling a mixture of Illumina and 454 reads, while in [47] VCAKE is combined with Newbler and Celera Assembler.

QSRA.

QSRA [20] is built directly upon the SSAKE algorithm. QSRA creates an hash table and a prefix-trie. Each hash entry stores a pair composed by the actual DNA sequence and the number of occurrences of the read. The prefix-trie contains the unassembled reads as well as their reverse complements, all indexed by the first 11 bases. In a SSAKE/VCAKE similar fashion, QSRA starts the extension phase finding all the reads which exactly match the end of the seed (the "growing" contig) for at least u bases (where u is a user-defined parameter) using the prefix-trie. If the number of matches is less than a user-defined threshold, but the quality values are available, QSRA will extend the growing contig as long as a minimum user-defined q-value score m is met.

SHORTY.

SHORTY [58] is a *de novo* assembler targeted to the assembling sequences produced by Solid sequencers. It takes in input deep coverage of solid reads ($100\times$) and a small set of seeds sequences. These seeds can be obtained from a set of Sanger sequences or even by assembling with another short-read assembler the short reads.

All the reads are stored in a compact trie that allows quick access and fast searches. After the trie construction, all the seeds are processed one-by-one. For each seed we extract from the set of reads those belonging to the seed together with the paired reads that are outside the seed sequence. From the seed sequence and from the overlap information coming from the reads it is possible to generate contigs. SHORTY reiterates these steps as long as it is possible to extend the contigs. At this point all the contigs generated from seed are considered together for further processing to generate larger contigs. The last step uses again the paired read information to build scaffolds.

4.2.2 Overlap-Layout-Consensus Based Assemblers

Assemblers based on the Overlap-Layout-Consensus (OLC) approach have to compute all the overlaps among the reads and use this information together with the coverage depth to reconstruct the original sequence. If available, the assembler can use the paired read information.

OLC assemblers build an overlap graph [124]. The first mandatory step is the computation of the overlaps between pairs of reads. This step involves an all-against-all pairwise read comparison. Usually, for efficiency issues, programs pre-compute the k-mer content among all reads and compute only the overlaps between pairs of reads sharing a predefined number of k-mers. This kind of overlap computation is particularly sensitive to three parameters: the k-mer size, the minimum overlap length and the percentage of identity required for an overlap. Larger parameter values are likely to produce more reliable but shorter contigs and at an higher computational cost. On the other hand, lower values can greatly reduce the computational needs, but at the price of producing too many mis-assemblies.

Once the reads' overlaps are computed, the obtained overlap graph is usually simplified by identifying problematic sub-graphs in order to reduce the complexity. This simplification step allows to create an approximate read layout.

Finally, the last step consists in performing Multiple Sequence Alignments (MSA) to obtain a precise layout and the consensus sequence. Again, this step is approximated by progressive pairwise alignments between overlapping reads, since no efficient method to compute optimal MSA is known [180].

OLC assemblers have been the unquestioned assemblers for more than ten years. The advent of Next Generation Sequencing machines characterized by new error schemata and by ultra-short reads length seemed to declare the end of an era. Nevertheless, some assemblers based on OLC have been proposed even to assemble short Illumina reads. Moreover, encouraged by longer reads length, some valuable assemblers designed for 454 reads have appeared.

EDENA.

EDENA (Exact *de novo* Assembler) [54] is an assembler conceived to process the millions of very short reads produced by the Illumina Genome Analyzer. In order to improve the assembly of very short sequences it adds exact matching and detection of spurious reads. EDENA utilizes only exact matches for two main reasons: (i) allowing approximate matches increases the number of spurious overlaps, and (ii) approximate matching is dramatically slower than exact matching. EDENA starts by removing from the dataset all redundant reads and all those containing ambiguous characters and, meanwhile, it creates a prefix tree. After all overlaps of minimal size h are computed, the overlap graph is constructed. EDENA constructs a suffix-array to compute all overlaps among reads. All the overlaps are loaded in a bi-directed graph structure, where for each read r_i there is a vertex v_i and two vertices v_i and v_j are connected by a bi-directed edge if r_i and r_j overlap. Each edge is labelled with the length of the corresponding overlap. Obviously, as in all OLC assemblers, the minimum overlap size is a crucial parameter for the assembly success.

The produced graph contains, in general, a large amount of branching paths hindering the construction of long contigs. EDENA executes a graph-cleaning step by removing *transitive* edges (an edge $v_1 \rightarrow v_3$ in presence of a path $v_1 \rightarrow v_2 \rightarrow v_3$ is dubbed transitive and can be removed), short dead-end paths (a branching path of short length), and bubbles (two paths starting and ending on the same node and containing similar sequences). Short-dead-ends and bubbles are a consequence of sequencing errors (reads with errors in the last bases) and of clonal polymorphism (in particular SNPs), respectively.

Once the cleaning phase is terminated the contigs can be generated exploring the reduced graph. Edena published version [54] is not able to scale on the large datasets produced by state of the art sequencers. However, a new version (EDENA V0.3) is under development aiming at overcoming this obstacle.

SGA.

OLC assemblers do not scale well to large datasets composed by short reads, and hence EDENA cannot be used in practical scenarios. Others OLC assemblers are suited for 454-based projects characterized by longer (and more expensive) reads and lower coverages (this second feature is partially a consequence of the first one).

As a matter of facts, the computation of (almost) all the possible overlaps is carried out by indexing reads. The large number of reads does not allow the use of standard data structures like suffix-trees and suffix-arrays. In [165] this problem was overcome by SGA, an OLC-based assembler suited for Illumina reads that uses the FM-index [40]. In particular, SGA is able to build the overlap graph in time proportional to $O(N)$, with N the total length of all the reads.

SGA builds the FM-index for the set of reads $\mathcal{R}$ using a variant of Ko-Aluru [81] algorithm for the suffix-array construction. The FM-index of the reversed reads is also computed. At this point, using the *backward search* algorithm [40] the overlaps between reads' tips can be easily and quickly computed. Moreover, SGA can directly compute only non-transitive edges and therefore directly compute the simplified and reduced version of the string graph. This last feature allows to boost SGA's performances.

Even though the large amount of details and technicalities presented in [165], SGA is the proof of how advanced data structures and algorithms proposed and adopted for string alignment are of primary importance in *de novo* assembly.

Newbler.

Newbler [111] is the proprietary *de novo* assembler of 454 Life Science Corporation [111]. Newbler assembles the reads in "flow space". In this format, a read is represented as a sequence of signals. The signal's strength is proportional to the number of direct repeats of that nucleotide at that position in the read. This choice is done to avoid the introduction of errors with an early base space conversion. This conversion can be postponed until the computation of the MSA, when multiple reads can help in identify and solve errors and/or ambiguities.

Newbler implements a double OLC strategy and is divided into three modules: Overlapper, Unitigger and Multialigner.

The Overlapper performs a complete all-against-all fragment comparison so that it identifies all possible overlaps between fragments. To assess the similarity between reads, it directly compares the *flowgrams* of each pair of reads. With the goal of increase efficiency, Overlapper uses a hashing indexing method to quickly identify fragments that might be considered as potential overlap candidates.

Based on the overlaps computed by the Overlapper module, Unitigger groups the reads into *unitigs*. These unitigs are a sort of trusted contigs, uncontested by reads external to the unitig. Unitigs are constructed from consistent chains of maximal depth overlaps. The unitigs serve as preliminary, high-confidence, conservative contigs that seed the rest of the assembly pipeline.

Finally, Multialigner takes all the reads composing the unitigs and aligns all the read signals to obtain the real unitig sequence.

The unitigs generated this way are sent through a contig optimization process composed by three steps. In the first one, an all-against-all comparison is performed and overlapping unitigs are joint. After this comparison, performed in nucleotide space, Newbler tries to identify repeat regions boundaries based on where contig sequences diverge from a common region. Contigs are broken at those boundaries. In the second optimization step, the contigs produced during the first step are used for a "restitching" operation: reads spanning two contig ends are used to join these contigs. The third and final optimization step is a quality-check step performed by aligning all the reads against the contigs and discarding contigs with low coverage.

As last step, the consensus is recomputed, using the flowspace. This choice allows to gain more precision and accuracy.

CABOG.

CABOG [118] assembler is a revised version of Celera Assembler [125] designed for 454 reads. CABOG, like Newbler, parses the native SFF files produced by 454 machines. Like Celera Assembler it is divided into independent modules.

The *Overlapped-based trimming* phase trims reads and identifies possible spurs and chimers (reads that join discontinuous genomic loci) by computing local alignments for all pairs of reads.

In the *anchors and overlaps* phase, CABOG uses exact-match seeds to detect possibly overlapping reads and builds the overlap graph. During this phase reads formed by k-mers occurring in single copy or with a number of copies larger than a precomputed threshold are not used.

Using the computed alignments, CABOG can build the overlapping graph G. In this phase a drastic and lossy data reduction is performed. The graph G is reduced to the *Best Overlapping Graph* (BOG) by keeping for each node (*i.e.*, each read) only the best edge (*i.e.*, the longest overlap). Moreover, all cycles are eliminated by deleting a randomly chosen edge. BOG is therefore acyclic, but paths in BOG can still converge due to overlaps that are not mutually best for both reads involved.

CABOG sorts reads by score, where the score of a read is defined as the number of other reads reachable from it in the BOG. Starting from the highest scoring reads, it follows the paths in the BOG to construct *unitigs* (trusted contigs). Unitigs spanning intersections in the BOG are broken. Further unitigs splitting is performed using paired read information.

Once unitigs are computed and simplified, the *contig*, *scaffold*, and *consensus* steps of the Celera Assembler are performed.

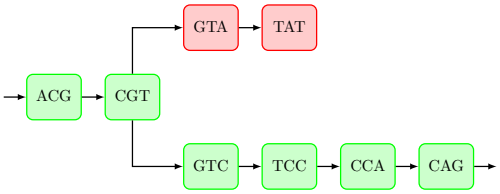
4.2.3 De Bruijn Graph Based Assemblers

De Bruijn graphs (DBGs) owe their success to their embedded capability of representing the myriad of reads produced by NGS in a reasonable amount of space.

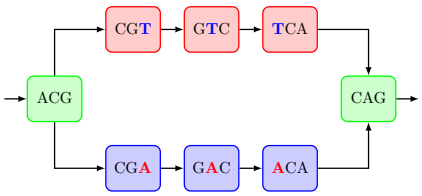
DBG approach is commonly known also under the name of *k-mer graph* or *Eulerian* approach [140]. In a de Bruijn graph, nodes represents k-mers. Two nodes are connected by an oriented edge if the k-mers they represent overlap for $k-1$ characters. The de Bruijn Graph approach starts

with the counter-intuitive step of reducing short reads in even shorter sequences (*i.e.*, k-mers). In this way reads are represented by paths in the graph. The main advantage of this technique is that every k-mer is represented only once despite the number of its occurrences. In an ideal setting with error-free reads and uniform coverage, the k-mer graph would be a de Bruijn graph and an assembly would be represented by an Eulerian path, *i.e.*, a path that traverses each edge exactly once. It is clear that in real-life situations, where the reads contain errors and coverage is not uniform, the assembly procedure is slightly more complicated.

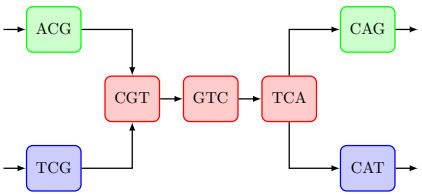
In this context, the assembly is a byproduct of the graph construction. Although many different implementations and heuristics have been proposed and implemented, the graph construction relies on a hash table that tracks all the k-mers represented in the reads. While in theory the memory used does not depend on the input size but only by the different number of k-mers, in practical situations (mammalian and plant genome sequencing projects) the amount of memory required is still the main bottleneck. This problem seems overcame by distributed assemblers that recently have been proposed [166, 154].



(a) Spurs: if a read contains a single error a branching path is open



(b) Bulge/Bubble: A Single Nucleotide Polymorphism causes a branching path that will soon converge.



(c) Fray rope: a repeat causes two or more paths to converge in a single node and proceed together until the end of the repeat.

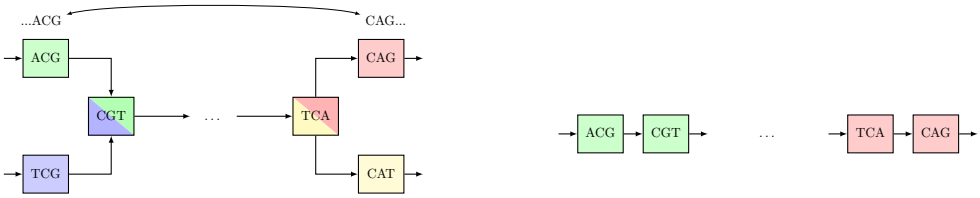
Figure 4.2: de Bruijn graph Errors. The three main sources of errors in a de Bruijn graph.

Several factors analysed in [119] complicate the application of k-mer ghraphs to sequence assembly. First of all, DNA is double stranded. Different implementations have been proposed to handle reverse-forward overlaps. A second problem is the identification and reconstruction of complex repeated structures present in real genomes. Repeats longer than k introduce complexity in the graph and, as a direct consequence, confuse the assembly task. Repeats collapse in a single path inside the graph, with the consequence that many paths can converge inside a repeat and then diverge (see Figure 4.2(c)). Assemblers usually use reads to understand what the right path is (they search for read-coherent paths) and in a similar way they use paired read information. The last problem is represented by sequencing errors. DBG-based assemblers use several strategies to deal with errors. Some assemblers pre-process the reads to remove errors by discarding/correcting reads containing low-quality bases or lowly represented k-mers. Other times they discard paths in the graph not supported by an high number of reads. Another technique consists in converting paths into sequences and using sequence alignment to collapse nearly-identical paths.

EULER-SR.

EULER was the first assembler based on de Bruijng Graph approach. The first version of EULER was suited for Sanger reads [141, 142, 140]. Despite the advantages of the method, it had only a

(a) Repeat resolution via read tracking: following the reads paths small repetitions can be easily resolved



(b) Repeat resolution via paired reads: longer repeats can be resolved through paired reads

Figure 4.3: De Bruijn Graph: Read Tracking. Two commonly used techniques to resolve repeats in De-Bruijn Graphs.

marginal success. With the advent of next generation sequencing and the need of processing large amounts of reads, EULER was adapted first to handle 454 GS20 reads [139] and, soon after, also the Illumina reads [133, 138]

The first step implemented by EULER is a correction phase. EULER identifies erroneous base calls looking for low-frequency k-mers. This filter is called *Spectral Alignment* [142]. A *Spectrum* T is a collection of l-tuples. A string s is called a T*-string* if all its l-tuples belong to T. Given a collection of strings $S = \{s_1, .., s_n\}$ from a sequencing project and an integer l, the spectrum of S is S_l (the set of all l-tuples from the reads $s_1..s_n$ and $\overline{s_1}$,..,$\overline{s_n}$, where $\overline{s_i}$ is the reverse complement read of s_i). Given S, Δ and l, EULER introduces up to Δ corrections in each read in S to minimize $|S_l|$. The idea is that reads that contain errors are composed by low-frequency k-mers. Instead of simply discarding these reads, EULER tries to correct them.

By reducing the total number of k-mers, the correction lowers the number of nodes in the graph and hence its complexity. This correction step can mask true polymorphism or delete valid k-mers belonging to low-coverage areas. OLC assemblers have an analogous base call correction step that uses overlaps rather than k-mers. EULER spectral alignment is able to cope only with mismatches and not with insertions or deletions.

EULER builds the k-mer graph or de Bruijn Graph from the set of filtered and corrected reads. The main drawback of this approach is that, since the basic units on which the graph is defined are k-mers rather than reads, the information about the original reads may become difficult to retrieve. EULER overcomes this problem by threading the reads through the graph. This implementation allows to easily resolve paths in presence of repeats that are spanned by a read as shown in Figure 4.3(a). By tracking reads on the graph, we can resolve repeats of size between k and read length.

In order to resolve repeats longer than the read length, EULER extends the described approach to paired reads. Paired ends spanning a repeat can be used to find paths containing repeats: as showed by Figure 4.3(b) a paired read allows to follow a path entering and leaving repeat smaller than the insert size. In other words, EULER considers a paired read as a read with some missing characters in the middle. Moreover, EULER can use the insert size information to better distinguish between paths.

After this threading phase, EULER performs some graph simplification at regions with low coverage or high coverage. First, EULER removes spurs, *i.e.*, short branching paths that are likely due to sequencing errors. If the quality information is available EULER uses it to improve the assembly. Many of the platforms produce reads with low quality values at the 3' ends. For this reason, EULER trusts more reads prefixes than reads suffixes.

The k-mer size parameter is a critical parameter when working with de Bruijn graphs. A small

k reduces the number of nodes and can result in an over-simplification of the graph (consider the case of k=1). On the other hand, a large k can yield a graph with too many nodes that is likely to produce a fragmented assembly (consider the case of k equal to the reads length). EULER solves this problem by constructing two k-mer graphs with different values of k. EULER detects the edges present in the smaller-k graph but missing in the larger-k graph. The idea is then to use a large k to resolve the gaps, and later to fill the gaps by using the edges coming from the smaller-k graph.

VELVET.

VELVET [188, 187] is probably the most cited *de novo* assembler for short reads. It has been used in several projects, including the apple scab genome project [148].

The first step of the algorithm consists in the de Bruijn graph construction. VELVET starts by hashing all the reads according to a user predefined k-mer length. The value of k is bounded from above by the length of the reads and it must be odd (so that a k-mer cannot be its own reverse complement). This way VELVET builds a "roadmap", *i.e.*, the information thanks to which each read can be rewritten as a set of k-mers combined with overlaps of previously scanned reads. A second database is created with the opposite information. It records, for each read, which of its original k-mers are overlapped by subsequent reads. The ordered set of original k-mers of that read is cut each time an overlap with another read begins or ends. For each uninterrupted sequence of original k-mers, a node is created. Once the graph is constructed, the reads can be traced through it by using the roadmap information.

VELVET does not make any read filtering step. However, it performs several graph simplification steps aiming at removing from the graph the paths that are likely to have been introduced by sequencing errors. In particular, VELVET identifies two common graph layouts that are likely to be a consequence of sequencing errors: *tips* and *bubbles*. A *tip* (sometimes called *spur*) is a chain of nodes disconnected on one end (see Fig. 4.2(a)) while *bubbles* (sometimes called *bulges*) are paths starting and ending on the same nodes and containing similar sequences (see Fig. 4.2(b)). Tips are usually a consequence of reads with errors in the last bases, while bubbles can be caused either by SNPs or by sequencing errors.

VELVET performs several graph simplification steps to reduce the impact of such structures and to improve the final result. The first simplification step consists in merging all the *chains* (*i.e.*, sequences of nodes with only one ingoing edge and only one outgoing edge). Subsequently, VELVET identifies tips shorter than $2k$ and removes them. The removal of these structures has only a local effect and does not cause disconnection of the graph. "Bubbles" are removed using the Tour Bus algorithm [188]: VELVET detects bubbles through a Dijkstra-like breadth-first search.

A fourth graph simplification aims at removing erroneous connection in the graph (*i.e.*, edges that connect sequences belonging to different genomic loci). Erroneous graph connections are not associated with easily recognizable sub-graphs and therefore are more difficult to detect. VELVET uses a coverage cutoff to remove connections with a coverage depth under a user defined threshold. This threshold together with the k-mer length is one of the most important VELVET's parameters.

VELVET uses paired information to resolve complex and long repeats. In [188], the scaffolding phase is done through the "Breadcrumb" algorithm. This algorithm was inspired by SHORTY [58]. It localizes on the graph simple paths (contigs) connected by paired reads. Using the long contigs as anchors, VELVET tries to fill the gap between them with short contigs. More recent versions of VELVET [187] use a more sophisticated algorithm, called *Pebble*. Pebble starts by identifying unique nodes (*i.e.*, contigs) with the only help of the contig coverage values, using a statistic derived from the A-statistic. Pebble tries to connect the unique nodes identified previously, with the help of paired-end information. For each unique node, it estimates the distances from that node by exploiting the given insert length distribution. The complete set of estimated inter-node distances is called the *primary scaffold*. At this point, the algorithm tries to close the gap between contigs by finding a path that is consistent with the layout. In [187] also the "Rock Band" algorithm is introduced. This module exploits long (*i.e.*, Sanger-like) reads to connect the nodes of the graph after the error-correction phase. The main idea is that if all the long reads which exit from a node

go consistently to another unique node and vice-versa, then the two nodes can be safely merged.

ALLPATHS.

ALLPATHS [22, 108, 45] is a whole-genome shotgun assembler that can generate high-quality assemblies from short reads. Assemblies are presented in a graph form that retains ambiguities, such as those arising from polymorphisms, thereby providing information that has been absent from previous genome assemblies.

ALLPATHS uses a read-correcting pre-processor similar to EULER's spectral alignment. ALLPATHS identifies putatively correct (*trusted*) k-mers in the reads based on quality scores and on the k-mers frequencies. These trusted k-mers are used to correct the entire set of reads.

A second pre-processing step is implemented by ALLPATHS to create *unipaths* (*i.e.*, maximal unbranched sequences in the k-mer graph). Unipaths are constructed by building a compact searchable data structure by indexing the k-mers so that the computation of all the overlaps is avoided.

The first graph operation is the spur erosion, named *unitig graph shaving*. This operation aims at removing short branching paths that are usually caused by sequencing errors. Once this operation terminates, a subset of unipaths are elected to be *seeds*. *Seeds* are the unipaths around which ALLPATHS constructs the assembly. A seed is a long unipath characterized by a low copy number (ideally one). The idea is to extend the seed's neighbourhood and to join more than one unipath together with the help of paired read information.

ALLPATHS partitions the graph to resolve genomic repeats by assembling regions that are locally non-repetitive. Partitions are assembled separately and in parallel. At the end, ALLPATHS *glues* the local graphs by iteratively joining sub-graphs that have long end-to-end overlapping stretches. Allpaths heuristically removes spurs, small disconnected components, and paths not spanned by paired reads.

The latest available version of ALLPATHS [45] is designed for Illumina reads of length 100. In particular, ALLPATHS requires two different kinds of paired reads: *fragment library* and *jumping library*. A *fragment library* is a library with a short insert separation, less than twice the read length, so that the reads may overlap (*e.g.*, 100 bp Illumina reads taken from 180 bp inserts). A *jumping library* has a longer separation, typically in the 3-10 Kbp range. Additionally, ALLPATHS also supports *long jumping libraries*. A jumping library is considered to be long if the insert size is larger than 20 Kbp. These libraries are optional and used only to improve scaffolding in mammalian-sized genomes. Typically, long jump coverage of less than $1\times$ is sufficient to significantly improve scaffolding.

The latest available ALLPATHS version [45] is able to accurately assemble the human genome, and to achieve a result close to Sanger sequencing assembly.

ABySS

ABySS [166] is a *de novo* sequence assembler designed for short reads. The single-processor version is useful for assembling genomes up to 40-50 Mb in size. The parallel version is implemented using MPI communication messages [50] and is capable of assembling larger genomes.

The assembly is performed in two major steps. First, without using the paired-end information, contigs are extended until either they cannot be unambiguously extended or they come to a blunt end due to a lack of coverage. In the second step, the paired-end information is used to resolve ambiguities and to merge contigs. In the third stage, mate-pair information is used to extend contigs by resolving ambiguities in contig overlaps.

ABySS constructs a de Bruijn graph in a way similar to VELVET and EULER. Like both algorithms already explained ABySS performs an error correction phase on the graph. In order to handle read errors, ABySS implements a strategy correspondig to a combination of EDENA, VELVET and EULER-SR algorithms.

The advantage of ABySS is its capability of assembling large genomes thanks to the MPI parallel version. The construction of the graph can be performed in a distributed way. Another,

more recent, assembler able to exploit several nodes in a cluster is RAY [17].

4.2.4 Branch-And-Bound Approach

SUTTA [130] differs from all other assemblers for its assembling strategy: it dispenses with the idea of limiting the solutions to just the approximated ones due to the NP intractability of AP, and instead it favors an approach that could potentially lead to an exhaustive (exponential-time) search of all possible layouts.

In order to limit the possibly exponential search space, SUTTA relies on a constrained search able to prune implausible layouts. This Branch-and-Bound (B&B) strategy is based on a set of score functions that combine different structural properties. SUTTA strategy cannot be associated to none of the previous three strategies, therefore, as SUTTA's Authors suggested, we inserted it in the new category of B&B based assemblers.

In the De Bruijn flattened NGS landscape, B&B represents a different prospective to Assembly Problem. Instead of focus the attention on inherently approximate heuristics this strategy tries to comprehensively solve the problem.

SUTTA

SUTTA solves Assembly Problem providing a layout consistent with a particular set of properties (*oracles*). Oracles must satisfy overlap conditions, mate pair constraint and when available optical maps constraints. SUTTA assembles each contig independently one after the other (in a way similar to greedy assemblers) using a Branch-and-Bound schema.

B&B basic idea is to extensively explore the complete solutions space. This possibly exponential exploration is limited only to optimal solutions that are reached through the use of *well chosen* score functions. SUTTA starts by considering one read per time: it builds a double-tree (D-tree) to compute a set of possible layouts. The D-tree is subsequently use to compute the best layout and extract the contig. The algorithm ends when all reads have been extended or have been used to build a contig.

The potentially exponential size of the D-tree is controlled by exploiting certain specific structures of the assembly problem that permit a quick pruning of many redundant and uninformative branches of the tree. In particular the pruning is implemented by discarding transitively inferable paths, paths not supported by paired reads, or paths with extremely low read coverage.

4.3 Assembly Validation

Assembly Validation is the task of evaluating and judging assemblers output. Once the assembled sequence is available it is extremely interesting to know and to gauge the sequence correctness and eventually to discover and hopefully correct misassemblies. Moreover, different assemblers use different heuristics and strategies, therefore it is also interesting evaluate assemblers' results, to select and to use only the best ones. In a way similar to the multitude of assemblers, there are several ways to evaluate assemblers and assemblies.

For more than 20 years, Sanger sequencing has been the unquestioned method of choice in almost all the large genome projects. Deluged by high coverage data, but hampered by their poor quality and short length, many new assemblers for short reads have resorted to filtering the reads into compressed graph structures (usually a de Bruijn graph) and additional heuristics for error correction and read-culling (*e.g.*, dead-end elimination, p-bubble detection, etc.) in order to handle such short sequences as best as possible. Several *de novo* projects have been launched with some success [96]. It is now commonly accepted that short reads make the assembly problem significantly harder [127], yielding final genome-assemblies of dubious value. To make matters worse, NGS data are often characterized by new and hitherto-unknown error structures, which can easily change within a year.

Assemblers and assemblies validation is becoming everyday more and more pressing. A recent investigation [164] showed that in the published and revised human genome [85] an average 10%

of assembled fragments were assigned the wrong orientation and 15% of fragments were placed in a wrong order. The draft sequence of the Human Genome [85], which was released in 2001, took several large teams more than five years to finish and validate (but only at a genotypic level). NGS technologies even worsen the situation, with tens of projects left at draft level. Despite the time and financial resources involved in the finishing of the Human Genome, it must be stressed that is was largely a manual process. In contrast, most of the current genome projects lack both time and money, forcing the developers to simply leave the assembly at a draft level (with many gaps and unresolved phasings). Alkan in [6] criticised two of the majors NGS achievements: the assembly of the Han Chinese and Yoruban individuals [100] both sequenced with Illumina reads. Alkan identified 420.2 Mbp of missing repeat sequences from the Yoruban assembly, and estimated that in both assemblies almost 16% of the genome was missing.

Even though these clear problems, there is a lack of standard procedures and methods to validate and evaluate assemblies. Several projects have been initiated to explore the parameter space of the assembly problem, in particular in the context of short read sequencing [143, 6, 103, 130, 189]. These analyses are urged by projects like *assemblathon* [37] (now at its second edition): assemblathon goal is to assess assemblers performances on common data sets. In its first edition the competition was performed on a simulated dataset, while the second and still running one was done on three real NGS datasets.

Therefore, it is of primary importance to gauge and to evaluate results achieved with NGS-based assemblers. These assemblers are evolving at a fast pace together with Second (and Third) Generation Sequencing Technologies. All ongoing assembly projects are based on NGS-reads, in particular on Illumina reads as a consequence of the good trade-off between quantity (*i.e.*, instrument throughput) and quality (*i.e.*, read length).

4.3.1 Standard Validation Metrics/Features

A commonly accepted way to validate and gauge assemblies is based on a *plethora* of *standard validation metrics*. We can identify four main groups: length-base statistics, long range information (LRI) based statistics, reference-based statistics, and simulation-based statistics.

Length-based statistics take into account only the size of the assembler output. The first, and obvious one, is the *assembly-length*. Generally, the genome length is known, therefore a correct assembly should have a length similar to the sequenced organisms. In a similar way, the *number of contigs* is used to gauge assemblies: closer is the number of contigs or scaffolds to the number of chromosome more connected the assembly is. Similarly, we can compute the *mean contig/scaffold length*: a longer mean contig/scaffold length suggests an high connected assembly. The queen (or the peasant as we will see in Chapter 5) of length-based metrics is the *N50*. N50 represents the size N such that 50% of the genome is contained in contigs of size N or greater[1] (see Figure 4.4). $N50$, in principle, should give an idea of the connectivity level of the assembly. A large $N50$ means that with a small number of large contigs we are able to cover more than half of the genome. This statistic has been used in almost all the assembly projects as a quality proof, however, a large $N50$ is not connected to assembly correctness. In a similar way one can define the $N10$, $N20$ up to $N90$.

The main problem of *all* length-based statistics is the fact that these statistics are not linked to assembly correctness and emphasize only length: an assembler that eagerly merges together contigs can produce assemblies characterized by a large $N50$ and by few long contigs. However, these long contigs are of no use if they contain too many misassemblies. Nevertheless, length-based statistics are the basic, and some times the only, instrument used to judge assemblers performances, especially when comparing different assemblers [188, 150].

A more reasonable way to assess assembly correctness is to use long range information independent from the assembly (*i.e.*, not used in the assembly process). Second Generation Technologies are able to produce *mate pairs*, that are pair of reads at a mean distance of $2 - 8$ Kbp. Mate

[1] In literature, some times $N50$ is used to indicate the largest contig such that the sum of all the contigs larger than it is at least half of the genome length, while $L50$ indicates the size of the $N50$ contig. However, in our experience the definition provided in the text is the most widely used.

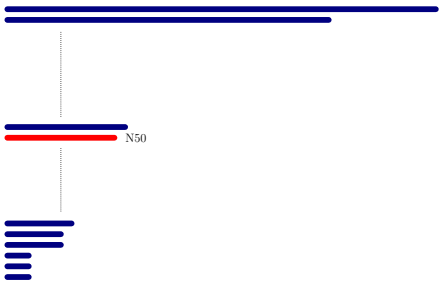


Figure 4.4: N50 represents the size N such that 50% of the genome is contained in contigs of size N or greater.

pairs are of primary importance to produce scaffolds, however if they are not provided to the assembler they can be used to gauge the assembly correctness: pairs should map on the assembly at the estimated distance and with the right orientation (this one depending on the sequencing technology being used). However, such an approach has several problems: (*i*) mate pair insert size variance can be large, making difficult to gauge correctness; (*ii*) the approach is limited by the insert size; (*iii*) mate pairs belonging to repetitive regions can be wrongly aligned leading to bad validation. Some of these problems, in particular the insert size limitation, can be overcome using Fosmid ends and BAC ends, however, the monetary cost of long range information data is an order of magnitude higher than mate pairs. Other two commonly used LRI-methods are *physical maps* [49] and *optical maps* [163]. Both allow to obtain the relative locations of different genes and other DNA sequences of interest in the genome. With this information in hand one can estimate the correctness of the assembly.

Everything would be much easier if the genome to be assembled is already available. This simple though is behind the reference-based statistics. At a first sight it seems contradictory to sequence and assemble and organism that has been already sequenced and assembled but there are at least two exceptions: assemblers comparison and closely related organisms. In the former case, we re-assemble an already available and finished genome in order to assess and evaluate the performances of different assemblers. The latter case arises when one sequence a new organism but there is an already assembled genome belonging to a different species that is closely related to the new one.

In both cases, the already available genome can be used as a *reference* to evaluate the correctness of the assembly. Also in this case several metrics can be used: percentage of correctly assembled contigs (*i.e.*, contigs that correctly align against the reference), number or percentage of errors (*i.e.*, wrongly aligned contigs), percentage of reconstructed genes and so on. The main drawback of such statistics is the fact that in general it is difficult to find a closely related sequence. Moreover, it is not clear if a tool that returns good results on a certain dataset will give the same results on an utterly different dataset. Reference-based metrics have been used to describe results on two human individuals in [100] that have been recently deeply criticized [6].

Simulation-based statistics are used to attain a result close to the reference-based statistics without the need to resequence and already assembled genome (hence, without the need to spend money). This is typical of assembler evaluation (*e.g.*, *assemblathon 1*). By mapping back contigs to the reference one can easily estimate the number of mis-assembled contigs and even the mis-assembly's type (*e.g.*, duplications, insertions). However, this method is biased by the read simulator algorithm: suitably chosen (most likely, non-realistic) simulation could produce as optimistic (or pessimistic) results desired.

4.3.2 Assembly Forensics and Feature Response Curve

Evaluation instruments are of primary importance for *de novo* assembly in order to critically assess assembler performances and to gauge results. Although standard metrics are widely used, such metrics are affected by several problems: some of them stress only length, some other are based on the non always realistic hypothesis of the availability of a reference genome belonging to a closely related genome. As noticed in [143, 130] there is particular need of new validation and evaluation methods able to capture the correctness and quality of an assembly without the need to generate further expensive data.

Phillippy and colleagues in [143] proposed a more intelligent approach to better inform the overall assembly quality and correctness based on the notion of *layout*. Given a set of reads $\mathcal{R} = \{r_1, r_2, .., r_n\}$ sequenced by an organism G a layout for the reads belonging to $\mathcal{R}$ is a permutation $r'_1, r'_2, .., r'_n$ such that for each $i \in \{0..n-1\}$ r'_i overlaps r'_{i+1}. If reads are provided in pairs, then the layout can also store the information about paired reads. An assembly can be seen as a layout of the reads (in a way similar to what explain for the Overlap-Layout-Consensus based assemblers).

Phillippy et al. [143] starting point is the consideration that *de novo* assembly is based on the double-barreled shotgun process, therefore the layout of the reads, and implicitly the layout of the original DNA fragments, must be consistent with the characteristics of the shotgun sequencing process. In particular the authors noticed that sequences of overlapping reads must agree and that the distance and the orientation between mated reads must correspond to the expected statistics. They noticed that mis-assembly events fall into two major categories: repeat collapse/expansion and sequence rearrangement. In the former case, the assembler fails in estimating the number of repeats in the genome, while, in the latter case, the assembler shuffles (translocates or inverts) the order of multiple repeated copies. So far, these features have been based on assembly of a genotypic sequence, though their extensions to haplotypic sequences can be achieved *mutatis mutandis*.

Single Nucleotide Polymorphisms (SNPs) are usually good indicators of collapsed or mis-assembled regions (see Figure 4.5). In fact, since single base read errors occur uniformly randomly, while SNPs can be identified by their correlated location across multiple reads, a collapse (or an expansion) can be recognized by the local variations in coverage. A missing repeat causes reads to stack up in the remaining copies, increasing the read density locally. Conversely, a repeat expansion causes a reduced read density among the copies (see Figure 4.6 and 4.7).

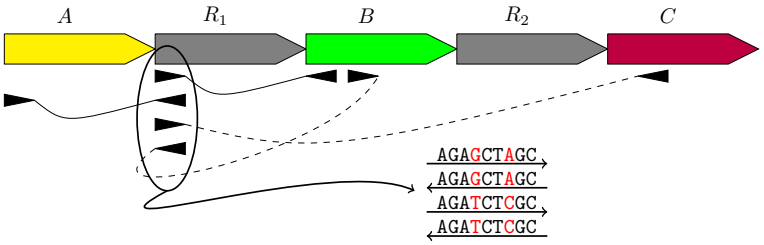
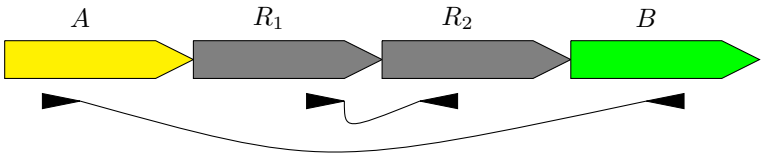


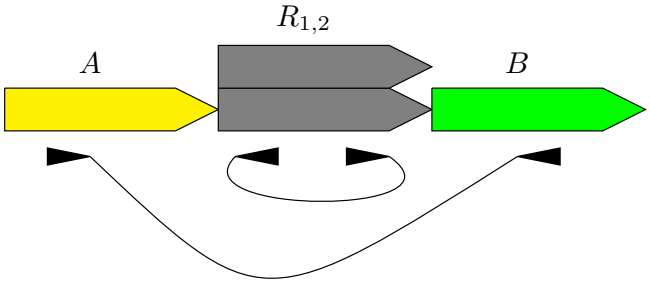
Figure 4.5: SNPs are good indicators of misassemblies: they cause reads to pile-up and are indicative of wrong layouts. Dashed lines represents wrong layouts induced by nearly identical sequences.

Mate pairs highlight incorrect rearrangements: these events are identified by the associated pair of reads being too close to or too distant from each other, mate pairs orienting in wrong directions or reads with an absent mate (in the assembly) or a mate in a different (wrong) contig. Obviously, multiple mate-pair violations are expected to co-occur at a specific location in the assembly in the presence of an error.

Another important way to assess assembly correctness introduced in [143] relies on k-mers. By comparing the frequencies of k-mers computed within the set of reads (K^R) with those computed solely on the basis of the consensus sequence (K^C), it is also possible to identify regions in the assembly that manifest an unexpected multiplicity. For each k-mer in the consensus, the ratio $K^* = K^R/K^C$ is computed. K^* has an expected value close to the average depth coverage.

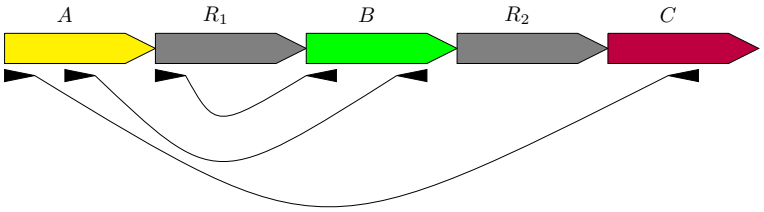


(a) Correct Assembly

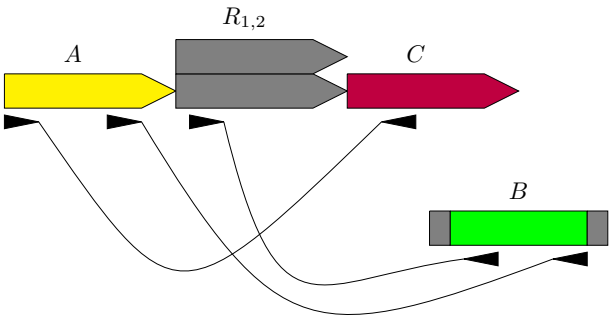


(b) Misassembly

Figure 4.6: Assembly Forensics: Collapse/Expansion events. Nearly perfect repeats (R_1 and R_2) cause collapse events that can be identified via layout inconsistencies.



(a) Correct Assembly



(b) Misassembly

Figure 4.7: Assembly Forensics: Collapse/Expansion events. Nearly perfect repeats (R_1 and R_2) cause collapse and misassemblies events that can be identified via layout inconsistencies and wrong pairs assignments.

Positions in the consensus where K^* differs from expected values can be hypothesized to have been mis-assembled. Further information can be extracted from unassembled reads (*i.e.*, leftovers). Unassembled reads that disagree with the assembly can reveal potential mis-assemblies.

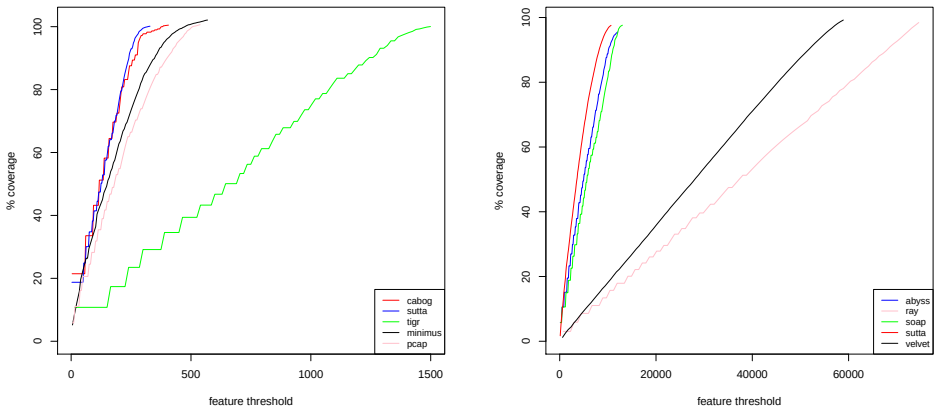
Phillippy and colleagues in [143] proposed a tool dubbed *amosvalidate* able to collect a set of features to be assoceted to each contig. In particular they identify 12 features based on read coverage, k-mer coverage and read layout. The 12 features are used to assess the overall assembly

quality and correctness. It is suggestive of feature analysis that if a contig is found to contain several features (of different types), then a likely explanation could be found in the contig's mis-assemblies. Despite the indirectness of how features diagnose problems in assemblies, this approach represents a significant improvement over the simple standard metrics described in Section 4.3.1. However, the results from feature analysis are strongly dependent on how the features are combined. It is expected that different features are symptomatic of different assemblers. Yet, it is not immediately clear how the simple feature counting can be used to compare the performances of two or more assemblers.

An innovative way to improve the *forensic method* of Phillippy and colleagues has been proposed by Narzisi and Mishra in [130] and it known under the name of Feature Response Curve (FRC). FRC captures the trade-off between quality and contig size more accurately.

The FRC shares many similarities with classical ROC (receiver-operating characteristic) curves, which are commonly employed to compare the performance of statistical inference procedures. Analogous to ROC, FRC emphasizes how well an assembler exploits the relation between incorrectly assembled contigs ("features") against genome coverage, when all other parameters (read-length, sequencing error, depth, etc.) are held constant. The FRC characterizes the sensitivity (coverage) of the sequence assembler as a function of its discrimination threshold (number of features).

After running *amosvalidate*, each contig is assigned the number of features that correspond to doubtful sequences in the assembly. For a fixed feature threshold w, the contigs are sorted by size and, starting from the longest, only those contigs are tallied, if their sum of features is $\leq w$. For this set of contigs, the corresponding approximate genome coverage is computed, leading to a single point of the Feature-Response Curve (FRC). FRC allows to easily compare different assemblies by simply plotting their respective curves. FRC can be applied to all the features or to a subset of them (or even just a single one, if a particular kind of error is of interest). Two examples of



(a) FRC of Brucella suis genome.

(b) FRC of E. coli genome.

Figure 4.8: Two examples of Feature Response Curve (FRC): in Figure 4.8(a) we plotted the FRC for the Brucella Suis assemblies obtained with 5 Sanger-assemblers. In Figure 4.8(b) we plotted the FRC for the E. coli assemblies obtained with 5 NGS-assemblers.

FRC are provided in Figure 4.8. Figure 4.8(a) shows the 5 curves obtained assembling an 8× Sanger coverage of the Brucella suis genome with CABOG [118], MINIMUS [169], PCAP [62], SUTTA [129], TIGR [170]. Steeper a curve is less features are necessary to cover the genome. In the example of Figure 4.8(a) CABOG and SUTTA produce the best assemblies, while TIGR performs slightly worse than the others. Similar analysis can be done looking at Figure 4.8(b). In this case we show the FRC obtained assembling a 130× Illumina coverage of the E. coli genome

composed by reads of length 36 bp. The five curves depict the results obtained with Velvet [188], SUTTA [129], Ray [17], ABySS [166], and SOAPdenovo [100].

Forensics features and Feature Response Curve represent a great breakthrough in assembly validation ad comparison: (i) the framework is not dependent on the availability of reference sequences or closely related genomes; (ii) analysis can be adapted to every genome and to every sequence technology; (iii) features are a direct consequence of the sequencing process.

However, there are several points that need to be explored and improved. The features proposed by Phillippy [143] and later used by Narzisi in [130] are not the only ones that can be extracted. More features can provide a better picture of the overall assembly quality. Moreover, many features are intricately correlated, thus amplifying certain errors while subduing others into less prominence. For example, an area with high k-mer coverage is likely to contain many paired read features. This example raises the question whether it would be possible to concentrate the analysis to only a handful of meaningful features or use a linear combination of few such features to create newer and better set of synthetic and meaningful features. It would be desirable to plot the FRC on a minimal subset of the most important features or on a small number of synthetic features capable of capturing the most important information (*i.e.*, variation).

A practically important aspect is the fact that the tool *amosvalidate* proposed in [143] is designed for Sanger-like projects (*i.e.*, long reads and relatively low coverages), while the present trend is towards using NGS data. In addition to the software performances (*amosvalidate* at the present moment cannot be used on large NGS projects) there is a more subtle problem duo to the fact that NGS-assemblers (ABySS [166] and SOAPdenovo [100] among the most popular) do not provide the layout as output (usually an *afg* file). A possible workaround is to map the reads back to the assembly, but this is obviously problematic especially for what concerns reads mapped in multiple places (*i.e.*, reads mapped on repeats).

4.4 *De Novo* Assembly in Practice

So far we analysed Assembly Problem under several prospectives. We first focused our attention on the computational problems related to AP, after we analysed available solutions and finally we discussed how to evaluate and compare different instruments and results.

Especially in the second part, we focused our attention on NGS-base solutions, in particular on Illumina-based tools. However, we have not discussed yet the practical achievable results with these tools. Moreover, it is not clear if NGS-based assemblies are comparable to those obtained with Sanger data.

In this last Section we will discuss and analyse results obtained with two real data sets. Both datasets belong to genomes that have already been sequenced and assembled with Sanger based technology. However, for different reasons, in both cases we are interested in performing *de novo* assembly.

The first dataset belongs to a grapevine variety dubbed Sangiovese. The *Vitis vinifera* sequencing project [69] focused on a highly *homozygous* variety (PN40024) in order to simplify the assembly process and the subsequent analysis. Grapevine genome has length $\sim$ 480 Mbp. The final goal of assembling Sangiovese dataset is to better understand and analyse the differences between PN40024 and Sangiovese. Sangiovese dataset comprises 6 Illumina lanes generated with an Illumina Genome Analyser II. The overall dataset forms a 89$\times$ coverage composed by paired reads of length 100 bp. The mean insert size is approximately 250 bp (see Table 4.2 for a complete description).

The second dataset belongs to a poplar variety dubbed Poli. Poli, like Sangiovese, is an highly heterozygous organism and belongs to the *Populus nigra* species. Poplar has been already sequenced [172]: the reference sequence has length $\sim$ 410 Mbp and belongs to *Populus trichocarpa* species. In this case *de novo* assembly is essential in order to study the differences between the two species. Poli dataset comprises 6 Illumina lanes generated with an Ilumina Genome Analyser II. The overall dataset forms a 82$\times$ coverage composed by paired reads of length 100 bp and 114 bp (only one lane). Three lanes forming a 48$\times$ coverage have insert size length of approximatively

250 bp, while the other three are characterized by an insert size of $\sim$ 500 bp (see Table 4.2 for a complete description).

Dataset	# lanes	Coverage	Read Length	Insert Size
Sangiovese	6	89×	100 bp	250 ± 50
Poli	6	82×	100 - 114 bp	250 ± 50 and 500 ± 100

Table 4.2: Sangiove's and Poli's description of the total coverage, insert size, and read length.

The aim of the evaluation was to study NGS limits especially compared to known Sanger assembly results. We were particularly interested in understanding how the coverage affects the assemblers performances. Therefore, for both Sangiove and Poli datasets we produced 9 subsets representing different random coverages: $10\times, 20\times, \ldots 80\times$ and a final set containing all the reads (89× for Sangiovese and 82× for Poli).

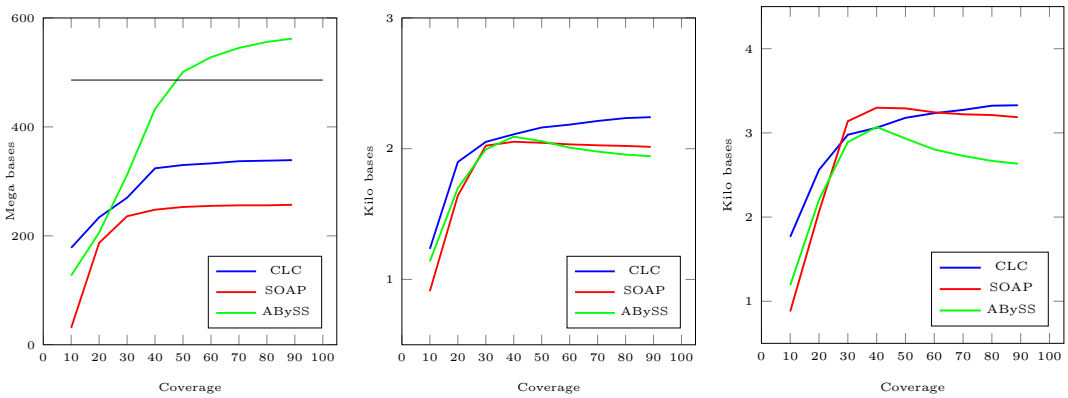
We choose 3 assemblers to assemble the 18 datasets: our goal was both to understand how coverage affects *de novo* assembly and also to estimate the performances of different assemblers. At the time of the experiments, the only available assemblers able to cope with large genomes and with large amounts of data were ABySS [166], CLC CELL3.0 [26], and SOAPdenovo [100]. Recently, other assemblers able to work on this huge datasets have emerged: ALLPATHS [45], RAY [17], SGA [165], and others. However, algorithms and data structures used by all these assemblers are almost the same [134]. The main differences among assemblers lay on the implemented heuristics and on the greedy choices made at several steps of the computation. The consequence of this fact, as we will see, is that results obtained with different assemblers do not differ too much from each other.

To evaluate assembler performances we used some of the standard statistics described in Section 4.3. In particular we based our analysis on *length-based* and *reference-based* statistics. As far as the former, we computed *total assembly length*, *mean contig length* and *N50 contig length*. For what concerns the latter statistic type we used the available reference sequences to compute *correctly reconstructed contigs* number, *correctly reconstructed contigs* length, *correctly reconstructed exons* number and *correctly reconstructed exons* length.

Even though in Section 4.3 we introduced the notion of *assembly forensics* and *Feature Response Curve* (FRC) we do not use them here. The *amosvalidate* pipeline [143] has been designed for small Sanger-based assemblies and has been tested only on bacterial genomes. Even if, as showed by Narzisi and Bud in [130], *amosvalidate* pipeline can be used also on NGS-based assemblies, it cannot be applied on datasets composed by hundreds of thousand of reads like the grapevine and the poplar ones.

Length-based statistics have been computed to have an idea of assembly's connectivity level. Although these statistics are of dubious value (as stated in Section 4.3 and as we will extensively see in Chapter 6) they are useful in this context to compare results obtained with NGS-data and Sanger-data.

Reference-based statistics, instead, are of primary importance to gauge assemblers ability to correctly assemble genomes. We concentrate our attention on two different aspects: correctly assembled contigs and correctly reconstructed exons. Contigs statistics have been computed by aligning contigs against the reference sequences (PN40024 genome for Sangiovese datasets and Tricocarpa genome for Poli datasets). We say that a contig is *correctly reconstructed* if it aligns against the reference with at least one hit which length is longer than 90% of the contig itself and with *similarity* higher than 90%. To compute the number of correctly reconstructed exons we aligned the coding regions of the two references against the assemblies. We say that an exon is correctly reconstructed if we are able to find it on the assembly sequence with a single hit of length 99% of its length and with similarity of 96%. We decided to align exons as a proof that NGS-based assemblers are able to reconstruct at least coding regions: coding regions are the most important sections of a genome, therefore one can accept an highly fragmented assembly if he knows that most of the genes are present in it.



(a) Assembly Length (b) Mean Contig Length. (c) N50 Length.

Figure 4.9: Sangiovese assembly comparison: Length-based Metrics.

Figures 4.9 and 4.10 show ABySS, CLC and SOAPdenovo performances on the 9 Sangiovese coverages. In Figure 4.9(a) we can see how CLC and SOAPdenovo are able to reconstruct the genome only partially. ABySS shows an utterly different behaviour, constantly improving the total assembly length as coverage increases. However, it is well known[2] that ABySS outputs nearly identical contigs in presence of heterozygous regions. In other words, if the two alleles have even a single difference, ABySS outputs both the possibilities, while other assemblers like CLC and SOAPdenovo choose one of the two (the most representative one) or use an ambiguous character (IUPAC alphabet). It is important to notice that both solution are acceptable: both approaches have some weak and some strong points.

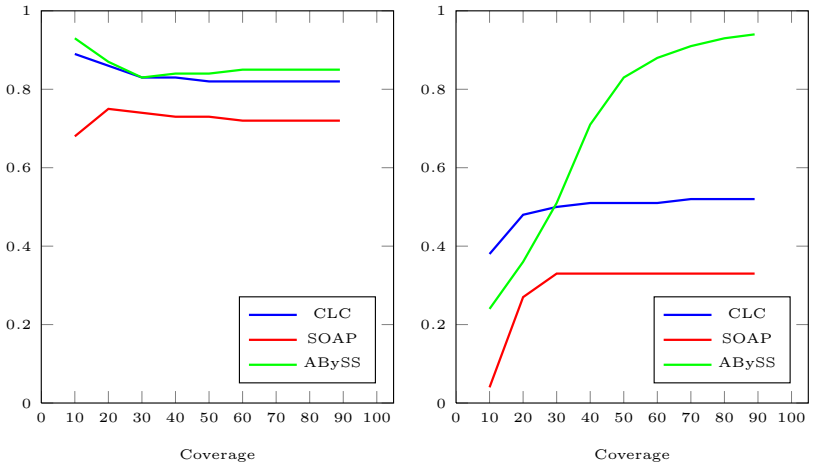
Statistics showed in Figures 4.9(b) and 4.9(c) stress that all the three assemblers stop to improve at a fast pace when coverage reaches 50×. Surprisingly this behaviour suggests us that a coverage higher than 50× gives us small enhancements. It is worth pointing out the mean contig length and the N50 contig length obtained with Sanger sequencing [69]: grapevine reference genome has a mean contig length of 37 Kbp and an N50 of 100 Kbp. With a 8× Sanger coverage and with a Sanger-based assembler (Arachne [11]) we are able to obtain a mean contig length 20 times longer and an N50 length 30 times longer.

Reference-based statistics are somewhat discouraging. In Figure 4.10(a) and 4.10(b) we can see how in general only 80% of the contigs are correctly aligned against the reference sequence and that these correctly aligned contigs cover less than 50% of the overall genome, at least with CLC and SOAP. It is worth noting how, with the ABySS exception, contig correctness and coverage seem not correlated when coverage is higher than 30×. Better news arrive from Figure 4.10(c) and 4.10(d). In this case we can appreciate how all instruments are able to reconstruct large part of the exonic regions (in particular ABySS and CLC).

In Figures 4.11 and 4.12 we can observe similar results for what concerns the Poli dataset. The conclusions we can draw are fairly similar to the ones discussed in the Sangiovese case: length-based statistics are an order of magnitude lower than those achievable with Sanger based project (Figures 4.11(a), 4.11(b), and 4.11(c)), contigs correctly aligning against the reference sequence represent a small portion of the entire sequence (Figures 4.12(a) and 4.12(b)) but this portion contains almost all the exons (Figures 4.12(c) and 4.12(d)).

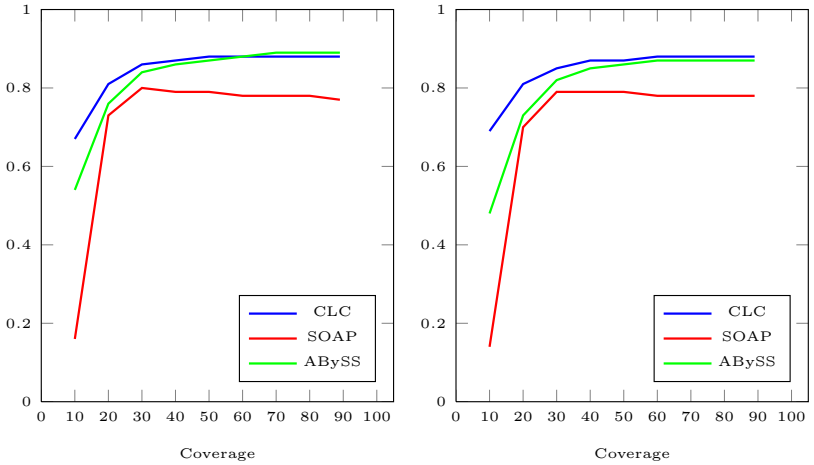
It is worth noting that the goal of these experiments was not to compare the performances of the three assemblers, but to understand their capabilities and their limits. The three assemblers, used on different real datasets, can produce utterly different results. However, the distance between the results achievable with Sanger-based projects are really far from those achievable with NGS-based ones.

[2] ABySS user group and personal communication with ABySS authors



(a) (#Correct Contigs)/(# Contigs)

(b) (Correctly Placed Length)/(Genome Length)

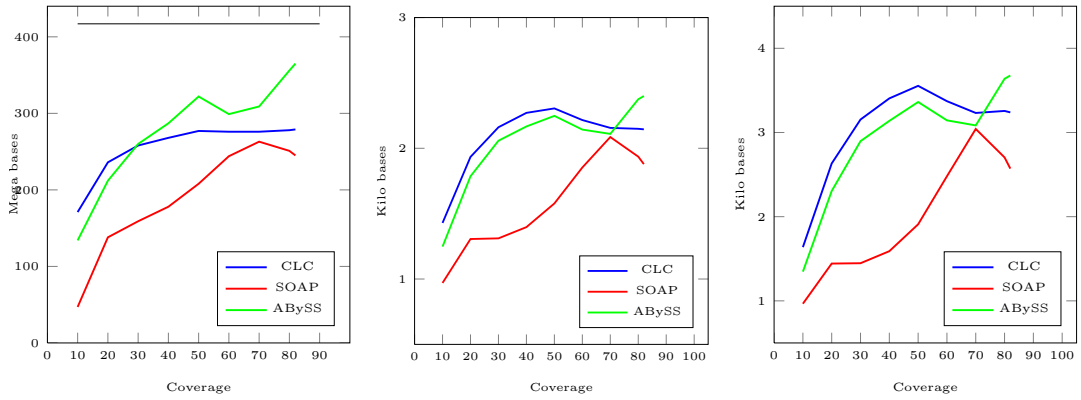


(c) (# Placed Exons)/(# Exons)

(d) (Length Placed Exons)/(Length Exons)

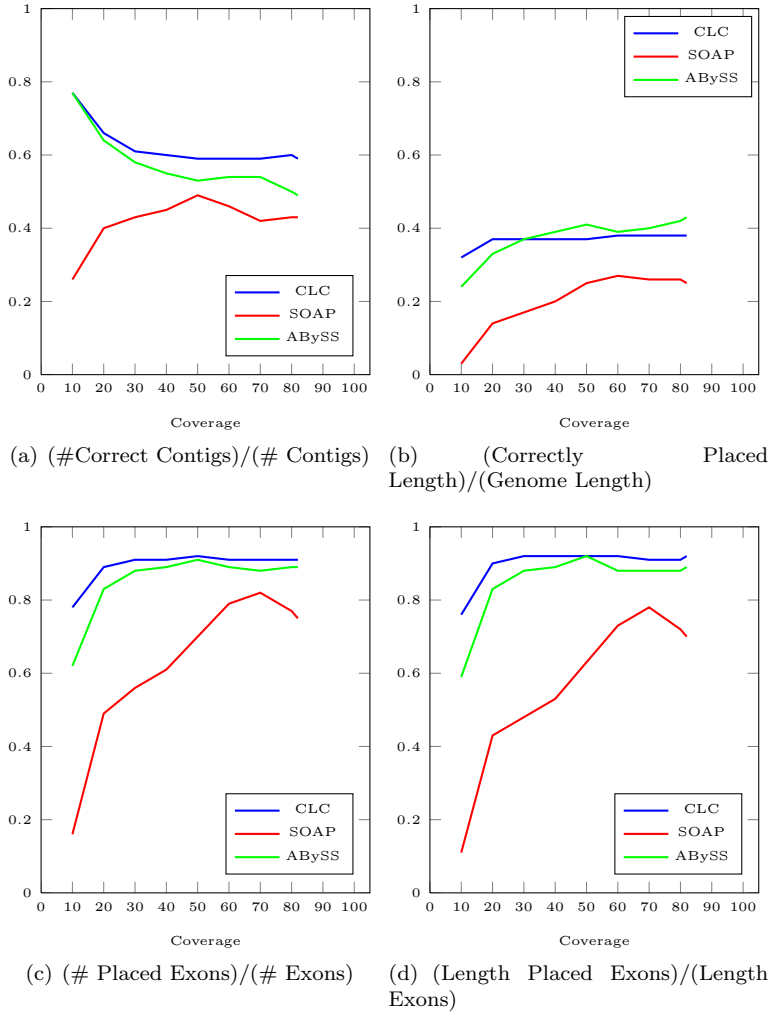
Figure 4.10: Sangiovese assembly comparison: Reference-based Metric.

In order to close the gap between new NGS-projects and old Sanger-projects both technology and implementation must improve. On the one hand new technology improvements will provide longer reads and new data types (*e.g.*, strobe reads) useful to produce more accurate and precise assemblies. On the other hand, more advanced implementations, able to use reads in a better way will give us the possibility to take more advantage from high coverages.



(a) Assembly Length (b) Mean Contig Length. (c) N50 Length.

Figure 4.11: Poli assembly comparison: Length-based Metrics.



(a) (#Correct Contigs)/(# Contigs) (b) (Correctly Placed Length)/(Genome Length)

(c) (# Placed Exons)/(# Exons) (d) (Length Placed Exons)/(Length Exons)

Figure 4.12: Poli assembly comparison: Reference-based Metrics.

4.5 Conclusions

In Section 4.1 we analysed *de novo* assembly computational problems. Section 4.2 was dedicated to the solutions proposed so far to solve assembly problem. In Section 4.3 we focused our attention on validation and evaluation problems. Finally, in Section 4.4 we presented a collection of real results obtained on two real datasets through three widely used tools.

The final picture is somewhat strange: we conclude that beside assembly problem formulations are inherently NP-hard, there are several practical solutions based on heuristics and approximations that we can use to obtain a solution. However, the obtained solutions are difficult to evaluate and to compare to each other. When a comparison is possible thanks to the presence of a reference sequence, we discover that results are dramatically distant from the ones achievable with Sanger-based projects.

Despite we can identify between three and four assembly paradigms, de Bruijn graph approach is the *de facto* standard in order to assemble NGS data, especially Illumina. It is clear that there is systematic failure of all *de novo* assemblers in correctly reconstruct the original sequence: this failure can be identify in the difficulties in assembling repeated regions.

However, a ray of hope is still present: as showed in the last Section NGS-assembler are able to reconstruct the exonic part of the genome, new technologies are approaching on the market with solutions able to simplify the *de novo* task. Moreover, assemblers are improving at a fast speed. In a recent published work [45] the assembler ALLPATHS was able to produce an assembly of a human genome comparable to the Sanger based data. Not surprisingly, the key feature of this tool is the use of particular type of reads.

5

De Novo Assembly: Solving the Puzzle

So far we have analysed and, in some extent, criticized the attempts to study and solve the Assembly Problem (AP). On the one hand we saw how all the proposed models theoretically jeopardize our possibility to solve AP due to their computational intractability (*i.e.*, NP-hardness). However, on the other hand, several tools based on heuristics and greedy strategies have demonstrated, in practice, that AP can be efficiently solved. Despite such results, several problems are still open. Moreover, public available NGS-assemblers have performances and results often not acceptable (see Section 4.4 of Chapter 4).

The aim of this chapter is to propose two new contributions to solve Assembly Problem under certain restricted hypothesis. In Section 5.1 we will describe *eRGA* [25, 176] a tool that connects Alignment Problem (discussed in Chapters 2 and 3) and Assembly Problem. eRGA works under the hypothesis that a reference sequence belonging to a closely related genome is available. In such a way, eRGA is able to improve results achievable with standard approaches. In Section 5.2 we will introduce a tool dubbed *GapFiller* [126] whose aim is to locally assemble genome regions and to return as output small fragments (less than 1000bp) that are certified to be correct. GapFiller may become the first step of a more complex assembly pipeline, or it may be used for target assembly to reconstruct insertion events among individuals of the same species.

5.1 Integrating Alignment and *De Novo* Assembly: eRGA

When a completely new organism is to be sequenced, the principal assembly strategy is the celebrated Whole-Genome Shotgun Assembly (WGSA). As we saw in Section 4.2 of Chapter 4 there are several tools (*assemblers*) able to solve the Assembly Problem. *De novo* assembly with NGS-reads turns out to be, in general, very difficult [127] and as we showed in Section 4.4 state of the art *de novo* assembly tools are likely to produce a set of highly fragmented contigs.

A possible approach to reconstruct the original sequence of a new organism is *Reference Guided Assembly* (RGA). The number of organisms whose genome has been completely sequenced has been increasing rapidly each year and, for this reason, is becoming viable to sequence an organism and then align produced reads against a closely related genome. RGA consists in two phases: first all the reads are aligned against the reference genome, and second a consensus sequence is extrapolated. Everywhere the coverage drops to zero there are two possible solutions: replace those areas with the reference sequence or with a sequence of N's.

This problem was already studied in the Sanger sequencing context. In [145] and [46] two methods were proposed able to use a reference sequences to assist the assembly of new organisms. The challenge has become even more interesting after the NGS outbreak and the possibility to sequence (and resequence) hundreds of new organisms at low cost and at deep coverage. As showed in Chapter 2 the number of available NGS-aligners is extremely large. Such tools are essential in order to perform reference guided assembly. The main problem with RGA and NGS is that—essentially for efficiency reasons—mapping algorithms are highly conservative: it is possible

to align reads only with a low number of errors and, usually, with no more than one gap. In other words we are able to reconstruct conserved regions, while we cannot reconstruct divergent and (usually) more interesting areas. While, for example, there are techniques that use paired reads information to identify insertions [90], there is no clear way to reconstruct them.

An interesting strategy to improve *de novo* assemblies is known under the name of "assembly reconciliation". As different assemblers run over the same data set may produce different results, the goal of assembly reconciliation is to merge the assemblies produced by different tools. This is done either to obtain longer contigs as well as to detect possible mis-assemblies. Some form of assembly reconciliation was applied with success in Sanger based projects like in [2].

eRGA propose an approach similar to assembly reconciliation. In presence of a closely related sequence one can perform both *de novo assembly* and *reference guided assembly* and then *merge* the two results to obtain what we call an *enhanced reference guided assembly* (eRGA).

5.1.1 Integrating Assemblies

In [31] a sequence coming from a related organism was used as reference while in [36] even if a reference sequence was present, it was well known that the sequenced organism was different from it. In those situations assembly programs can lead to a fragmented assembly, while reference guided assembly can easily fail to assemble the most divergent areas. In other words, both reference guided and *de novo* assembly methods have some weakness and some strengths. In [131] a tool (MAIA) has been proposed to integrate multiple *de novo* assemblies and multiple reference guided assemblies. This tool uses the output of different assemblers and of different reference guided assemblies obtained with several reference sequences with the goal of improving the final assembly result. MAIA constructs an overlap graph from the pairwise alignments of all the contigs. In large and repetitive genomes, like plants genomes, this step is computationally expensive and could easily lead to a large number of ambiguous or false overlaps.

In two independent works Casagrande et al. [24] and Zimin et al. [191] suggested two methods which goal was to merge two draft assemblies. In different ways, they proposed the construction of a graph which allowed to localize areas that could be merged. Zimin method relied on a global alignment step, while Casagrande avoided this step by using the layout provided by the assemblers. Both methods have been developed for Sanger sequencing projects. Once the graph is constructed, it can be used to merge and extend contigs and to highlight possible errors in the assembly. In particular, cycles witness a situation in which the two assemblers disagree. In this case Casagrande *et, al.* used one of the two assemblies as an anchor to resolve conflicts (*Master Assembly*).

5.1.2 Reference Guided Assembly Approaches

When a reference sequence A and a set of reads $\mathcal{R}$ are given, there are essentially two possible ways to perform reference assembly. The standard way consists in simply aligning all the reads in $\mathcal{R}$ against the reference A and then obtaining some consensus sequences. In the ongoing we will refer to this method as standard-RGA (s-RGA), in a similar way we will call the consensus sequence produced s-A. The other way is to first perform *de novo* assembly on $\mathcal{R}$ and then align the resulting contigs against the reference A. We will call this second method *de novo*-RGA (dn-RGA), and the consensus sequence produced in this way will be called dn-A. Both the output sequences have "N" everywhere the coverage drops to 0. With the purpose of simplifying the discussion we suppose A composed by a single sequence (and therefore s-A and dn-A are composed by a single sequence). It will be clear that this is not a limitation.

In presence of next generation sequencing data, we have to use aligners like SOAP2 [94] and rNA [177] to obtain s-A. Even aligners like rNA, specifically designed to align highly divergent reads, are considered "conservative", especially if compared to BLAST-like aligners (*i.e.*, long reads aligners). In particular NGS aligners allow a limited (usually one) number of indels. For this reason the length of s-A is almost the same of A. The sequence dn-A is obtained in three phases, first reads are assembled using a short read assembler [188, 96], the resulting contigs are then aligned

against A, and after this phase the consensus is generated. Contigs can be aligned using tools like BLAST [7] or others that allow us to place reads on a reference with low similarity constraints.

Several tools have been proposed to address this task. OSLay [152] computes a synthetic layout of the contigs using a reference sequence to anchor *de novo* sequences; the Mauve aligner [153] gives as output an ordered version of the *de novo* contigs; PGA [190] is able to layout the contigs with more than one reference genome at a time using global searches. All these tools implement or use a BLAST-like [7] search to align contigs against the reference. This alignment technique allows us to place reads on a reference with low similarity constraints. In particular, the contigs can be aligned against the reference sequence allowing partial hits and gaps.

In presence of a closely related genome, one can produce both s-A and dn-A. As we saw, both assemblies have some good and some bad points. In general, this situation is similar to the already studied situation of assembly reconciliation [24]. Given the two assemblies s-A and dn-A we will show that the same ideas can be applied in this different context to obtain an enhanced reference assembly.

5.1.3 The Merge Graph

In order to formalize the discussion in a simple setting, we will work with two strings, Δ and Γ over the alphabet $\Sigma = \{a, c, g, t, N\}$. Given two indexes i and j belonging to a string we define the *interval* $[i, j] = \{i, i + 1, \ldots, j\}$.

Given a string S (in particular for $S \in \{\Delta, \Gamma\}$) we define the interval set I_S containing all the possible intervals belonging to S as the set:

$$I_S = \{[i, j] | i < j \wedge i, j \in \{0, \ldots, |S| - 1\}\}$$

With S_{ij} we identify the S's substring $S[i, \ldots, j]$. If $S_{ij} \in \{a, c, g, t\}^*$ we will call it a *(pure)contig*, while if $S_{ij} \in \{N\}^*$ it will be named *gap*. If $S_{ij} \in \{a, c, g, t\}^*$ is such that $i = 0 \vee S[i - 1] = N$ and $j = |S| - 1 \vee S[j + 1] = N$, then S_{ij} is a *max-contig* an we define in an analogous way a *max-gap* (in general we will speak of max-area).

Given two strings δ and γ the function $\mathcal{D}(\delta, \gamma)$ returns a number between 0 and 1 representing the percentage of difference between the two strings under the Levenshtain distance. In particular, if ι represents a global alignment between δ and γ with edit distance d, then the percentage of difference is the ratio between the edit distance d and the length of ι ($\mathcal{D}(\delta, \gamma) = d/|\iota|$). This definition can be easily be extended to distance metrics different from the edit distance.

The merge graph $G_M^{\Delta,\Gamma} = (V, E)$ is a directed graph such that $V \subset I_\Delta \times I_\Gamma$ can be partitioned into four sets called *gap-nodes* (V_g, gap against gap), *delta-nodes* (V_δ, a Δ-contig against a Γ-gap), *gamma-nodes* (V_γ, a Δ-gap against a Γ-contig) *merge-nodes* (V_m, contig against contig). We also fix two parameters λ and s. The former one guarantees a similarity constraint between regions belonging to Δ and Γ. The latter one, instead, introduces a locality constraint between regions of the two strings connected by the merge graph. More precisely, the set V_g is defined as:

$$V_g = \{\langle [i, j], [k, l]\rangle | (\Delta_{ij} \in \{N\}^* \wedge \Gamma_{kl} \in \{N\}^*) \wedge \ |i - k| \leq s \wedge |j - l| \leq s\}$$

The set V_δ is defined as:

$$V_\delta = \{\langle [i, j], [k, l]\rangle | (\Delta_{ij} \in \{a, c, g, t\}^* \wedge \Gamma_{kl} \in \{N\}^*) \wedge \ |i - k| \leq s \wedge |j - l| \leq s\}$$

a V_γ is defined in an analogous way. The last set, V_m, is defined as:

$$V_m = \{\langle [i, j], [k, l]\rangle | \Delta_{ij} \in \{a, c, g, t\}^* \wedge \Gamma_{kl} \in \{a, c, g, t\}^* \wedge \mathcal{D}(\Delta_{ij}, \Gamma_{kl}) \leq \lambda$$

$$\wedge \ |i - k| \leq s \wedge |j - l| \leq s\}$$

In other words a gap-node connects two gaps in the sequences, a δ-node or a γ-node connects a contig with a gap in the other sequence, while a merge node connects contigs that must be *similar*.

All the nodes in V must respect the following global property:

$$\forall\langle[i,j],[k,l]\rangle \ \in V \wedge \ \forall\langle[i',j'],[k',l']\rangle$$

$$\in V \setminus \langle[i,j],[k,l]\rangle \ ([i,j]\cap[i',j']=\emptyset \wedge [k,l]\cap[k',l']=\emptyset)$$

The last property tell us that every interval appearing in a node on a sequence (either Δ or Γ) does not overlap with *any* other interval belonging to the same sequence (Δ or Γ, respectively).

The set of oriented edges E is defined as:

$$E=\{(p,q)\in V_y\times V_z | y,z\in\{g,\delta,\gamma,m\} \ \wedge \ y\neq z \ \wedge \ p=\langle[i,j],[k,l]\rangle$$

$$\wedge\ q=\langle[j+1,u],[l+1,v]\rangle \ for\ some\ i,j,k,l,u,v\}$$

Less formally, the last definition tell us that the intervals stored in two nodes p and q connected by an edge (p,q), are subsequent and belong to two different sets of nodes.

Definition 11 (Merge Graph $G_M^{\Delta,\Gamma}=(V,E)$) *$G_M^{\Delta,\Gamma}=(V,E)$, with V and E defined as before, is a merge graph for the strings Δ and Γ if its non-oriented version is connected, $p=\langle[0,j],[0,l]\rangle$ is the (only)* source (first node)*, $q=\langle[i,|\Delta|],[k,|\Gamma|]\rangle$ is the (only)* sink (last node)*, and for each max-contig and max-gap S_{sr} with $S\in\{\Delta,\Gamma\}$, there exists a path $p_1,\ldots,p_l$ with $l\geq 1$ such that $p_1=\langle[s,o],[a,b]\rangle$, $p_l=\langle[u,r],[c,d]\rangle$ or $p_1=\langle[a,b],[s,o]\rangle$, $p_l=\langle[c,d],[u,r]\rangle$ for some a,b,c,d,u,o.*

Lemma 4 *Given the strings Δ and Γ, the graph $G_M^{\Delta,\Gamma}$ is acyclic.*

Proof 2 *A path in $G_M^{\Delta,\Gamma}$ is a sequence of gap, copy and merge nodes. Every node is composed by two intervals, and from the definition of edge it follows that a path induce a growing function between the intervals stored in the path. A cycle must contain an edge connecting a node $\langle[i,j],[k,l]\rangle$ to another one $\langle(m,n),(u,v)\rangle$ such that $m<i$ and $u<i$. Therefore such edge cannot exists and hence $G_M^{\Delta,\Gamma}$ must be acyclic.*

Merge Graph and Global Alignment

Given the strings Δ and Γ, there is a deep connection between merge graph $G_M^{\Delta,\Gamma}$ and a global alignment between them. We will show how we can obtain a global alignment from a merge graph and how a specific global alignment allows us to build a merge graph. In the ongoing we will refer to such global alignment as *Merging Global Alignment* (MGA). The merge graph $G_M^{\Delta,\Gamma}$ can be used to *extract* a family of *edit strings*. From each node we can produce an edited version of the two substrings that are represented. From delta and gamma nodes we simply extract the contigs while from the merge and gap nodes we can produce an edited version of one of the two strings. In the gap nodes case the edited version will contain only insertions and deletions (see figures 5.1(b) and 5.1(c)). Once the edit strings are computed the corresponding MGA can be calculated.

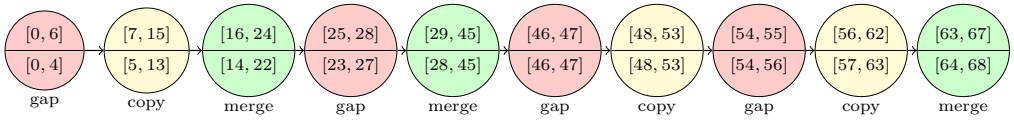
The other direction is more complicated. A MGA is a global alignment that respects two kinds of local properties: *locality* and *similarity*. In general a global alignment does not guarantee these local properties, hence we can easily construct a global alignment that violates a local constraint (see figure 5.1(d)). As a consequence of the Merge Graph definition, the global alignments we are seeking must respect the following properties: if Δ_{ij} and Γ_{kl} are aligned one against the other, then $|i-k|\leq s$ and $|j-l|\leq s$ (locality); if Δ_{ij} and Γ_{kl} are aligned and at least one is a max-contig or $\Delta[i-1]=\Gamma[l+1]=N$ (or $\Delta[j+1]=\Gamma[k-1]=N$) then $\mathcal{D}(\Delta_{ij},\Gamma_{kl})\leq\lambda$ (similarity).

If such an alignment exists, calling Δ' and Γ' the two strings over the alphabet $\{a,c,g,t,N,-\}$ returned by the global alignment between Δ and Γ, we can build $G_M^{\Delta,\Gamma}$ by simply reading from left to right Δ' and Γ'. For each position we have to judge if a new node begins or if we can continue extending the current one.

The determination of this global alignment can be computationally cumbersome. We cannot simply use an algorithm that calculates an optimal sequence alignment because the algorithm may make a choice that can create an optimal global alignment that is not necessarily an alignment

```
          0           11          21          31          41          51          61
Δ:  NNNNNNNaag gtttaaggtc ctacaNNNNa ctcatcataa aaacccNNNN NNNNNNctct aggtaaaa
Γ:  NNNNNNNNNN NNNNggccta cacNNNNNac tcatcatcaa aaacccNNaa aaaaNNNNNN NNNNaaaaa
```

(a) The strings Δ and Γ ($|\Delta| = 68$, $|\Gamma| = 69$)



(b) A possible $G_M^{\Delta,\Gamma}$ for Δ and Γ with $s = 2$ and $\lambda = 0.2$

```
Δ′:  NNNNNNN AAGGTTTAA GGTCCTACA- NNNN- ACTCATCATAAAAA-CCC NN NNNNNN NN- CTCTAGG TAAAA
Γ′:  NNNNN-- NNNNNNNNN GG-CCTACAC NNNNN ACTCATCATCAAAAACCC NN AAAAAA NNN NNNNNNN AAAAA
Λ:   MMMMMII MMMMMMMMM MMIMMMMMMD MMMMD MMMMMMMMMSMMMMDMMM MM MMMMMM MMD MMMMMMM SMMMM
```

(c) A possible Global Alignment obtained from $G_M^{\Delta,\Gamma}$ (Λ edit string to transform Γ into Δ: `M` means match, `S` means substitution, `I` means insertion while `D` means deletion)

```
Δ′:  NNNNNNN AAGGTT- TAAGGTCCTACA- NNNN- ACTCATCAT-AAAAACCC NN NNNNNN NN- CTCTAGG TAAAA
Γ′:  NNNNNNN NNNNNNN ---GG-CCTACAC NNNNN ACTCATCATCAAAAACCC NN AAAAAA NNN NNNNNNN AAAAA
                     constrain
                     violated
```

(d) A Global Alignment that does not allow the creation of MG

Figure 5.1: An example of $G_M^{\Delta,\Gamma}$ construction and of the corresponding MGA extraction.

that respects all the local constraints (see Fig. 5.1(d)). We will now sketch a complete algorithm that given the strings Δ and Γ generates all possible $G_M^{\text{s-}A,\text{dn-}A}$.

The algorithm starts by reading the two sequences from left to right. For every contig in Δ and Γ we can recursively compute all the possible alignments that satisfy the locality and possibly the similarity constraints. More in detail consider Figure 5.2. Let us assume that the latest generated node is $\langle [z, i], [m, v] \rangle$. From the merge graph definition we have that at least one between i and v must be the end of a max-area, in this case i. At this point we have to calculate the nearest (from i) max area end, n in the case of Figure 5.2. So the node we are going to create is $\langle [i, k], [v, n] \rangle$ with $n - s < k < n + s$ (paying attention to some special case we can reduce the search space). In order to generate all the possible graphs we have to recursively generate all the nodes. In the case we are generating a merge node we have also to check if the similarity constraint is respected. The algorithm terminates because at every step it proceed forward along both strings.

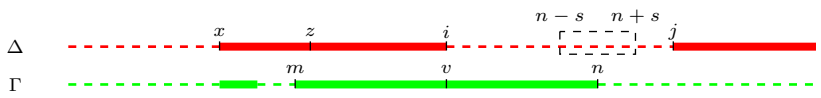


Figure 5.2: A possible situation during MGA construction.

Minimal Merge Graph

Given two strings Δ and Γ it is clear that the existence of a $G_M^{\Delta,\Gamma}$ depends on two thresholds s and λ. By setting s to be large enough we can easily compute a merge graph composed only by gap, delta and gamma nodes. In a similar way, another trivial solutions can be found setting λ equal to 1. In this case the merge nodes have only to respect the locality constraint.

It is clear that a merge graph of a pair of sequences is interesting when s is a small constant if

compared to Δ and Γ lengths and when λ is not close to 1. We will show that these two requests can help us in designing a practically better algorithm.

5.1.4 The Merge Graph and Reference Guided Assembly

A merge graph is a data structure able to describe a global alignment between two strings with the further request of local constraints. This data structure can be used to describe the relations between two strings and to extract a consensus.

A merge graph can be used to obtain an improved assembly by merging the results achieved with s-RGA and dn-RGA. When working with s-A and dn-A it becomes interesting to use an useful *assembly reconciliation* concept [24]: one of the two sequences is elected to be the *Master Assembly* (MA), that is the assembly we believe to be correct. Hence, when we find a merge node, instead of computing a consensus we simply keep the sequence coming from MA. In practice the MA will almost always be dn-A as it reconstructs the regions present in the sequenced organism and absent in the reference. If $G_M^{\text{s-}A,\text{dn-}A}$ is available it can be used to extract a new assembly. For each node $p =< [i,j],[k,l] >$ we extract the sequence dn-A_{kl} if p is a gamma or a merge node, s-A_{ij} if p is a delta-node or the shortest between s-A_{ij} and dn-A_{kl} if p is a gap node. This assembly method is named e-RGA (*enhanced Reference Guided Assembly*) and the sequence produced is dubbed e-A .

An Approximate Construction

The main problem of the *brute force* algorithm presented in Section 5.1.3 is the fact that for each max-area we must generate $2s$ different nodes to test the feasibility of all the possibilities. Moreover, in the merge node case we must perform a global alignment, hence for each max-contig S_{ij} the brute force algorithm has a worst case complexity of $\mathcal{O}(s * (j-i)^2)$. Additionally the parameters s and λ are unknown and, in general, there is no clear way to know them in advance or to at least approximate them. This problem is a consequence of the fact that in different nodes (*i.e.*, in different locations) both the locality and the similarity constraints could be (locally) utterly different.

When working with s-A and dn-A we have that the merge graph $G_M^{\text{s-}A,\text{dn-}A}$ must exist for some $s \in \{0,..,|\text{dn-}A| - |\text{s-}A|\}$. This follows directly from the construction of the two strings.

It is more difficult to limit λ. A practically good approximation is the percentage of difference allowed in the *de novo* contig alignment.

The particular context provided by s-A and dn-A allows us to further improve the construction algorithm concentrating ourselves only on a significant subset of all the global alignments associated to $G_M^{\text{s-}A,\text{dn-}A}$.

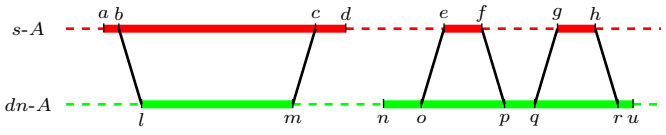


Figure 5.3: Example of the window heuristic for e-RGA construction.

We will show that thanks to the some intrinsic properties of s-A and dn-A the brute force algorithm can be improved avoiding the generation of all the possible global alignments. For each max-contig in dn-A we can easily identify the regions that will form a merge node once that s is fixed. Let us consider Figure 5.3. The max-contig dn-A_{lm} will form the merge node $< [x,y],[l,m] >$ for some x and y between b and c with $b = l - s$ and $c = m + s$. As a matter of fact s-A_{xy} and dn-A_{lm} must be very similar: during the construction of the two strings we were able to reconstruct the same area by placing directly reads and contigs (generated with the same reads) on the same piece of reference. In order to identify x and y we can proceed in the following way: we split

dn-A_{lm} into the set $\{dn_1^l, dn_2^l, .., dn_k^l\}$ of k non overlapping sequences of length d and we locally align each sequence on a small (s-based) window of s-$A_{b,c}$ allowing a percentage of difference of at most λ, with λ the percentage of difference allowed while building dn-A. Thanks to the similarity between the sequences we will place most of the dn_i^l and we will approximately identify x and y. By building an index over s-A and assuming λ to be small enough (*i.e.*, the number of errors allowed while aligning dn_i^l is small) the procedure takes time $\mathcal{O}(m-l)$. This procedure can be extended also to more complex situations (see right part of Figure 5.3) to identify all the merge nodes.

Once all the merge nodes are computed we can link them with the appropriate alternation of delta, gamma and gap nodes. With all the merge nodes fixed we can either decide to produce all the possible graphs like in the brute force algorithm or proceed in a greedy way by minimizing the total length of e-A.

Clearly this approach cannot be applied to two general strings Δ and Γ because we have neither an hypothesis that allows us to limit s and λ nor a high percentage of similarity between the sequences associated a merge node.

5.1.5 eRGA: Implementation and Results

The pipeline represented in Figure 5.4 has been implemented using several third-part tools and a set of Perl scripts to parse results and to obtain the consensus sequence e-A. The first step

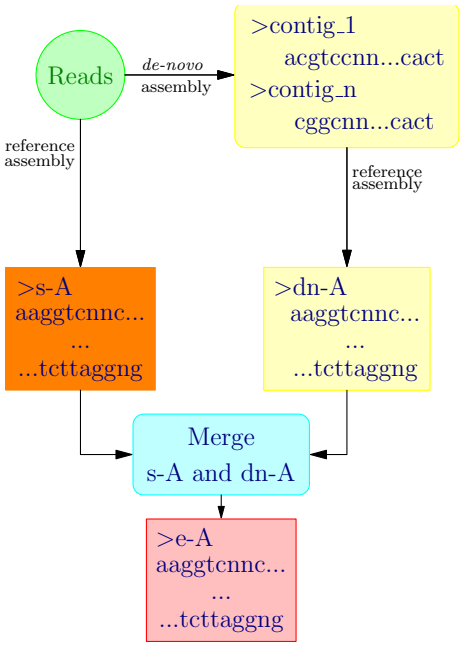


Figure 5.4: e-*RGA* pipeline implementation.

to construct s-A is to employ a short-string aligner to align all the reads against the reference sequence, and subsequently extract a consensus sequence. In all cases in which a read is found in multiple occurrences, we randomly choose one of the alignments. We used the short-string aligner rNA [144] and the pileup command provided by samtools [94] to extract the consensus sequence.

dn-A was obtained by first performing *de novo* assembly with ABySS [166] and CLC assembler Cell 3.0 [1]. We noticed that, using contigs from both assemblies, the amount of genome reconstructed in dn-A greatly improves. The delicate phase of mapping contigs against the reference sequence was accomplished with CLC-Workbench (www.clcbio.com). Although we used these specific tools, clearly the production of s-A and dn-A can be carried out using different software

without significant modification of the pipeline. The core of e-RGA is the $G_M^{\mathsf{s}\text{-}A,\mathsf{dn}\text{-}A}$ construction and e-A generation. These crucial phases are implemented within a Perl script that uses BLAST [22] to perform the approximate alignment. The program first memorises both s-A and dn-A, and localises all the max-areas (max-contigs and max-gaps).

The Datasets

We tested e-RGA on several data sets with the goal of demonstrating that it can be successfully used in different situations. In particular we performed tests on genomes of various sizes to show how this method can easily scale to large projects.

The first dataset (Chloroplast) consists of an Illumina multiplexed lane used to sequence four conifer chloroplast genomes with single reads of length 45 base pair (bp). The genome length is close to 120 Kbp. A reference sequence for spruce chloroplasts is totally missing: as suggested by Cronn et al. [31] a pine chloroplast (Pinus thunbergii or Black pine) was used as a reference. The e-RGA pipeline turned out to be useful both for the absence of a reference and for the strong reads'contamination.

The second dataset (Microbe) is composed by an Illumina lane used to sequence a microbial genome (Methylomirabilis oxyfera) with single reads of length 32 bp. The genome length is approximately 2.7 Mbp. A reference sequence is present, however reads were sampled from a meta-population that was expected to be different from the reference genome [36]. This is another context in which e-RGA can be used.

The third dataset (Sangiovese), comprises 6 Illumina lanes used to re-sequence a grapevine variety (Sangiovese, Rauscedo clone R24) with 100bp paired-end reads for a 89× total raw coverage. For this dataset, we used as reference sequence the genome of the highly homozygous grape clone, PN40024, used as reference genotype from the French-Italian Consortium for grape genome characterisation [69].

The fourth, and last dataset (Poplar) comprises 6 Illumina lanes used to sequence a Poplar individual belonging to the Populus nigra species, with 100bp paired-end reads for an 85× total raw coverage. In this case, we used as reference sequence the Populus trichocarpa genome [172].

All these datasets represent different situations and different application fields for e-RGA. In the Chloroplast case the absence of a reference prevents us to use a standard reference guided approach, while the high contamination content jeopardize the possibility to perform *de novo* assembly. In the Microbe dataset, we have a closely related sequence, but it would be interesting to identify non conserved regions. The last two datasets are challenging for their length and composition: both grapevine and poplar are characterised by long, repetitive genomes (480 Mbp and 417 Mbp respectively); moreover, both the sequenced individuals are highly heterozygous. These three conditions (length, repetitiveness and heterozygosity), together with the presence of two reference genomes, are ideal for give a chance to e-RGA. It must be noticed that, while in the Sangiovese dataset we used a reference sequence belonging to the grapevine species, in the Poplar dataset the reference belongs to a different species. This difference is important in order to understand the results.

In all the four datasets, before assembling and aligning, all the reads were filtered for quality. In the Sangiovese and Poplar case we also eliminated all sequences belonging to chloroplasts and mitochondria. For this purpose we employed rNA filtering option [177] (see Chapter 6).

Results Discussion

In Table 5.1 we report the results for Chloroplast and Microbe datasets. For each experiment we computed the coverage achieved by placing with the same tool and the same parameters the reads on A (the sequence used as reference), on s-A, on dn-A and, on e-A. Moreover we reported some standard assembly statistics like number of contigs (# Contigs), mean contigs' length, and N50.

As we can see from Table 5.1 all the standard metrics are improved by e-RGA. As the reference genome was composed by only one sequence, it can look strange that the $N50$ is not close to the expected reference length. As a matter of facts, to compute the $N50$ statistics we break the

	Chloroplast				Microbe
Experiment	Norway	White	Black	Red	
A coverage	40.3×	104.5×	34.7×	18.8×	10.0×
s-A Coverage	46.1×	117.2×	39.5×	20.8×	13.5×
# Contigs	326	307	323	339	7986
Mean contig (*bp*)	280	289	273	265	221
N50 (*bp*)	778	883	853	763	583
dn-A Coverage	51.0×	136.0×	45.2×	23.4×	14.7×
# Contigs	106	59	106	159	2630
Mean contig (*bp*)	871	1715	902	609	254
N50 (*bp*)	1477	2759	1288	1002	710
e-A Coverage	55.5×	142.8×	48.1×	24.7×	17.7×
# Contigs	129	113	130	156	5600
Mean contig (*bp*)	794	951	798	699	344
N50 (*bp*)	2614	3313	2465	1842	1168

Table 5.1: Chloroplasts and Microbe datasets results. For all the techniques used, we show the coverage obtained mapping reads back to the sequences, the number of contigs, the mean contig length, and the $N50$ length computed both on the expected genome length

sequence (A, s-A, dn-A, and e-A) every time a sequence of 5 unknown characters (Ns) occurs. We can appreciate that the number of aligned reads against e-A (that can be computed from the coverage) is higher than the number of aligned reads against the other sequences.

Tables 5.2 and 5.3 summarise the results from the Sangiovese and Poplar datasets. As a measure of the assembly quality and correctness, we report the percentage of aligned reads (the same reads used to perform reference and *de novo* assembly), the number of contigs reconstructed, the mean contig length, the $N50_g$ (the length of the longest contig such that the sum of all the contigs greater than it represents half the expected genome length) and, in brackets, the $N50_c$ (the length of the longest contig such that the sum of all the contigs greater than it represents half the total contig length), and the percentage of Ns in the sequence. The $N50_g$ gives us a normalised value that describes the connectivity level of the assembly.

As for Chloroplast and Microbe datasets, these statistics have been computed for the reference sequence A, for the s-RGA output s-A, for the dn-RGA output dn-A, and for the e-RGA output e-A. For these two particular datasets, we also computed the statistics for the *de novo* assembly output *dn* (in particular CLC Cell 3.0 output) as a practical demonstration that assemblers gave us a results that must be improved.

In the Sangiovese case (Table 5.2), we can see how the mean length obtained through e-RGA is longer than the other approaches, and although the dn-A mean length has a close value, we must consider the fact that these contigs cover only half the genome length as described by the high percentage of unknown characters. In the Sangiovese dataset, the most impressive results are the $N50_g$ and $N50_c$ improvements. Both e-A's $N50_g$ and $N50_c$ are better than those of s-A, and they largely improve the results achievable with *de novo* assembly alone. This shows that our pipeline can effectively improve the final assembly result.

Similar results are summarised in Table 5.3 for the Poplar dataset. Owing to the distance between the sequenced organism (*Populus nigra*) and the reference genome (*Populus trichocarpa*), Poplar results can look less promising than Sangiovese's ones. However, the number of mapped reads against e-A is higher than the number of reads mapped against both s-A and dn-A. The fact that we are able to map a higher number of reads against *dn* should also be a consequence of the distance between the reference and the sequenced genome. As far as the standard assembly statistics are concerned ($N50_g$, $N50_c$ and mean contig length), we can again see how the e-A results are better than those achievable by simply mapping reads or contigs back to the reference. Despite *de novo* assembly results look much better than those obtained by other approaches, we

	% Aligned Reads	# Contigs	Mean Contig	$N50_g$ ($N50_c$)	%Ns
A	80.21%	-	-	-	3.00%
s-A	80.99%	246752	1758 bp	8514 bp (9901 bp)	7.64%
dn	53.10%	289854	1942 bp	1753 bp (3328 bp)	0.70%
dn-A	50.71%	109833	2246 bp	600 bp (3947 bp)	47.70%
e-A	81.77%	198194	2282 bp	12494 bp (14219 bp)	6.40%

Table 5.2: Sangiovese datasets result. For all the techniques used, we show the percentage of aligned reads, the number of contigs, the mean contig length, the $N50$ length computed both on the expected genome length and on the total contig length, and the number of unknown characters "N".

must stress the fact that *de novo* assembly alone gives us a set of 116.683 unordered contigs, with no information about their position in the final genome. A further measure of the improvements introduced by the use of e-RGA is the number of successfully aligned paired reads (*i.e.*, paired reads that align on the sequence at the expected distance and orientation). In both datasets, e-A is the sequence on which the largest number of constraints is respected.

	% Aligned Reads	# Contigs	Mean Contig	$N50_g$ ($N50_c$)	%Ns
A	55.00%	-	-	-	2.14%
s-A	58.00%	778065	365 bp	525 bp(1105 bp)	25.22%
dn	67.84%	116683	2728 bp	2906 bp (4487 bp)	0.40%
dn-A	37.00%	77370	1335 bp	0 bp (2085 bp)	62.46%
e-A	59.00%	558762	482 bp	957 bp (1959 bp)	18.56%

Table 5.3: Poplar datasets result. For all the techniques used, we show the percentage of aligned reads, the number of contigs, the mean contig length, the $N50$ length computed both on the expected genome length and on the total contig length, and the number of unknown characters "N".

5.2 Closing the Gap: GapFiller

A clear, practical, limitation of Next Generation Sequencing (NGS) is the read length. The reduced read length with respect to Sanger sequencing is problematic for the alignment task (*i.e.*, it is more difficult to disambiguate reads belonging to repetitive regions) and it is dramatic for *de novo* assembly (repeats longer than read length pose serious problems to assemblers). In particular, as reads' length decreases, the number of "unsolvable" repeats increases; namely, the repeated structure complexity of (any region within) a genome depends on the data used to assemble it [127].

Results from old Sanger-sequencing projects tell us that everything become much easier in presence of long reads. The situation would be ideal if such long reads can be produced at the same cost of today available NGS sequences. In particular this would give the possibility to use assemblers like Arachne [11] and PCAP [61] that already proved their capabilities with long reads (*i.e.*, Sanger reads).

Illumina state of the art sequencer, is by default able to read the tips of DNA fragments of length $\sim 600 - 1000$ bp[1]. The instrument is able to return only the first and the last 100 bp, leaving therefore a gap of length approximately $400 - 800$ bp. With all NGS technologies it is not unusual to reach extremely high coverages that guarantee that each base of the sequenced genome is covered by more than one read [86].

The idea behind GapFiller (GF) is to close the gap within given paired reads, using an alignment algorithm and some techniques to avoid errors. GF can be viewed as a "local" assembler, as its

[1]There are also several protocols to produced mate-pairs at a longer distance.

main target is to produce accurate longer sequences, with respect to NGS reads' length. The goal is to produce contigs that are certified correct, in order to allow simpler and accurate subsequent analysis. A second aim is to produce an input for an assembly pipeline, using an opposed strategy with respect to shotgun approach.

GapFiller reduces the assembly complexity in two different ways: firstly, it concentrates the assembly effort on a limited area ($\sim 400 - 800$ bp, independently from genome length) thus avoiding to work with complex and often intractable graph structures; secondly it outputs contigs that can be treated like long reads that can be used more effectively in a *de novo* assembly pipeline to resolve repeats and complex genomic areas.

Our method is based on a *seed-and-extend* schema aimed *closing the gap* between the two reads of a paired read. Similarly to other seed-and-extend based assemblers like SSAKE [182], SHARCGS [35], and QSRA [20] GapFiller selects one read and tries to extend it using reads that overlap it. The main drawback of seed-and-extend assemblers is their inherent incapability to assemble complex (*i.e.*, repeated) genomes. GapFiller does not aim at producing a *de novo* assembly, but it only concentrates on closing the gap within paired reads. The advantages of our method lie in the generation of correctly certified contigs and, as a by-product, in the identification of "difficult" areas (*e.g.*, repeats, low coverage regions), thus avoiding the production of wrong contigs.

5.2.1 A local Seed-and-Extend Strategy

GapFiller uses a local seed-and-extend (*i.e.*, greedy) strategy to close the gap among the paired reads. Given the set of sequenced reads $\mathcal{R}$ and (r^1, r^2) a pair of reads belonging to $\mathcal{R}$ corresponding to a paired read, the strategy is to compute all the overlaps between r^1 and all the reads memorized in $\mathcal{R}$. Once overlaps have been computed, one can *extract* a consensus sequence and reiterate the procedure on it. The *extension* ends when the read r^2 is found/reached. As a matter of facts, there are situations in which this procedure can fail: repeats may conduce the extension on a path that will never reach the mate, errors during the consensus computation may prevent other overlaps to be found or may induce wrong ones. In these cases, we need to define some strategies able to stop the extension and eventually discard what have been produced.

GapFiller firstly stores all *useful* reads in a memory efficient data structure that allows to quickly compute overlaps between the reads and the contig being constructed. In a second phase reads are selected and subsequently extended. Such reads will be identified by the name of *seed* reads. The extension phase halts when a stop condition is reached. Depending on the stop condition, the produced contig is labelled as *trusted* or *not trusted* (*i.e.*, positive or negative).

Another important point is how to choose the reads to be extend (*i.e.*, *seed reads*): the strategy of simply pick up the first not used read and extend it easily fails in reconstructing contigs that are uniformly distributed on the genome. The problem is that reads belonging to repetitive regions are more likely to be chosen. Therefore we need a clever strategy to extend reads belonging to unique regions of the genome.

Another major stumbling block is the overlap computation: the cardinality of $\mathcal{R}$ can be in the order of millions, if not billions in the case of mammalian genomes. A data structure able to store in a feasible amount of space all the reads and able to quickly compute the overlaps is therefore mandatory.

Definitions

Let Σ be an alphabet and Σ^* the set of the words in Σ. For every $S \in \Sigma^*$ we will denote with $|S|$ the number of characters of a and with $S[p, \ldots, p + l - 1]$ the sub-sequence of S starting in $p \in \{0, \ldots, |S| - 1\}$ and of length $l \in \{0, \ldots, |S| - p\}$. We will refer to $S[p, \ldots, p + l - 1]$ as *prefix* if $p = 0$, *suffix* if $p + l = |S|$, and as the p-th character of S, if $l = 0$, and we will simply write $S[p]$.

In order to quickly identify overlaps between the reads' tips and the contig being extended, we use an approach closely related to the Hamming-aware hash function presented in Chapter 3. As extensively showed the Karp and Rabin fingerprint based alignment algorithm can be extended to deal with mismatches, by replacing the simple comparison between fingerprints with a more

refined test. In particular, they noticed that by choosing q to be a Mersenne number ($q = 2^w - 1$, for some $w \in \mathbb{N}$) it could be checked in linear time whether two strings align against each other at a small Hamming distance.

Given a string $S \in \Sigma^*$ and its numerical representation $s \in \mathbb{N}$ in base $|\Sigma|$, the hash function f_H is defined as

$$f_H \colon \Sigma^* \rightarrow \{0, \ldots, q-1\} \tag{5.1}$$

$$S \mapsto f_h(S) := s \mod q, \tag{5.2}$$

where q is a prime of the form $q = 2^w - 1$, for some $w \in \mathbb{N}$. The value $f_H(S)$ is called the *fingerprint* of the sequence in $S \in \Sigma^*$ coded with s.

In the context of quickly compute overlaps between a sequence and a large set of reads, the use of f_H significantly reduces the size of the set employed for the search of the overlapping reads. In practice, all the input reads are indexed by the *fingerprint* of their L-length prefixes, where L is a fixed parameter. So, when a sequence S has to be extended, the suffix-prefix overlaps at a given (limited and constant) Hamming distance with other sequences are quickly computed. Formally, given a set of reads $\mathcal{R} \subset \Sigma^*$, the maximum allowed Hamming distance k, the set $\mathcal{Z}(k, q)$ of the *witnesses* (see Chapter 3 for more details), a sequence S, and a parameter $l \geq L$, the following set

$$\mathcal{R}(S, l) := \{r \in \mathcal{R} \mid (f_H(r[|r| - l, \ldots, |r| - l + L - 1]) - f_H(S[|S| - L, \ldots, |S| - 1])) \mod q \in \mathcal{Z}(k, q)\} \tag{5.3}$$

contains all the reads whose L-prefix overlaps the L-prefix of the l-suffix of S with Hamming distance less than k. False positives can be present but, as showed in [144], their amount is limited. On this ground we have that the search for the overlapping reads can be restricted to those belonging to $\mathcal{R}(S, l)$.

Seed and read selection

It is of utmost importance to use only correct reads during the extension phase in order to avoid the generation of wrong meta-reads. Several tools are available to perform error correction on Illumina data using the so-called "read spectrum" (consider QUAKE [77] and Hammer [115] to mention the most recent ones). Other tools discard reads or try to improve their reliability using quality information (rNA [178] and QSRA [20]). We decided to opt for a combined strategy: reads are first trimmed and filtered using their quality information using a specific rNA option (see Chapter 6) and then, using 16merCounter, another tool developed by us (see Chapter 6), we further filtered reads based on their k-mer spectrum.

As far as 16merCounter is concerned, the idea is to compute the distributions of all k-mers belonging to the (filtered) reads and, for every read, find the average k-mer frequency. Formally, the frequency of a given k-mer z in the set $\mathcal{R}$ is given by

$$F_z := \sum_{r \in \mathcal{R}} \left| \{p \in \{0, \ldots, |r| - k\} \colon r[p, \ldots, p + k - 1] = z\} \right| \tag{5.4}$$

hence, the average k-mer frequency of a read $r \in \mathcal{R}$ can be defined as

$$F_r := \frac{1}{|r| - k + 1} \sum_{p=0}^{|r|-k} F_{r[p,\ldots,p+k-1]}. \tag{5.5}$$

16merCounter computes the 16-mer frequency: we have chosen $k = 16$ in order to guarantee that a k-mer can be stored in 32 bits and hence that the computation requires a fixed and feasible amount of RAM (16 GB). The idea is that, at least with data obtained by oversampling the genome tens of times, low frequency 16-mers might be a reliable signal of presence of sequencing errors. We ran 16merCounter on our datasets and extracted a threshold value for the 16-mer average frequency in order to discard error-affected reads. To do that we simply analyzed the histogram containing the number of 16-mers for every fixed frequency and selected the minimum value among the two peaks (see Figure 5.5).

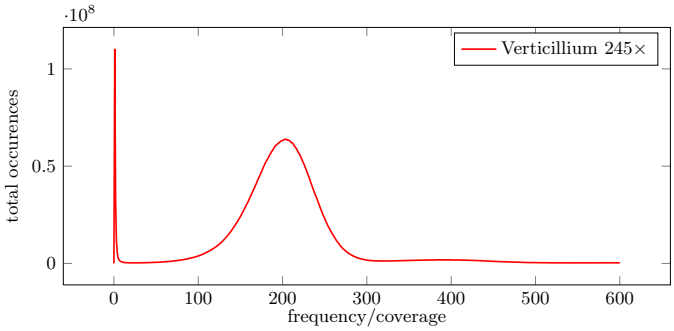


Figure 5.5: Verticillium is a fungi whose genome measures approximately 32 Mbp. In this picture we plotted the frequency of the 16-mers belonging to a $\sim 245\times$ read coverage composed by Illumina reads of length 100 bp

Contig extension

In the contig extension phase, reads are selected one after the other and each one of them is used as *seed* in order to create a new contig. Once a *seed* read is selected, the suffix-prefix overlaps with other reads are computed and, if a sufficiently high level of global similarity is reached, they are clustered in a consensus sequence, that is subsequently used to perform further extensions. The procedure continues while some overlapping reads exist and the consensus sequence is *highly representative* of the clustered reads. If either one of the previous conditions is not met, the current sequence is returned in output.

We will denote with $\mathcal{R} \subset \Sigma^*$ the set of the input reads for GapFiller and with $r_0 \in \mathcal{R}$ a seed read. At step 0 the current sequence is initialized with the seed $S_0 := r_0$. Suppose that at the i-th step of the algorithm the current string is S_i. The procedure to build the sequence S_{i+1} at the $(i+1)$-th step, starting from S_i, is explained in the following. Firstly, the sets $\hat{\mathcal{R}}(S_i, l) \subseteq \mathcal{R}(S_i, l)$ of overlapping reads are selected (see (5.3))

$$\hat{\mathcal{R}}(S_i, l) := \{r \in \mathcal{R}(S_i, l) \colon d_H(r[0, \ldots, l], S_i[|S_i| - l, \ldots, |S_i| - 1]) \leq \delta_1\} \tag{5.6}$$

for every $l = L, \ldots, L + \Delta$, where $L, \Delta, \delta_1 \in \mathbb{N}$ are fixed parameters and $d_H \colon \Sigma^l \times \Sigma^l \to \mathbb{R}^+$ is the Hamming distance. In the following we will refer to Δ as slack and to L as minimum overlap length. We will also denote with $\hat{\mathcal{R}}(S_i) := \bigcup_l \hat{\mathcal{R}}(S_i, l)$ the set of the reads overlapping S_i. We define the starting position of $r \in \hat{\mathcal{R}}(S_i, l)$ with respect to S_i as $I(r) := |S_i| - l$. In order to compute reliable extensions, we require the number of reads to be at least m, a parameter depending on the coverage. If there exists no l such that the l-suffix of S_i is covered by at least m reads of $\hat{\mathcal{R}}(S_i)$, then the algoritms stops; otherwise, the starting position of the consensus string c is

$$I(c) := \min\Big\{|S_i| - l \colon \sum_{h \geq l} |\hat{\mathcal{R}}(S_i, h)| \geq m\Big\} \tag{5.7}$$

while its length is given by

$$|c| := \max\Big\{|S_i| - l + |r| \colon r \in \hat{\mathcal{R}}(S_i, l) \wedge \sum_{h \leq l \wedge q \geq |r| - l} \big|\{r' \in \hat{\mathcal{R}}(S_i, h) \colon |r'| \geq q + l - h\}\big| \geq m\Big\}. \tag{5.8}$$

The consensus sequence c is then computed by selecting the most represented character at every position. For every $j = 0, \ldots, |c| - 1$ and for every $\sigma \in \Sigma$ we define the number of occurrences of σ in position j (with respect to c) among the overlapping reads as

$$occ(\sigma, j; c) := \Big|\Big\{r \in \hat{\mathcal{R}}(S_i, l) \colon 0 \leq j + I(c) - |S_i| + l \leq |r| - 1 \wedge r[j + I(c) - I(r)] = \sigma\Big\}\Big|. \tag{5.9}$$

We define the j-th character of the consensus string as the most represented character in position j

$$c[j] := \arg\max_{\sigma \in \Sigma} occ(\sigma, j; c) \tag{5.10}$$

and we denote the *representation rate* with $v(j;c) := \max_{\sigma\in\Sigma} occ(\sigma, j; c)$. For a fixed threshold $T \in [0, 1]$ a character belonging to the consensus sequence c in position j is said to be a *low represented character* if the following holds

$$\frac{v(j,c)}{\sum_{\sigma\in\Sigma} occ(\sigma, j; c)} < T. \tag{5.11}$$

An extension is accepted only if

$$\left|\left\{j \in \{0, \ldots, |c|\} : \frac{v(j,c)}{\sum_{\sigma\in\Sigma} occ(\sigma, j; c)} < T\right\}\right| \leq \delta_2 \tag{5.12}$$

with δ_2 an user defined threshold. If condition (5.12) holds, the new sequence $S_{i+1} := S_i[0, \ldots, |s| - I(c) - 1].c$ is built and the algorithm goes to the $(i+2)$-th step; otherwise, S_i is returned and labeled as non-trusted.

Stop criteria

The algorithm described in the previous section may potentially extend a contig for an arbitrarily large number of times, without checking any "global" properties of the current sequence. With our method the extension phase halts for four different reasons:

- the available overlapping reads are less than m;
- the consensus string c contains more than δ_2 low represented characters;
- contig length exceeds the maximum length;
- the seed mate is found.

Let S_i be the contig obtained at the i-th step, starting from seed read r_0. The first criterion applies when the consensus string c is empty ($|c| = \emptyset$) and this happens, in particular, when there are no more than $m - 1$ overlapping reads. This kind of scenarios are likely to appear in presence of low covered regions. In such a case the contig produced is labelled as `NO_MORE_EXTENSION`.

The second criterion applies when the consensus being constructed is likely to be the consequence of the presence of reads belonging to different genomic locations. More precisely, this situation is likely to appear when the consensus extension is "trying" to exit from a repeat. In such a situation the extension is halted and the contig is labeled as `REPEAT_FOUND`.

The third criterion is satisfied as $|S_i| > L_{\max}$, where $L_{\max}$ is fixed at the beginning of the algorithm and is usually set to the maximum insert size, plus a tolerance value. In such a situation we could have been able to continue the extension but, however, we would not be able to find the mate read. This suggests that the contig produced may be wrong or, at least, that it contains a high number of unreliable bases. When the maximum allowed length is exceeded, the computation is halted and the contig labeled `LENGTH_EXCEED` is returned.

The fourth criterion is used to stop the extension as the mate $\tilde{r}_0$ of the seed r_0 is found. At the generic i-th step, every $p \in \{0, \ldots, |S_i| - |\tilde{r}_0|\}$ is checked to see whether the following condition is satisfied

$$d_H(S_i[p, \ldots, p + |\tilde{r}_0|], \tilde{r}_0) \leq M \tag{5.13}$$

where M is the maximum number of mismatches allowed between $\tilde{r}_o$ and S_i. Inequality (5.13) is satisfied if and only if the mate is found in S_i at position p with no more than M mismatches. This control is done on-the-fly and hence the positions already checked at the i-th step will not be checked at the $(i+1)$-th one. The *mate-check* criterion is used as a guarantee of correctness of the whole contig. This is in contrast to previous criteria introduced to avoid errors committed during the extension. In other words, the first and the second criteria are strictly *local*, as no information collected in previous steps is used. In this last case the contig returned is labeled as `MATE_FOUND`.

Data structures

GapFiller's *core* is the module working during the extension phase. In this step pairs of reads $(r, \tilde{r})$ are chosen and the (current) seed read r is extended in order to reach its mate $\tilde{r}$, that is known to be at a given (approximate) distance. At this point, we assume that the set $\mathcal{R}$ has already been filtered by discarding reads whose average 16-mers frequency is below the user-defined threshold.

The basic idea is to pre-compute all the information useful to speed up the computation of overlaps needed to perform the extension phase. Suppose that GapFiller is working at the $(i+1)$-th step of an extension, with $i \geq 0$, and let S_i be the current contig. When constructing the consensus sequence c (see Figure 5.6) we are always interested in obtaining overlaps between *suffixes* of S_i and *prefixes* of reads belonging to $\mathcal{R}$.

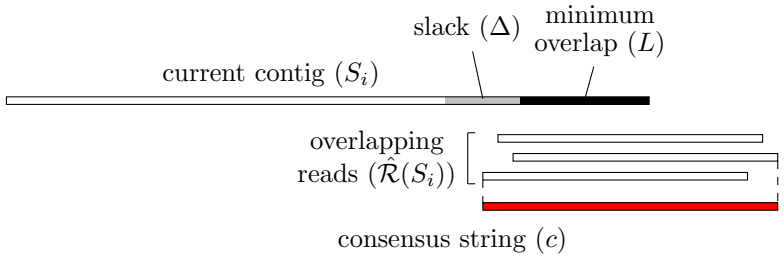


Figure 5.6: The suffix-prefix strategy of GF. Reads overlapping for L characters are selected and the consensus string c is computed; a small number of low represented characters (less than δ_2) in c is a signal that reads come from the same region.

GapFiller overlap computation is based on the rNA alignment algorithm [178]. In particular, a data structure similar to the one proposed in [144] is built. A simplified schema of GF's data structure is presented in Figure 5.7. The basic idea behind GF is the possibility to obtain in a

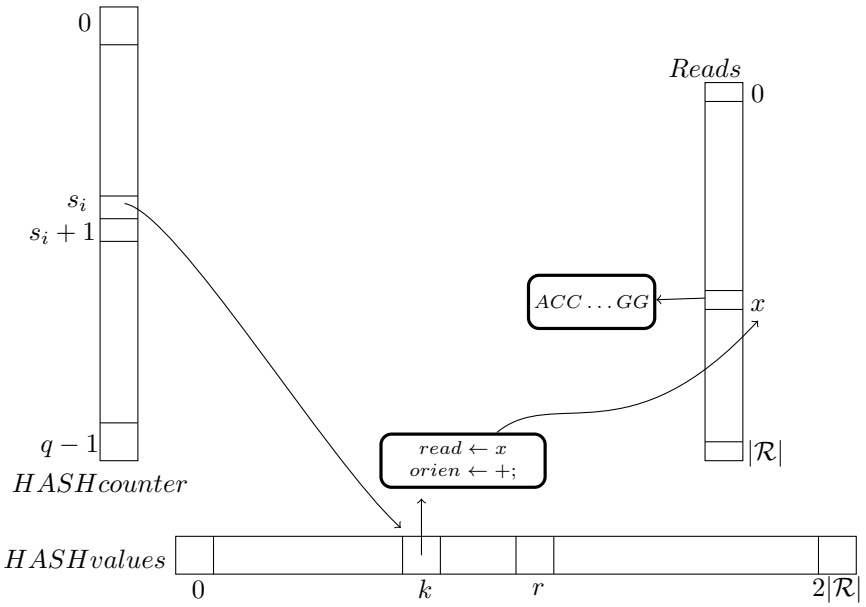


Figure 5.7: The GapFiller data structure is composed of three array whose lengths depend on the number q (used to compute the fingerprints) and on the number of reads $|\mathcal{R}|$.

fast and efficient way the set of reads whose prefixes overlap a suffix of the partial contig being

constructed. Even though rNA solves a more general problem, its strategy can be used in the GapFiller context. rNA is based on a *mismatch-aware* hash function that allows to identify portions of the reference that are likely to contain an occurrence of the searched pattern at a given Hamming distance. The main advantage of rNA'a hash function is its false positive rate that, in general, is low if the fingerprint is obtained computing the value modulo q of the numerical representation of the searched pattern, with q a Mersenne's number ($q = 2^w - 1$). Therefore we used the rNA hash function to find reads that are likely to overlap a suffix of S_i: those reads are subsequently checked to see if they really overlap with S_i or not.

Obviously, all the data must be stored in the main memory, which requires a careful engineering of the data structures. Since overlaps between reads and the the current contig can take place on both strands, reads must be stored together with their reverse complement.

With the goal to save as much memory as possible, reads are stored in an array of integers, in which a base needs 2 bits instead of 8 (A=00, C=01, G=10, and T=11). The data structure used to compute overlaps and to construct contigs is built from the reads. Three arrays are used to represent in a compact way reads stored in $\mathcal{R}$ and to compute overlaps among them:

1. $HASHcounter$: it is an array of pointers to $HASHvalues$. In position i it stores the first position in $HASHvalues$ such that a read r or its reverse complement has a prefix whose fingerprint is i.

2. $HASHvalues$: each array entry stores the read's location in the array *Reads* together with a boolean value indicating whether the fingerprint has been computed from the original read or from its reverse complement. For this reason the size of $HASHvalues$ is twice the number of reads in $\mathcal{R}$;

3. *Reads*: this array stores the reads and other useful informations like k-mer coverage, paired read location, and read status (used, not used, *etc.*).

The overall memory requirement for GF depends on the size of $HASHcounter$ and on the number of reads. As for rNA, a reasonable value for q is $2^{30} - 1$. Such a number guarantees a reduction of the number of false positives (*i.e.*, reads reported to align with the contig suffix, even though they do not overlap with it). As far as the number of reads is concerned, we can limit q, without loss of generality, to 2^{31}: with state-of-the-art Illumina technology, such a number of reads represents approximately a 70× coverage of the human genome. An Illumina read of length 100bp requires two memory locations in $HASHvalues$ of 4 bytes each (31 bits to access array *Reads* and one bit to store the overlap orientation) and one entry in *Reads* of 9 bytes (7 bytes to store the read's numerical representation, one to store the mate position in *Reads*, and one more byte to store information about the k-mer coverage and whether the read has been already used or not). In total the amount of memory required is $4q + 2 * 4|\mathcal{R}| + 9|\mathcal{R}| = 4q + 17|\mathcal{R}|$ bytes.

In order to compute the overlaps between the current contig S and the reads, one has to compute the fingerprints of the substrings $S[|S| - L - y, \ldots, |S| - y]$ with $y \in \{0, \ldots, \Delta - 1\}$. Let us indicate with s_i the fingerprint computed from $S[|S| - L - i, \ldots, |S| - i]$ (see Figure 5.7). GF uses this number to retrieve reads whose L-length prefix is likely to match a substring of S close to the sequence's end. In particular GF accesses all $HASHvalues$ positions between $HASHcounter[s_i]$ and $HASHcounter[s_i + 1]$ and, subsequently, accesses *Reads* to identify the set of candidate overlapping sequences $\mathcal{R}(S, l)$ (in Figure 5.7 GF scans all positions between k and $r-1$ of $HASHvalues$). Finally, the set $\mathcal{R}(S, l)$ is used to compute $\hat{\mathcal{R}}(S, l)$, the set of real overlapping reads. This is done by checking all candidate reads singularly. Due to the fact that only a limited number of mismatches is allowed in this phase and that the employed hash function guarantees a low false positive rate, this step is extremely fast.

5.2.2 Results

GapFiller outputs a set of labelled contigs. The label describes the level of reliability of the sequence, in particular we divide GapFiller's output in two sets: *positive/trusted* contigs are those

labelled `MATE_FOUND`, while *negative/non-trusted* contigs are those labelled `NO_MORE_EXTENSION`, `REPEAT_FOUND`, `LENGTH_EXCEED`. Trusted contigs are those that we consider certified correct and can therefore be used in subsequent analysis. Non-trusted contigs are defined in this way because we were not able to find the mate read and hence we have no way to estimate the correctness.

We decided to perform experiments on both simulated and real data. Despite being aware that simulated experiments results are deeply connected with the capacity of read simulators to successfully reproduce the error schema, we are also conscious that they are the only way to precisely estimate the capacity to correctly (even if locally) assemble reads.

We simulated five bacterial genomes, producing several coverages in order to show how GapFiller's performances scale at different coverages. Moreover, in order to test correctness, we aligned the output contigs against a precise region of the reference, as seed reads' coordinates and orientation are known. The experiments on real datasets were performed on a set of paired-reads extracted from a homozygous *Vitis vinifera* variety. In this case we tested correctness performing a local alignment.

Dataset

The reference genomes used for simulated experiments were downloaded from `http://www.ncbi.nlm.nih.gov`; we used SimSeq, the tool used in Assemblathon 1 [37], to generate paired reads coverages. In particular we performed our experiments on five bacterial genomes (see Table 5.4). We generated different coverages of 100bp-long paired reads, with insert size 600 ± 150bp, using error profiles provided by SimSeq for reads 1 and 2, respectively.

Organism	Genome length (bp)
Alcanivorax borcumensis	3,120,143
Alteromonas macleodii	4,412,282
Bacillus amyloliquefaciens	3,980,199
Bacillus cereus	5,699,545
Bordetella bronchiseptica	5,339,179

Table 5.4: GapFiller: Reference genomes for simulated datasets.

The real dataset is constituted by a 30× Illumina coverage of a grapevine variety PN40024. The peculiarity of this dataset is the fact that reads have been sequenced from the same individual used in the original Sanger-based assembly project [69], thus providing us an easy to validate scenario. The original paired reads are 110bp long and the estimated insert size is 400 ± 100bp.

For each input dataset (real and simulated) we filtered read for quality using rNA `--filter-for-assembly` option and we discarded reads with low average 16-mer frequency (see (5.5)) using 16merCounter to compute 16-mers frequency. For instance, the first plot depicted in Figure 5.8 tells us that a large number of 16-mers occurs less than 10 times, within a paired reads dataset generating a 30× coverage. All these *low-frequency* 16-mers, with high probability, are error-affected. Since reads whose average 16-mers frequency is below 10 must contain at least a low-frequency 16-mer, we do not consider them. It is not even useful to adopt a higher low-frequency threshold, as discarding a large amount of (even correct) data may negatively affect GapFiller's performances.

Design of experiments

We used simulated experiments in order to evaluate GapFiller's ability to correctly reconstruct the gap between two paired reads and in order to assess the reliability of the output classification (`NO_MORE_EXTENSION`, `REPEAT_FOUND`, `LENGTH_EXCEED`, and `MATE_FOUND`). In particular we used these easy to create and validate datasets to explore how coverage and Δ value (see Figure 5.6) affect GapFiller's extension phase.

For this purpose, we extensively tested GapFiller's performances on *Alcanivorax borcumensis*, performing experiments on all the coverages between 30× and 130× with jumps of 10, and trying for every coverage three different Δ values (20, 30, and 40). Notice that $\Delta + 1$ is the number

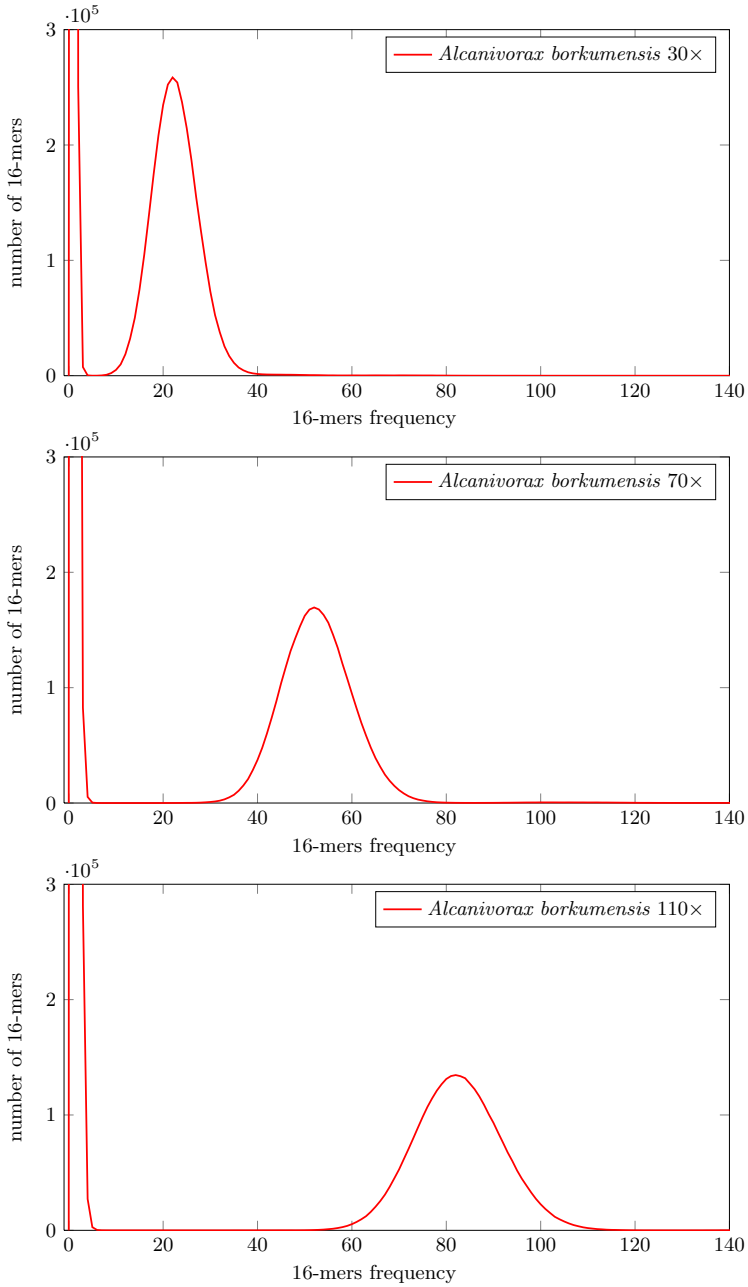


Figure 5.8: *Alcanivorax borkumensis*. 16-mers frequency plot on different simulated coverages.

of suffixes of the current contig to be searched for overlapping reads during a single step of the algorithm. As far as the other four *in vitro* genomes are concerned, we simulated only the coverages between 50× and 90× as a consequence of the results obtained on *Alcanivorax borcumensis*.

As far as the real experiments are concerned, we assembled the 30× coverage with two different Δ values: 20 and 40.

A fundamental parameter is the minimum overlap size L. We decided to fix it in all the experiments to be equal to 50. The value of m, the minimum number of reads required in order to compute the consensus sequence, was automatically set on the basis of the coverage and of the value of Δ being used. In particular, given a set of $|r|$-long paired reads, generating a uniform coverage C, the expected number of reads starting in a $(\Delta + 1)$-long subsequence of the genome is at most

$\lceil(\Delta+1)C/|r|\rceil$. To take into account the fact that C is not perfectly uniform, and that at least two reads are necessary in order to safely extend a contig, we chose $m = \max\{2, \lfloor(\Delta+1)C/4|r|\rfloor\}$. Such parameter is essential in order to label a contig with `NO_MORE_EXTENSIONS`.

The maximum allowed number of low-represented characters δ_2 has been fixed to 8 in all performed experiments. When a consensus sequence is found with more than δ_2 low-represented characters the contig is labelled `REPEAT_FOUND`.

We set the maximum length of a contig to the expected mean insert length plus six times the expected variance (*i.e.*, 1500bp and 1000bp for simulated and real datasets, respectively). Every time the contig being constructed exceeds such a measure, it is labelled `LENGTH_EXCEED`.

We allowed a small amount of mismatches when looking for the presence of the mated read in the contig being constructed with the parameter M. In all of the performed experiments we set M to 5. As soon as in the contig being constructed the mate pair is localized, the contig returned with the label `MATE_FOUND`.

All the experiments were performed on a machine with 8CPU (2500GHz) and 32GB of RAM. Our tool requires a small amount of memory and is extremely fast. For example, setting $\Delta = 40$, GapFiller requires 34 minutes and 4.5GB of RAM on *Alcanivorax borcumensis* 50× and 6 hours and 35 minutes and 5.2GB of RAM on *Bacillus cereus* 90×. To perform the real experiments on *Vitis vinifera*, 29 hours and 31.7GB of RAM were needed.

Statistical analysis

The post-processing phase of GapFiller's output is aimed at both quantitative and qualitative analysis. The first is focused on evaluating the amount of trusted contigs our tool is able to produce, the second on result validation. The main goal is to compare the performances on different input coverages and Δ values.

Due to their nature, simulated experiments allowed to precisely estimate correctness by aligning the contig in the exact place where it is supposed to occur in the reference genome. More precisely, we used the Smith-Waterman alignment algorithm [168], assigning a score of 1 to a match, -1 to a mismatch, and -2 to an indel. For instance, let us consider a contig S generated by extending a seed read r, and suppose that r has been extracted from the genome G at position x, on the positive strand. To test its correctness, S is aligned against $G[x, x+|S|+g-1]$, where g is the maximum number of allowed indels, depending on a user-defined threshold for the alignment score. We say that S is *correctly aligned* if and only if the ratio between the best alignment score of S against $G[x, x+|S|+g-1]$ and $|S|$ is at least 0.95 (for instance, we allow up to 5 mismatches, 1 indel and 1 mismatch, or 3 indels every 200bp, on average). For this particular choice of the alignment score, g is fixed to be $\lceil 3|S|/200\rceil$.

Alignments performed in this way allowed us to divide contigs in four subsets: *true* and *false* positives and *true* and *false* negatives, depending on the contig classification and correctness (see Table 5.5). This gave us the possibility to estimate not only the percentage of correctly recon-

	correctly aligned	not correctly aligned
trusted	true positive (TP)	false positive (FP)
not trusted	false negative (FN)	true negative (TN)

Table 5.5: Contig post-processing classification.

structed contigs, but also to evaluate GapFiller's ability to discern between trusted and not trusted contigs. Using these definitions it is possible to provide a measure of GapFiller's capability to capture correct contigs (*sensitivity*) and to recognize those containing errors (*specificity*).

When using a real dataset reads provenance is unknown, so we tested output correctness by aligning the contigs against the reference genome, using BLAST, and setting the percentage of identity to be at least 95% in order to accept an alignment. In real cases it is interesting to extract two pieces of information from alignments: as in the simulated case, we computed the number of (trusted) contigs that correctly align against the reference; moreover, we analized coverage profile in order to estimate the percentage of the genome being reconstructed by GapFiller.

Results Discussion

Experiments performed on *Alcanivorax borcumensis* showed how GapFiller's performances improve as the coverage increases (see Figures 5.10, 5.11, and 5.12). We can clearly appreciate how GapFiller's sensitivity increases with coverage and reaches a plateau around a 90× (Figure 5.10). Therefore we deduce that it is not necessary to provide coverages higher than 90-100×, as the performances of GapFiller tend to stabilize beyond this value. More important is the fact that specificity is constantly high. This means that almost all contigs declared trusted are actually correct. We also found that the percentage of uncovered bases is negligible, being less than 0.01% even with low input coverages (30×, 40×).

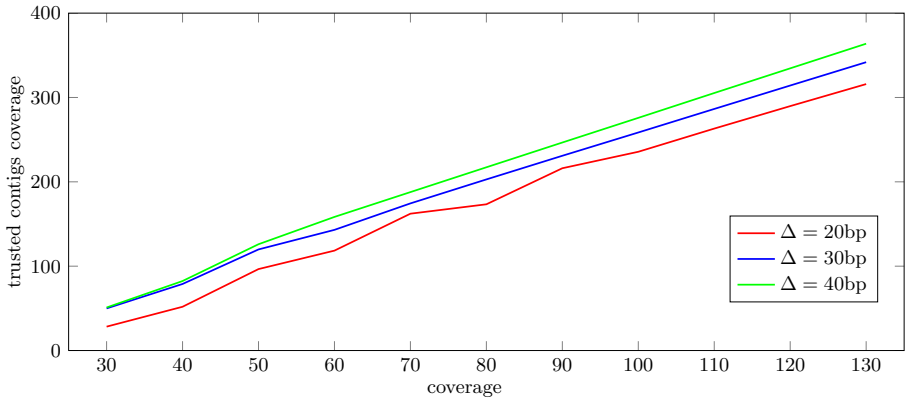


Figure 5.9: *Alcanivorax borkumensis*: trusted contig coverage as a function of the input coverage.

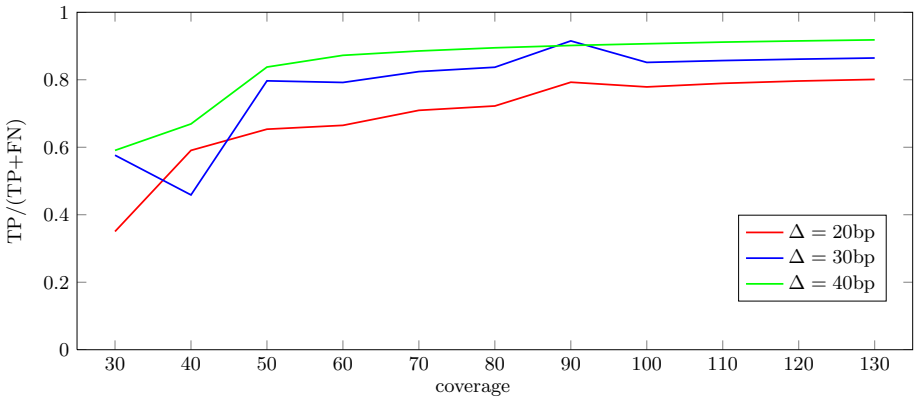


Figure 5.10: *Alcanivorax borkumensis* sensitivity for different input coverages.

As a consequence of the results obtained with *Alcanivorax borcumensis* we limited the experiments on the other four bacterial genomes to coverages between 50× and 90×. From Figures 5.12–5.16 we can see how higher coverages make the percentage of true positive contigs increase, and conversely make the number of false negatives decrease. Going in more detail on the results, we observed that the majority of non-trusted contigs are labelled `NO_MORE_EXTENSION`, meaning that GapFiller stops a contig extension depending on some input dataset features (low covered regions and/or error-affected reads). Another possible scenario is the one in which GapFiller computes a wrong consensus without recognizing it (depending on δ_2 value, see (5.12)).

Figures 5.12–5.16 show that a Δ's large value allows GapFiller to produce better results, in terms of true positives and false negatives rate; in other words, (up to a limit) reaching good output could be possible without providing higher coverages, but increasing the value of Δ.

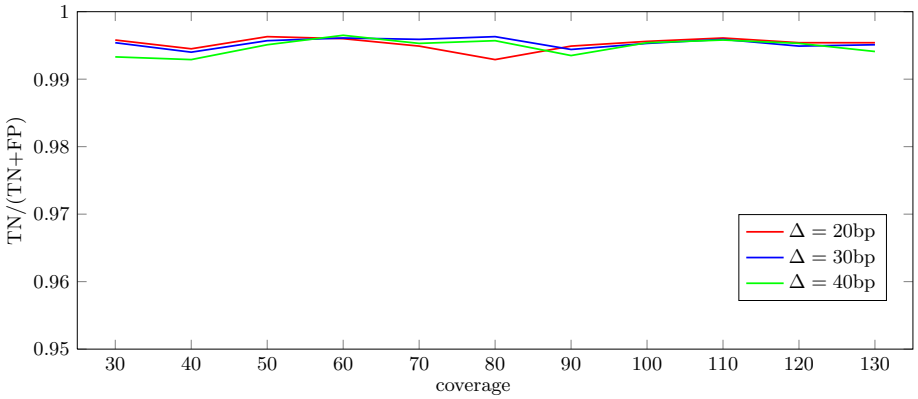


Figure 5.11: *Alcanivorax borkumensis* specificity for different input coverages.

Table 5.6 shows the results obtained on a real 30× Illumina coverage composed of paired reads of length 110bp of a grapevine variety (PN40024). In such a case the alignments have been done using BLAST and considering a contig to be correct if and only if it aligns against the reference genome with a similarity of at least 95% and for at least 90% of its length. From Table 5.6 we

	$\Delta = 20$	$\Delta = 40$
total output coverage	37.58×	41.76×
certified output coverage	17.40×	22.86×
true positives	37.73%	52.60%
false positives	0.12%	0.19%
false negatives	61.16%	46.27%
true negatives	0.99%	0.78%
covered reference (all)	86.11%	82.00%
covered reference (certified)	64.19%	67.10%
sensitivity	38.16%	53.12%
specificity	89.30%	80.69%

Table 5.6: *Vitis vinifera* (486,198,630bp). Results on a 30× input coverage.

can appreciate how, even with a low coverage, the number of false positives is negligible. Also the percentage of reference covered by trusted reads is satisfactory: using only a 30× Illumina coverage we were able to cover more than 60% of the original genome with certified contigs. It must be stressed that the percentage of covered genome, computed by aligning the output contigs with BLAST, represents a lower bound for its real value. Also notice that a real coverage is not uniform, in particular for our dataset 10% of the genome is covered by no more than 5 reads. In these regions the distance between two reads occurring in the genome is expected to be 20bp, on average, and hence there is no way for GapFiller to continue an extension, or to begin a new one, within these areas. For instance, the minimum number of overlapping reads m is set to 2 when the slack is equal to 20, and to 3 when the slack is equal to 40 (see Design of experiments).

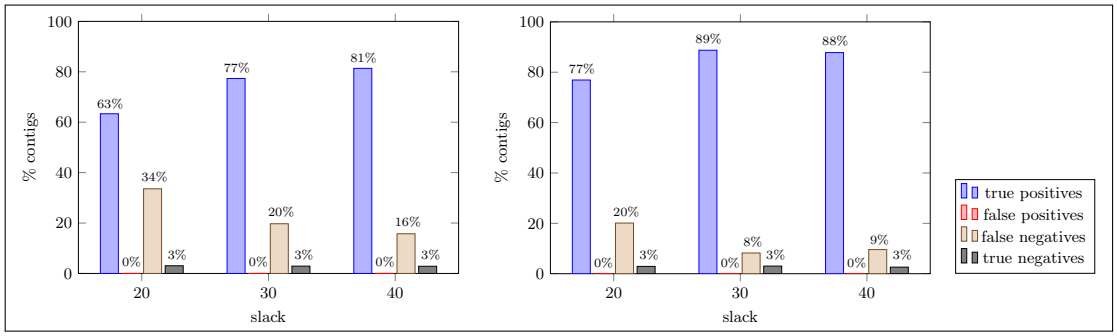


Figure 5.12: *Alcanivorax borkumensis* (50× and 90× input coverages). TP, FP, FN, and TN rates for different values of the slack parameter Δ.

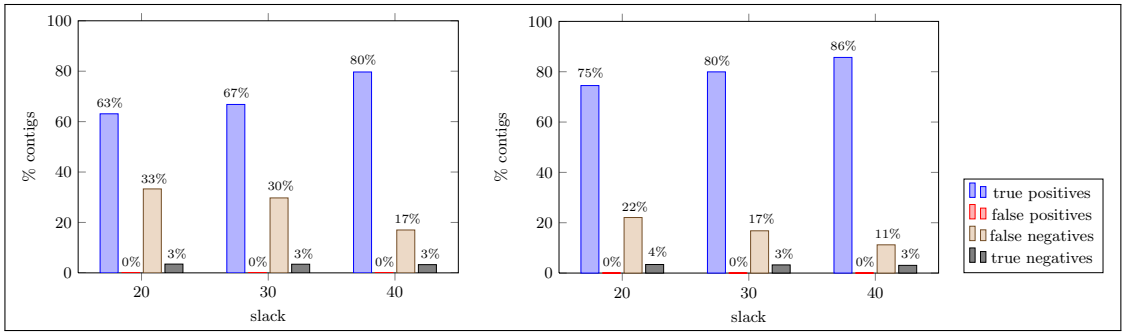


Figure 5.13: *Alteromonas macleodii* (50× and 90× input coverages). TP, FP, FN, and TN rates for different values of the slack parameter Δ.

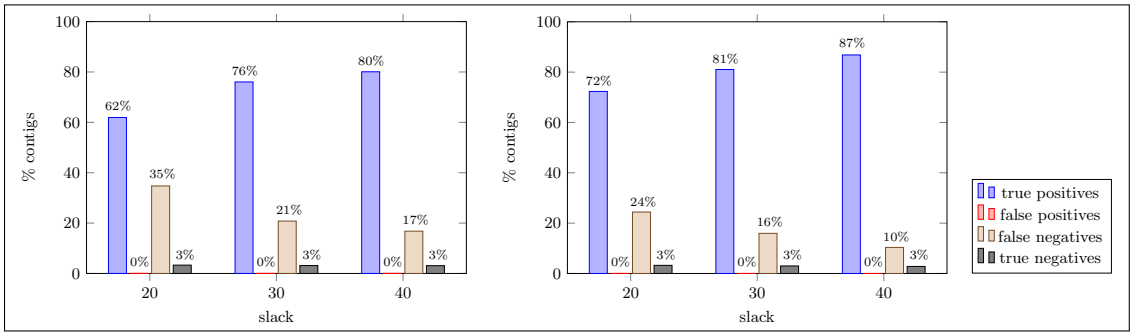


Figure 5.14: *Bacillus amyloliquefaciens* (50× and 90× input coverages). TP, FP, FN, and TN rates for different values of the slack parameter Δ.

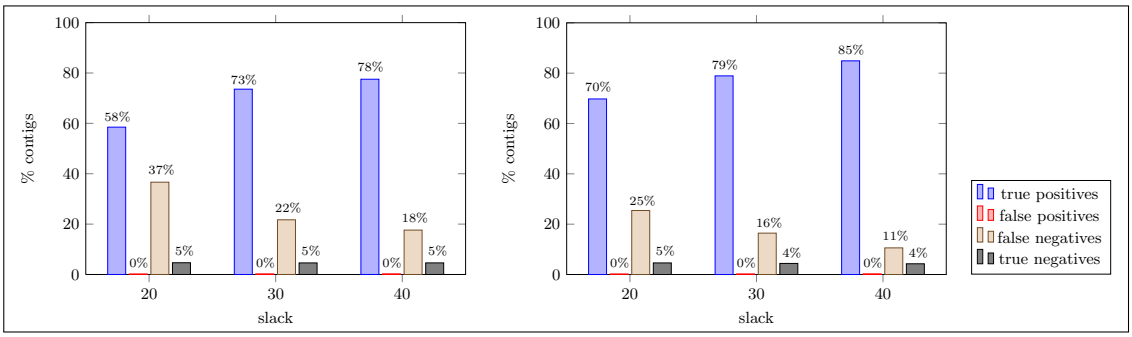


Figure 5.15: *Bacillus cereus* (50× and 90× input coverages). TP, FP, FN, and TN rates for different values of the slack parameter Δ.

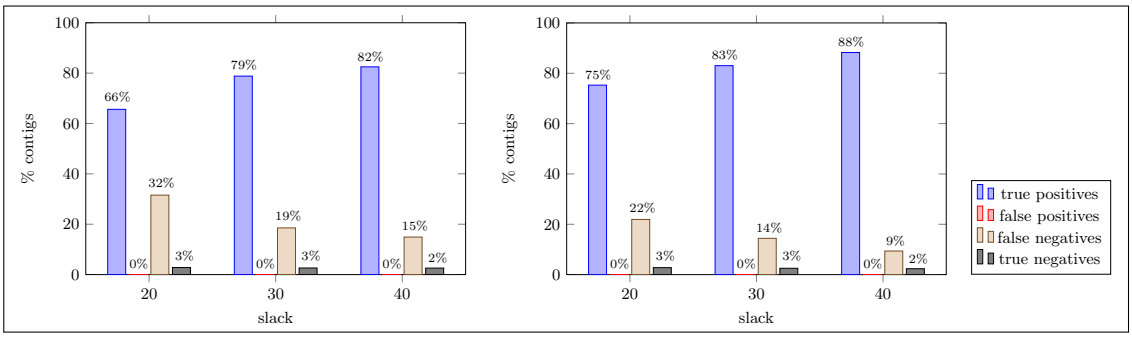


Figure 5.16: *Bordetella bronchiseptica* (50× and 90× input coverages). TP, FP, FN, and TN rates for different values of the slack parameter Δ.

5.3 Conclusions

Even though it is a well known and studied NP-complete problem, *de novo* assembly is practically solved by a large number of tools. The outbreak of NGS technologies contributed to make assembly problem more difficult. As a matter of facts, 5 years after the NGS-revolution, results achievable with NGS-assemblers and NGS-data are still far away from those achievable with Sanger-assemblers and Sanger-data. However technology improvements on the one hand, and software enhancements on the other one, are closing this gap.

In this Chapter we have proposed two strategies to partially solve *de novo* assembly. In both cases, we have used additional hypothesis in order to simplify the assembly problem and give a solution as accurate as possible.

eRGA is a pipeline that allows us to assemble genomes in presence of a related reference genome. We showed how it can be used in several practical situations, both on small genomes or on large and complex ones. eRGA's key feature is its capability to reduce the complexity of assembly problem using a reference genome as anchor.

GapFiller is a tool moving the first steps in the bioinformatics world. It is designed to locally assemble the gap between pairs of reads. GapFiller's strength lies, on the one hand, in the ability to produce an output that does not need validation, and, on the other hand, in being a *local* assembler, making it useful when studying limited regions of a genome. In a *de novo* assembly project, GapFiller is supposed to be used in two directions. It can be a preprocessing step, as the trusted contigs provided can be used as input *meta-reads* for an assembler for long reads, or, as an opposite strategy, it can be used to join the contigs produced by a *de novo* assembler in a scaffolding-like phase. GapFiller's applications to structural variations analysis include indels detection and validation; in particular, it can be used to assemble insertions occurred in a sequenced organism, with respect to a reference genome. It is of primary importance to notice how, while there is a large number of tools able to *identify* structural variations, there is no widely accepted strategy so far in order to *reconstruct* structural variations in re-sequencing projects. We believe that the localized GapFiller strategy can be used in order to "fill this gap" and move several approaches from identification to reconstruction.

6

De Novo Assembly: Validating the Puzzle

Validation techniques are of primary importance in order to assess data quality and results correctness. In particular in *de novo* assembly and read alignment data evaluation and results assessment are fundamental. However, while in sequence alignment the reference can be used to gauge reads' quality by simply counting the number of successful aligned reads, in *de novo* assembly this task is inherently more difficult.

Reads are the first and fundamental brick for all subsequent analysis. Therefore, a bias or some unexpected error in these foundations can jeopardize all analysis attempts. Therefore techniques to asses reads quality and to improve reads usage are necessary to improve the final result of every pipeline, and in particular are mandatory when performing *de novo* assembly.

Validating *de novo* assembly results is extremely challenging. As discussed in Chapter 4 Section 4.3 there are several (often contradictory) ways to validate and gauge assembly quality and correctness. Moreover there is a lack of study on how validation metrics are correlated among them and on how effective they are in gauging assembly correctness and assemblers ability to correctly reconstruct the genomic sequence. The ability to evaluate assemblers output is mandatory to design better assemblers: an assembler that optimize a score function based on a number of highly representative metrics (that can be computed on-the-fly, while creating contigs) allows to produce an assembly that is for definition optimal in relation to the optimized metrics.

In this Chapter we will focus our attention on how to improve data quality and on how to validate assemblies in the NGS context. In Section 6.1 we will discuss two techniques able to improve reads usage and gauge reads correctness. The first one (Section 6.1.1) is a filtering pipeline developed at the Institute of Applied Genomics to filter low quality and contaminated reads with the aim to improve assembly results. The second approach (Section 6.1.2) is the program *16mer-Counter* able to rapidly and efficiently extract statistics from large reads sets. 16-mers, and k-mers in general, allow to compute a large number of statistics on the sequenced reads and on the genome being sequenced. In Section 6.2 we will analyse the so called *forensics features* and improve FRC analysis described in Chapter 4 Section 4.3: employing multivariate techniques we will study relationships among different features and we will see how to select a certain number of representative features to better describe assemblers performances and, hopefully, to better design future assemblers. Moreover we will evaluate the differences between real and simulated datasets with the goal of estimating the reliability of simulation techniques.

6.1 Read Validation and Read-based Analysis

As described in Chapter 1, all sequencing technologies are affected by sequencing errors that depend on the particular instrument being used. Moreover, all NGS technologies require a certain amount of laboratory preparation, during which DNA has to be treated in a certain way (amplified, purified, *etc.*). During laboratory preparation steps, some errors may occur: environmental contamination is always a primary risk, bad reagents or kits can be accidentally used, sequencer itself can be

affected by technical problems.

When some of these problems take place we want at least to identify and hopefully correct them. We will now analyse two approaches: the first one aims at identifying and removing contaminated and low quality reads, the second, instead is able to provide as output some useful statistics which allow to estimate assembly composition, assembly length and in some cases assembly complexity.

6.1.1 rNA --filter-for-assembly

The Quality Information

Usually sequencing platforms provide every sequenced read paired with *quality information*. The typical (but not unique) format is the so called *fastq* format. A fastq file stores reads in four lines: the first one is the *header* line, it begins with '@' and contains the read name that uniquely identifies the read in the file and some other additional information. The second line is the raw sequence in letters, the part that we usually refer to as *read*. The third line contains a comment, it starts with a '+' and most of the times it does not contain any valuable information. Lastly, the fourth line contains the quality information: quality information is stored in a form of a string that has the same length of the read. The i-th quality string char stores the quality value for the i-th read char.

Sequencers do not directly output fastq files. Usually a more complex pipeline is set up, in which the images or the intensity files produced by sequencers are analysed by a *base caller* [39, 117]. A base caller analyses sequencer's images or intensities and "calls" the most plausible base, together with a confidence score (*i.e.* quality score). Quality scores are usually dubbed after the first base caller Phred [39].

A Phred quality value Q is a *integer mapping* of the probability p that the corresponding base call is incorrect (*i.e.* the probability that the letter outputted by sequencer and base caller is wrong). In particular, given p the quality score Q is defined in the following way:

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

Phred Quality Score	Probability of incorrect base call	Base call accuracy
10	1 in 10	90 %
20	1 in 100	99 %
30	1 in 1000	99.9 %
40	1 in 10000	99.99 %
50	1 in 100000	99.999 %

Table 6.1: Phred Quality Scores.

Table 6.1 shows how different Phred scores corresponds to different probabilities of incorrect base calling. In order to store quality in a compressed format, ASCII characters are used: Sanger format encodes a Phred quality score from 0 to 93 using ASCII 33 to 126, while Illumina encodes a Phred quality score from 0 to 62 using ASCII 64 to 126.

Read Filtering Pipeline

Quality information allows us to process reads and to discard low quality reads (*e.g.* reads that have mean quality length under a given threshold). Moreover, most sequencing technologies, especially the *wash-and-scan* ones (*i.e.* Illumina), are characterized by low quality bases at read's ends.

Errors at read's beginning and end are problematic both in alignment and in *de novo* assembly. Align reads with low quality bases can prevent the possibility to correctly align sequences. Aligners like MAQ and BOWTIE use quality information to give different weights to mismatches, however align parts of a read that have high probability to be wrong is counter-intuitive an reduces aligners

performances. In *de novo* assembly, especially with de Bruijn Graphs, low quality bases can cause many of the problems discussed in Section 4.2 of Chapter 4. Especially at high coverages (higher than 100×) wrong base calls can be so frequent that the assembler is not able to identify them.

Another common problem is read contamination. Sometimes, during library preparation, some contamination can occur (*i.e.* with other samples), while other times the sequenced DNA is inherently contaminated by some pathogen or organelles (*e.g.*, in plants chloroplast and mitochondria are always sequenced together with genomic DNA). This kind of contamination does not pose particular problems to alignment, especially in presence of high coverages. However, such a contamination creates problems to *de novo* assembly for at least two reasons: first contaminants can have spurious overlaps with genomic DNA thus leading to mis-assemblies, second assemblers often try to estimate important statistics from reads [166, 100] that, as a matter of facts, are supposed to belong to only one genome. As an example of how problematic contamination could be, consider that Alkan in [6] noticed that 15% of the insertions identified in the African Yoruban genome [100] belonged to contaminants and not to new genomic variations.

For these reasons it is a common practice to perform a read filtering phase prior to every analysis. In particular, this filtering phase is of primary importance in *de novo* assembly. Therefore, we will consider the *read filtering* a *de novo* assembly (pre-)step. A complete pipeline must implement the following steps:

- perform read trimming based on quality scores information. In particular remove from reads low quality tips and eventually discard reads shorter than a predefined threshold;
- align *trimmed* reads against a supposed contamination database;
- extract from the alignment all reads that do not align and store them in a suitable format (*i.e.* fastq format);

This procedure can look straightforward: we only need a software able to perform quality trimming, an aligner, and another tool able to extract unaligned reads from an alignment (usually a sam/bam file). However there are several *practical points* that make things more difficult:

- in *de novo* assembly projects datasets are composed by hundreds of millions of reads. Reads themselves pose serious storage problems: a pipeline that in a first phase saves trimmed reads, then in a second phase stores all alignments and in a third and last phase processes again alignments more than quadruplicates the used space.
- A pipeline that performs the aforementioned steps is extremely slow especially if one of the three steps is not parallelized. Moreover, as far as the second step concerns, employing a standard aligner (see Table Table 2.1 in Chapter 2) is counter-intuitive because we are interested in saving the negative information (*i.e.* reads that do not align).
- As a matter of facts, reads are always provided in pairs. The filtering pipeline has to keep this information at all the stages. Paired reads are usually stored in two files: the i^{th} read of file one is paired with the i^{th} read of file two.
- A more subtle problem is the fact that sometimes we do not know if the contamination took place. In other words we are interested in discovering a possible contamination.

In order to prevent these problems, we implemented an integrated solution that performs all the filtering steps at once allowing to run the overall procedure at high speed and without need to store intermediate files. The proposed solution is extremely useful when a contamination reference is already present. However we will see a straightforward way to check for contamination presence and in case remove it.

Due to the deep connection between this problem and string alignment we integrate the filtering pipeline inside rNA (refer to [144] and Chapter 3). However the main application of the filtering pipeline is *de novo* assembly (rNA must be launched with option `--filter-for-assembly`).

rNA filtering pipeline needs as input the raw (Illumina) reads in fastq format, the hash table of the supposed contamination pathogen(s)/organelles and other standard rNA parameters (number of mismatches, quality threshold). Almost always, reads are provided in pairs, usually divided in two files with the i-th read of file one paired to the i-th read of file two (this is the standard Illumina format). rNA filtering pipeline processes one pair of reads per time:

1. both reads are trimmed using an algorithm similar to the one proposed in [1]. First low quality bases (by default bases with Phred quality value lower than 20) at the beginning and at the end of reads are removed. After that, the mean quality value of the remaining part of the read is computed: if the length of the trimmed read and its mean quality value are higher than two user defined thresholds the read is further processed, otherwise it is discarded (by default the mean Phred quality is 20, while the minimum read length is 25). Notice that in a pair one read can have high quality and therefore proceed to step 2, while the other read can be discarded at this step.

2. Reads not discarded at step 1 are aligned against the contamination database. If one alignment is found the *pair* is declered *contaminated*. Even if only one read is aligned against the contaminant both are discarded: paired reads are sequenced from the same fragment and so must belong to the same DNA sequence.

3. Once step 1 and step 2 are over all the information to print the filtered reads is available: if both reads successfully passed trimming and contamination check their trimmed version is saved on two different files (one read per file) preserving the pairing information. If one read did not pass the quality trimming step while the other passed both the trimming and the contamination step, then its trimmed version is saved in a third file storing reads that become single ended. If one of the two reads is discovered to belong to the contamination, then both reads are discarded.

All the procedure is parallelized, no intermediate files are produced and only the minimal information (reads) is saved. This procedure is easy to use when the contaminant is known: this is true in several cases. In plants genomes, for example, chloroplast and mitochondrial DNA are often known. In other situations, there are known environmental contaminations: for example in the first assemblathon edition [37] the simulated data contained a given amount of contaminated reads belonging to *E. coli* genome. It is still unclear how to proceed if one wants to understand if a contamination took place or not.

At the Institute of Applied Genomics (IGA) we proposed and successfully used a pipeline based on the following principle: contamination is almost always a consequence of pathogens or organelles that are small organisms and therefore are sequenced at an extremely high coverage. Therefore, if one assembles a small subset of the data produced in a *de novo* assembly project, it is expected that the assembler will reconstruct large contaminant's parts. This allows us to propose the following *de novo contamination pipeline*:

1. assemble a small subset of the data being used for *de novo* assembly (*i.e.* the first produced data). This will produce an highly fragmented assembly.

2. BLAST [7] *de novo* contigs against NCBI database.

3. If the results of step 2 highlight the presence of contaminants, then for each of them obtain the sequence.

4. Build a contamination database and run the read filtering pipeline.

As an example, consider that this procedure, applied to a small dataset (a $0.5\times$ of the overall coverage) of assemblathon 1 [37] was able to identify the *E. coli* contamination.

6.1.2 Comparing Experiments and Genomes Using k-mers

As extensively explained in Chapter 4, *de novo* assembly is a difficult task: we showed that the problem is computationally hard, and that, even if many tools aim at solving it, their results are far from being satisfactory especially in NGS context.

As explained in Section 1.1.1 of Chapter 1 organisms can be classified on the basis of the number of copies of chromosomes (haploid, diploid, and polyploid) and on the fact that these copies are identical (homozygous) or different (heterozygous). Usually all this information is known before the start of the sequencing process, however, while the number of chromosomes copies is easy to gauge, the heterozygosity level is more difficult to precisely estimate. As a matter of facts, higher is the organism heterozygosity more difficult the assembly task is: the heterozygous loci are responsible for the bubbles structures in de Bruijn Graphs (see Chapter 4). Conservative assemblers break contig extension in presence of bubbles leading to an highly fragmented assembly, while non conservative ones try to merge bubbles with the danger of creating misassemblies.

It would be useful to know in advance a more precise estimation of the heterozygosity level. This estimation can be used, for example, to confirm the estimated levels and to choose an assembler able to handle specific heterozigosity levels.

In the following we will show how k-mers can be successfully used not only to confirm and eventually gauge the heterozigosity level, but also to obtain other extremely useful information on the genome content and on the data quality.

Analysing Genomes with k-mers: Tallymer

As extensively explained in Chapter 4 de Bruijn Graph based assemblers are the most promising and used assemblers in the NGS-context. They start with the counter-intuitive idea of breaking short NGS reads in even smaller pieces and combining them in a graph structure. This process allows to avoid an all-against-all comparison, to save memory, and to simplify data representation.

In general, using k-mers in place of reads is useful in many contexts. In [82] Kurtz and colleagues present the tool *tallymer*. Tallymer is based on enhanced suffix arrays [3] and it is a collection of programs for k-mer counting and indexing of large sequence sets. Tallymer builds a database of the k-mers belonging to the input sequences. For each different k-mer the information on its multiplicity is stored. Once the database is constructed, it is possible to query it with k-mers belonging to the same input set or with others belonging to different sequences. Moreover, several interesting statistics can be inferred from k-mers themselves.

Tallymer can be used on assembled sequences as well as on sets of reads. In particular the tool can be used to produce the so called "k-mer frequencies" throughout a genome sequence: given a genome G, we first build the database of all the k-mers belonging to G, then for each position i in G we plot the frequency of the k-mer $G[i..i+k-1]$. In this way we are able to classify regions as unique (frequency close to one) or repetitive (frequency higher than one). Figure 6.1 shows in blue the k-mer frequencies obtained with tallymer on the grapevine (PN40024) genome, while in green the prediction obtained with ReAS [101]. We can see how the k-mer prediction coincide with the more "biological" ReAS prediction.

Another common use of Tallymer is the *uniqueness ratio* determination. The uniqueness ratio is the ratio of k-mers occurring exactly once relative to all k-mers in the set. This information is useful to estimate the minimum *perfect* overlap between read pairs in *de novo* assembly: a large overlap can be too stringent and discard too many overlaps, while a short overlap can hamper overlap computation as a consequence of spurious overlaps. For a given k-mer, tallymer allows to compute the uniqueness ratio of the k-mer in the reads. Computing the uniqueness ratio for several values of k (see Figure 6.2(a)) one can estimate the minimum required overlap. The same analysis can be done on several assembled genomes in order to gauge the different repeat content (see Figure 6.2(b))

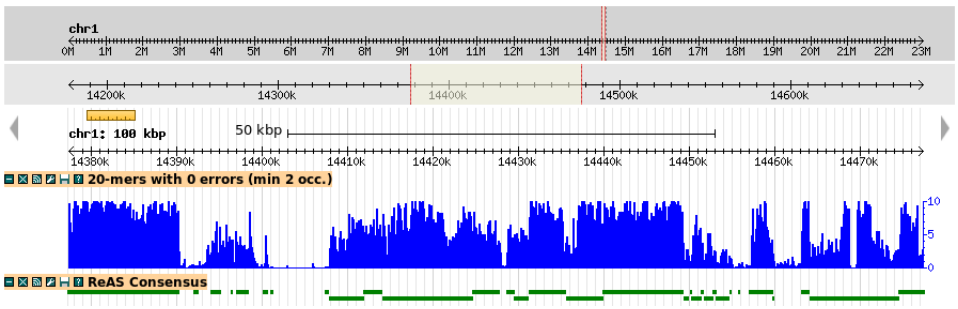
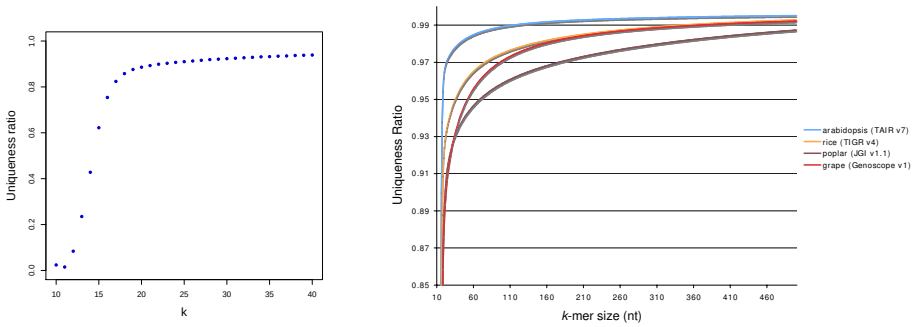


Figure 6.1: Visualization of k-mer frequencies in Grapevine (PN40024) genome. k-mer frequencies are represented in $\log_{10}$ scale



(a) k-mer uniqueness ratio on 1× coverage

(b) k-mer uniqueness ratio on four genomes

Figure 6.2: k-mer uniqueness ratio in Grapevine (PN40024) genome.

Analysing Reads with k-mers: Jellyfish and 16merCounting

Tallymer is well suited to operate on already sequenced genomes. However, analysis similar to the one just described can be useful also on reads. As a matter of facts, tallymer is not able to scale on large NGS like datasets. The major problem is the amount of resources needed (time and memory) to index all reads (hundreds of millions) in a suffix array. In order to overcame these limitations a new tool dubbed Jellyfish [110] has been proposed by Marcais and colleagues. Jellyfish is based on a multi-threaded, lock-free hash table optimized for counting k-mers up to 31 bases in length. Once all the input sequences are read it is possible to compute a set of k-mer statistics (*i.e.* single copy k-mers, multiple copy k-mers, *etc.*) or to query the database. In particular, an interesting analysis consists in counting for a given k the number of different k-mers that occur with different multiplicities. In other words count how many different k-mers occur in our data set $1, 2, \ldots$ times. This information allows to plot an histogram or a line showing the k-mer distribution as a function of their frequency (*e.g.*, see Figure 6.3).

Jellyfish hash-table has size $M = 2^l$ for some user chosen l. The key representing the k-mer is encoded as an integer in the interval $U_k = [0, 4^k - 1]$. At this point an hash function maps U_k elements into the interval $[0, M - 1]$. Jellyfish uses a open addressing hashing schema [30] in order to handle collisions: if the k-mer x is mapped to a location $i \in [0..M - 1]$ already used to count the occurrences of another k-mer $y \neq x$ then a quadratic reprobing function is used to compute another, hopefully empty, cell. If during the table population, the hash table is almost full (*i.e.*, if the load-factor is close to 1) Jellyfish saves the hash-table on disk and initializes a new one. Once all the data has been processed the hash tables can be merged.

One of the main Jellyfish's advantages is the parallelization level during the hash table population. Parallelize the hash table population is a difficult task due to concurrent writings (*i.e.* different processes that try to update the same memory location). A lock, such as POSIXs `pthread_mutex`, can serialize access to the hash table and permits its use in a multi-threaded environment. However, if such a lock is used no concurrency is achieved, and therefore there is no gain in speed in the updates of the hash table. In addition, the overhead of maintaining the lock is incurred. Jellyfish exploits CAS (Compare and Swap) assembly instruction that is present in all modern multi-core CPUs. The CAS instruction updates the value at a memory location provided that the memory location has not been modified by another thread. Technically, a CAS operation does the following three operations in an atomic fashion with respect to all of the threads: reads a memory location, compares the read value to the second parameter of the CAS instruction and if the two are equal, writes the memory location with the third parameter of the CAS instruction (see Algorithm 5). If two threads attempt to modify the same memory location at the same time,

Algorithm 5: Compare and Swap (CAS) Algorithm.

Input: *location*, *oldvalue*, *newvalue*
Output: *oldvalue* if writing failed, *newvalue* otherwise

```
1 currentvalue ← read at location;
2 if currentvalue = oldvalue then
3 |  set location to newvalue;
4 return currentvalue
```

the CAS operation can fail. In this case the CAS operation returns the value previously held at the memory location (*i.e. oldvalue*). Hence, one can determine if the CAS operation succeeded by checking that the returned value is equal to the old value. Unlike a lock that serializes the access to some shared resource, the CAS operation only detects simultaneous access to a shared memory location.

Thanks to these technicalities, Jellyfish is able to populate the hash at a very fast pace, however in presence of large datasets it creates several intermediate hash tables (every time the load factor is higher than a predefine threshold the hash is saved on secondary memory). In order to query these hash tables or to extract information about the k-mer composition one has to merge them: this step is inherently sequential and requires a large amount of time.

Draw by the need of a fast tool able to count k-mers and independently to Jellyfish, we developed a similar tool dubbed 16merCounter able to count the 16-mers from a large amount of data It is extremely interesting how the 16merCounter solution is similar to Jellyfish, although more simple. Given a 16-mer, 16merCounter converts it into its 32-bit representation. In other words, 16-mers are mapped into a number in $[0...2^{32}-1]$. In this way each 16-mer is used to access an array of size $2^{32}-1$. Every time a 16-mer is read its corresponding memory location is incremented. In order to obtain parallelization and fast running times, we used the CAS function described in Algorithm 5. 16merCounter does not need to write intermediate files on disk. The memory usage is independent from the input: 16merCounter uses 16 GB independently from the fact that the input file consists of a couple of mega bases or hundreds of giga bases. As we will see, in the NGS context it is not unusual to work with amount of data larger than 100 Gbp.

Comparing 16-mers

Both Jellyfish and 16merCounter allow to efficiently and rapidly count and query the k-mers present in a set of reads. The idea is to obtain a plot like the one in Figure 6.3 where we plot on the y axis the total number of 16-mers that occur with a certain frequency or coverage (x axis). This plot was obtained on a $\sim 245\times$ read coverage of the Verticillium genome (*Verticillium dahliae*). These plots are of primary importance to estimate two important informations: genome size and heterozygosity level.

As a matter of facts, genome size can be inferred through laboratory experiments [185]. However there are situations in which, mainly for resource's shortage, this information is not available (*i.e.* assemblathon second edition). Genome length is mandatory to estimate the genome coverage: if $\mathcal{R} = \{r_1, \ldots, r_2\}$ is the set of sequenced reads from genome G, then the read coverage C is defined as

$$C = \frac{\sum_{i=1}^{n} |r_i|}{|G|}$$

Knowing read coverage is fundamental to produce enough sequences and to set important parameters of *de novo* assembly tools. Another interesting and highly connected coverage is the k-mer coverage. The k-mer coverage is the ratio between the number of k-mers stored in the reads and the total amount of k-mers in the genome. If we suppose $\mathcal{R}$ composed by reads of the same length L, then:

$$C_k = \frac{|\mathcal{R}| * (L - k + 1)}{|G| - k + 1} =\sim \frac{|\mathcal{R}| * (L - k + 1)}{|G|}$$

k-mer plots can give us a clear idea of what the k-mer coverage is. In a perfect world, where all reads are error free and the genome is perfectly homozygous, we expect that all k-mers occur with the same frequency. However, sequencing errors, not-uniformly sequenced areas, and repeats induce a normal-like distribution like the one visible in Figure 6.3. The leftmost peak is a consequence of sequencing errors: k-mers that occur only once are likely to be a consequence of sequencing errors. The second peak (the peak of the normal distribution) gives us a clear estimation of the *k-mer coverage*.

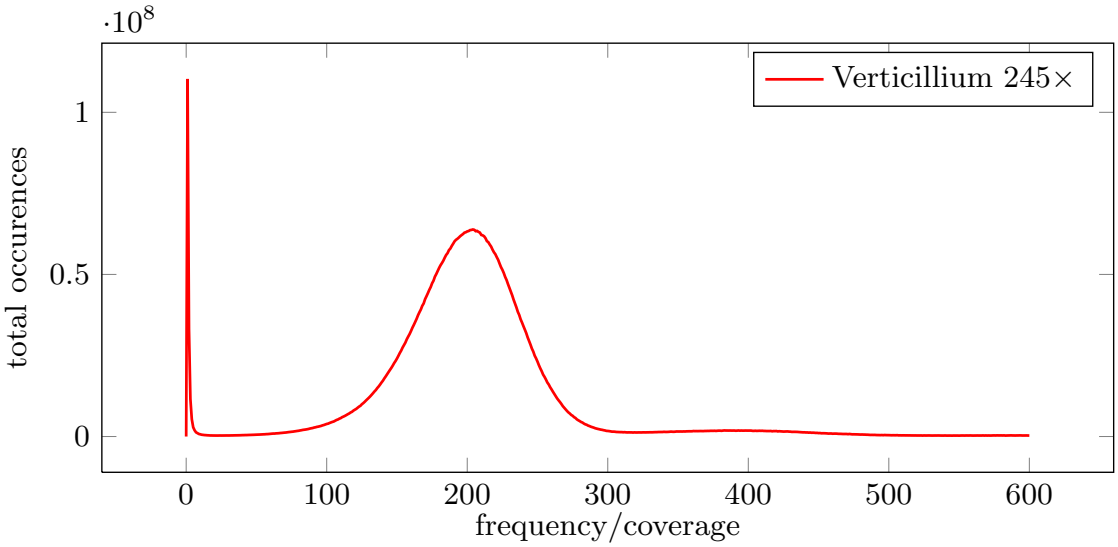


Figure 6.3: k-mer profile of an $\sim 245\times$ Vericillium read coverage composed by 100 bp long paired reads. The plot has been computed by 16merCounter.

k-mer coverage is deeply linked to read coverage and can be used to estimate genome size. In particular, if C_k is the k-mer coverage and L is the read length (*i.e.* $|r_i| = L$) then the following holds:

$$C_k = \frac{C * (L - k + 1)}{L}$$

Therefore, knowing C_k allows to compute $C = C_k * L/(L - k + 1)$ and consequently obtain the genome length.

Let $\mathcal{R} = \{r_1, \ldots, r_2\}$ be the set of sequenced reads. Let C be the unknown read coverage, C_k the estimated k-mer coverage through a k-mer plot, then the genome size $|G|$ can be computed in

the following way:

$$|G| = \frac{\sum_{i=1}^{n} |r_i|}{C} = \frac{\sum_{i=0}^{n} |r_i|}{C_k * L/(L-k+1)}$$

As an example, consider Figure 6.3. The red line represents the 16-mer profile of a set of reads belonging to the Verticillium genome (*Verticillium dahliae*). We will show that the analysis describe so far allow us to estimate the genome length. The total length of the sequenced reads is 7.544.197.520 bp and the mean read length is 100 bp. The k-mer coverage inferable from Figure 6.3 is $\sim 207\times$. Therefore the estimated read coverage is

$$C = \frac{C_k * L}{(L-k+1)} = \frac{207 * 100}{100 - 16 + 1} = 243.5\times$$

and consequently the estimated genome length is

$$|G_{verticillium}| = \frac{\sum_{i=1}^{n} |r_i|}{C} = \frac{7.544.197.520}{243.5} = 30.482.871,4$$

Verticillium real genome size is 30.299.901 bp, our method induced an overestimation of approximatively 1% ($\sim$ 1.8 Mbp). It is clear that the error is not negligible, but due to the fact that this estimation comes for free we believe it can be effectively used when the genome size is unknown (*e.g.* assemblathon second edition).

k-mer profiles are important in order to estimate heteozigosity levels. Each SNP affects k different k-mers that overlap the SNP. Let s be the SNP rate throughout the genome G. Usually $s << 1$, moreover we can work under the realistic hypothesis that two or more SNPs unlikely occur within a given k-mer. Then $|G| * s * k$ positions of the genome will affect k-mer variance due to heterozygosity. These particular sequences are expected to occur with average depth $C_k/2$ instead of C_k. Therefore, if the sequenced genome is affected by heterozygosity and the coverage is high enough we expected to see a second peak in the k-mer profile at half the k-mer coverage.

This situation is clear in Figure 6.4 where we computed the k-mer plots for three different values of k (16 with 16merCounter, 20, and 24 with Jellyfish) on a 89$\times$ read coverage of Sangiovese grapevine variety genome (known to be highly heterozygous). We can see how in all cases the rightmost peak is in the expected position, and at half of the k-mer coverage another peak is visible. We can appreciate how a larger k allows to better visualize the presence of the second peak. This fact highlights a lack of 16merCounter algorithms that is limited to 16-mers.

From Figure 6.5 we can see the plots of two different coverages obtained from two different libraries (*i.e* two different sets of DNA fragments) of *Populus nigra* genome. The 73$\times$ coverage was the first one to be produced. Such dataset was filtered for quality and checked for contamination without demonstrating any particular problem. Also alignment on a closely related genome (*Populus trichocarpa* genome) did not suggested any particular bias. Encouraged by these results we proceed to assemble the dataset, however at this stage we encountered several problems. The most important one was the fact that only 40% of reads used to assemble the data aligned on the contigs.

To further explore this situation we plotted the 16-mer coverage and we noticed the unexpected shape of the plot: *Populus nigra* genome is expected to be highly heterozygous, but the green line of Figure 6.5 suggests an absolute absence of heterozigosity. Skeptical about the dataset quality we produce and sequenced another library with the hope to obtain a not biased dataset. The 16-mer coverage of the new dataset is represented by the blue line in Figure 6.5. The blue plot has two peaks at the expected positions and shows an interesting third peak on the right. This third peak is probably a consequence of a known recent duplication event [172]. The reasons of the failure of the first library is still unknown, but is most likely the consequence of some bad reagent used at some step of the library preparation.

The example of Figure 6.5 shows how k-mer plots can be used to estimate the overall data quality. The fact that the k-mer plot has a totally unexpected shape is an extremely good indicator that the data is unreliable.

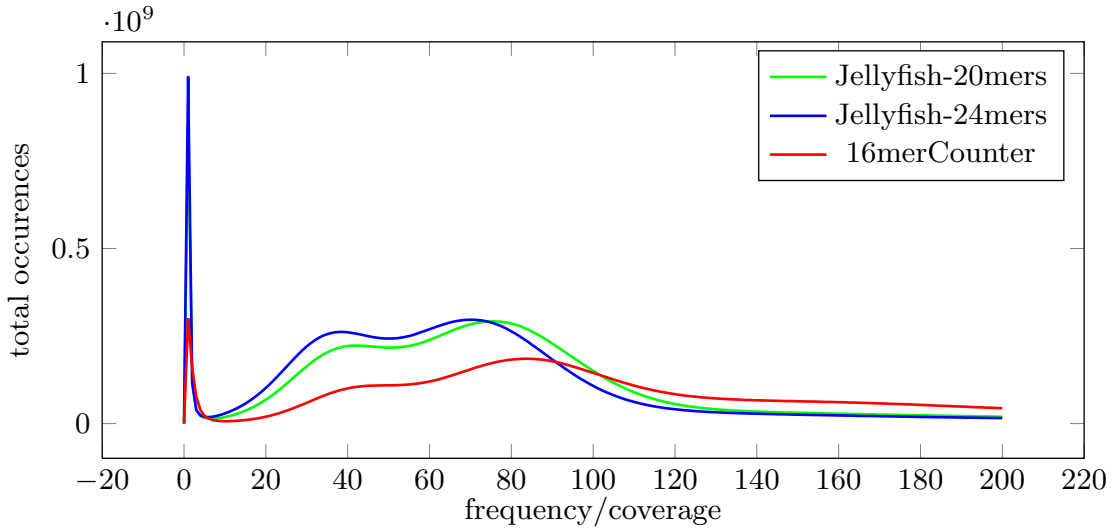


Figure 6.4: k-mer profile of an 89× Sangiovese coverage composed by 100 bp long paired reads. We draw the k-mer profile using 16merCounter (red), and Jellyfish with k equal to 20 (green) and 24 (blue).

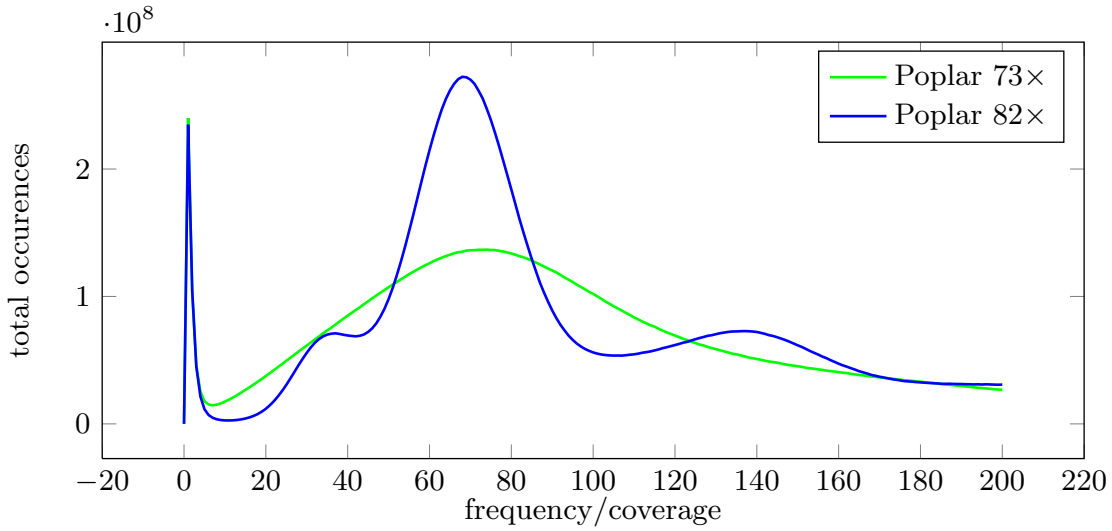


Figure 6.5: Both 16-mer profiles have been computed on an high Illumina read coverage of the Poplar genome (Poli variety belonging to *Populus nigra*). Data used to plot green and blue curves belonged to two different libraries (*i.e.* collection of DNA fragments). The green library was clearly affected by low complexity problems, while in the second a more careful preparation avoided this bias.

In Figure 6.6 we can appreciate the 16-mer plot of a 50× coverage of the spruce genome that is highly repetitive. The plot has been realized with 16merCounter: even if the two peaks are visible, the short k-mer length causes the slow decreasing shape on the right side. A larger k-mer (20 or 24) avoids this shape as the fact that 20-mers and 24-mers are more likely to occur in single copy. However, Jellyfish with k-mer size equal to 24 required one week of computation in order to compute the profile of a 30× coverage. Most of the computational time spent by Jellyfish has been used to merge the large number of intermediate hash tables. 16merCounter, despite limited

to 16-mers, was able to compute the 16-mer distribution in less than 9 hours on the same machine used by Jellyfish.

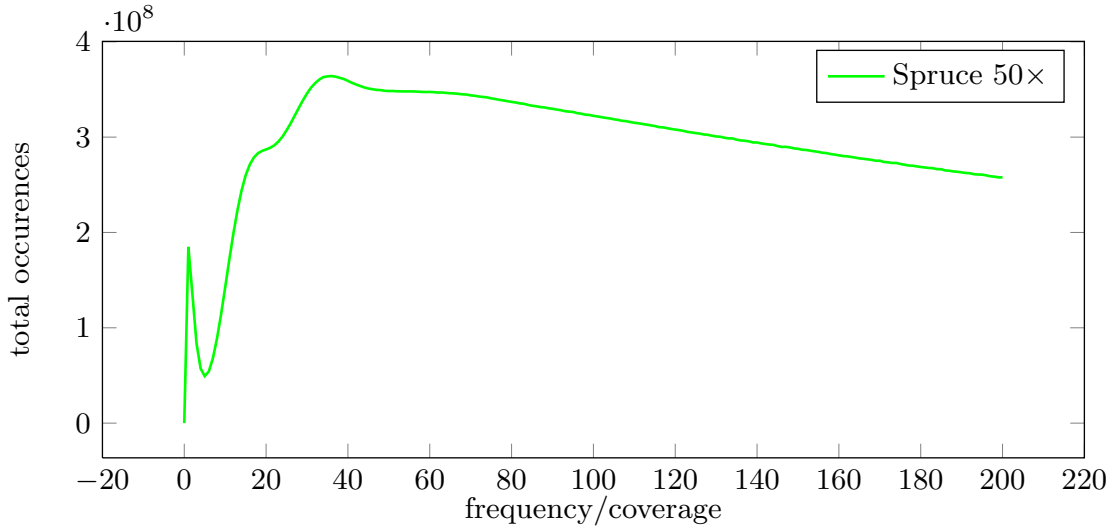


Figure 6.6: Spruce genome has length ∼ 20 Gbp. Therefore a 50× coverage is composed by 1000 Gbp. 16merCounter needed 8 hours and 16 GB RAM. Jellyfish on a reduced dataset of 30× required the same amount of RAM and required a week of computation (mainly merging tables).

6.2 Assembly Forensics: Gauging the Features

In Section 4.3.2 of Chapter 4 we introduced the forensics analysis proposed by Phillippy and colleagues in [143]. We saw how using the *amosvalidate* pipeline we cannot only validate assemblies, but also gauge assembler performances using the Feature Response Curve (FRC) developed by Narzisi and Mishra [130].

As explained in Section 4.3.2 *de novo* assembly is based on the double-barreled shotgun process, therefore the layout of reads (*i.e.* how reads are used in the assembly and how pairs are ordered) must be consistent with the characteristics of the shotgun sequencing process. In particular Phillippy and colleagues detected various *features* that predict or that witness misassemblies (*forensics features*). Among the most important ones, they noticed that paired reads found at unexpected distance or with unexpected orientation are likely to highlight the presence of insertion/deletion and translocation events. In a similar way regions lacking paired reads or regions with an unexpected read coverage (too high or too low) are indicative of assembly problems. Another important feature is the k-mer distribution, computed as the ratio between the k-mer coverage of the reads and the coverage of the assembly along the assembly sequence itself. Phillippy also noticed that reads not used by assemblers (*i.e.*, *leftovers*) can be helpful in retrieving errors in the consensus generation. For a more complete discussion on forensics features and the way in which they are computed refer to Section 4.3.2 of Chapter 4.

Forensics features allow to count the number of suspicious position in an assembly, however it is not clear how the simple feature counting can be used to compare different assemblies and/or different assemblers. Narzisi and Mishra introduce the Feature Response Curve for this purpose. After running *amosvalidate*, each contig is assigned the number of features that correspond to putative misassemblies. For a fixed feature threshold w, the contigs are sorted by length and, starting from the longest, we consider only those whose total sum of features is $\leq w$. For this set of contigs, the corresponding approximate genome coverage is computed, leading to a single

point of the Feature-Response Curve (FRC). FRC allows to easily compare different assemblies by simply plotting their respective FRCs (refer to Section 4.3.2 of Chapter 4 for a detailed discussion).

Despite the advantages of such methods over the plethora of standard validation metrics (Section 4.3.1) assembly forensics, and therefore FRC, have some problems that need to be addressed with a deep analysis. The *amosvalidate* pipeline computes a total of 12 features (*i.e.*, forensics features) that are listed below:

1. `BREAKPOINT`: Points in the assembly where leftover reads partially align;
2. `COMPRESSION`: Area representing a possible repeat collapse;
3. `STRETCH`: Area representing a possible repeat expansion;
4. `LOW_GOOD_CVG`: Area composed of paired reads at the right distance and with the right orientation but at low coverage;
5. `HIGH_NORMAL_CVG`: Area composed of normal oriented reads but at high coverage;
6. `HIGH_LINKING_CVG`: Area composed of reads with associated mates in another scaffold;
7. `HIGH_SPANNING_CVG`: Area composed of reads with associated mates in another contig;
8. `HIGH_OUTIE_CVG`: Area composed of incorrectly oriented mates;
9. `HIGH_SINGLEMATE_CVG`: Area composed of single reads (mate not present anywhere);
10. `HIGH_READ_COVERAGE`: Region in assembly with unexpectedly high local read coverage;
11. `HIGH_SNP`: SNP with high coverage;
12. `KMER_COV`: Problematic k-mer distribution.

The main problem is to understand the relationship among different features and *how* and *if* they correlate. A standard consideration [143] is the fact that positions in the assembly with many different features are likely to represent misassemblies. However, results from feature analysis are strongly dependent on how the features are combined, especially when the relationships among features are ignored. For example, an area with high k-mer coverage is likely to contain many paired read features. Another important problem of forensics features is their lack in specificity opposed to their high sensitivity [143].

An interesting point is the possibility to reduce the feature space and concentrate the analysis to only a handful of meaningful features. Even more interesting, it would be desirable to use a linear combination of few such features to create newer and better set of synthetic and meaningful features.

There is a lack in the study of how features are related among each other and how they represent assembly correctness. As an example consider the N50: despite being one of the most used features there is no evidence, to the best of our knowledge, that demonstrates the connection between such a measure and assembly quality.

Therefore we decided to concentrate our efforts on a deep study of available features (in particular of the forensics features) in order to better understand their capabilities in predicting assembly correctness. As a side effect of our analysis, we were able to evaluate the limits of simulation based tests, often used as a proof of assembler's abilities. One of our main objective was to improve the Feature Response Curve (FRC). It would be desirable to plot the FRC on a minimal subset of the most important features or on a small number of synthetic features capable of capturing the most important information (*i.e.*, variation).

In order to attain our objectives we used unsupervised learning methods to *extract* and *select* a subset of relevant features to understand their inter-relationships. We obtained several de-novo assemblies (more than 300 assemblies) by assembling different genomes (45 genomes) with a wide range of assemblers (10 assemblers). For each assembly we extracted the 11 forensics features

computed by *amosvalidate* (`HIGH_LINKING_CVG` and `HIGH_SPANNING_CVG` have been collapsed into a single feature) and we used PCA and ICA (Principal/Independent Component Analyses, respectively) to extract and select a set of synthetic features and a set of highly informative features, respectively. Moreover we explored the relationship among the 11 forensics features and two additional commonly used metrics: N50 and number of contigs (`NUM_CONTIG`).

When counting the number of features on a contig we used the following approach: single point features (`SNP` or `BREACKPOINT`) are counted as a single feature, while features that affect a contig's subsequences (*e.g.*, `KMER_COV`) of length l account for $\lceil l/w \rceil$ features, with w assuming a predefined threshold. In all our experiments w was kept fixed at 1 Kbp.

We also studied the relationships among features in the case of long (*i.e.*, Sanger-like) reads as well as short (*i.e.*, Illumina-like) reads. Moreover, in each case we worked with both real and simulated datasets in order to quantify the differences between the features obtained from the two kinds of data.

6.2.1 Multivariate Analysis

In order to explore and analyse relationships among features we made extensive use of multivariate analysis. One of the main problems of data mining and pattern recognition is model selection that aims at avoiding overfitting throughout model parsimony, which often involves dimensionality (or degrees-of-freedom) reduction. The key idea is to reduce the dimensionality of the data set by sub-selecting only those features, which jointly describe the most important aspects of the data. Furthermore, dimensionality reduction allows a better understanding of the problem by focusing on the important *components*, and in highlighting hidden relationships among the variables. Recently, research focusing on dimensionality reduction has seen a renewed interest as their importance in both supervised and unsupervised learning has become obvious. Techniques based on PCA, ICA, shrinkage, Bayesian variable selection, large margin classifiers, L_1 metrics, regularization, maximum-entropy, minimum description length, Kolmogorov complexity, etc. are all examples of Occam's razor, trimming away unnecessary complexity.

In the context of sequence metrics, our interests lie primarily in unsupervised learning approaches. Two main techniques can be used to reduce the dimensionality of a problem: feature extraction and feature selection. Feature extraction techniques combine available features into a new reduced set of *synthetic* features, representing the most important information. Among the techniques mostly used involving linear models, the following three dominate: Principal Component Analysis (PCA) [73], Independent Component Analysis (ICA) [64], and Multilinear Subspace Learning (MSL) [106].

Feature selection techniques focus on finding a minimal subset of relevant features containing the most important information in the dataset. Usually these methods try to select a subset of features that maximizes correlation or mutual information. Since this problem in general can be intractable, practical approaches are based on greedy methods that iteratively evaluate and increment a candidate subset of features [67]. Other common methods are based on Sparse Support Vector Machines (SSVM) [14], and PCA and ICA techniques, as discussed earlier [18, 146].

We chose to perform PCA in order to extract the most important components capable of succinctly describing assembly correctness and quality. PCA components emphasize the connections among features and their correlations. Moreover, the PCA results can be used to understand redundancy in a given set of features. Once PCA has established a high degree of redundancy, we can use ICA to select the most important features in order to parsimoniously use only those to compare assembly performances.

PCA: Principal Component Analysis

Principal Component Analysis (PCA) is a popular multivariate statistical technique with many applications to a large number of disparate scientific disciplines [73]. It finds a set of new variables (Principal Components) that account for most of the variance in the observed variables. A principal component is a linear combination of optimally weighted observed variables.

PCA analyses a matrix (*i.e.*, a table), whose rows correspond to observations and columns to variables that describe the observations. In our case, the observations are *de novo* assemblies of different genomes performed with several assemblers, while columns are the features describing the quality and correctness of the assemblies. PCA extracts the important information from the data table and compresses the size of the table by keeping only the important information, thus simplifying the description of the data set. New variables, each one a linear combination of the original variables and called principal components (PCs), are computed in order to achieve these desiderata. The first PC is required to achieve the largest variance reduction (*i.e.*, the component that "describes" the largest part of the variance of the data set). The second component is computed under the constraint of being orthogonal to the first component, while accounting for the largest portion of the remaining variance. The subsequent components are computed with similar criteria.

PCs are described by eigen-vectors that represent the linear combination over all the original variables (*i.e.*, features). Eigen-vectors are ordered according to a monotonically decreasing order of eigen-values. The eigen-vector with the largest eigen-value explains the main source of variance, with the remaining ones explaining successively smaller sources of variance. PCA on a dataset described by p variables returns p eigen-vectors (*i.e.*, p PCs). However, we are interested in keeping only those PCs that capture as much of the important information in the data as possible. A widely used rule of the thumb is to fix a variance threshold, which determines the eigen-vectors that can be safely discarded (*i.e.*, retain only those PCs that account for a certain amount of variance). A practically used heuristic value for variance threshold is often taken to be 80%. A more robust method is based on random matrix theory (RMT). By fitting the Marčenko-Pastur distribution [71] to the empirical density distribution of the eigen-values, one can determine the less informative eigen vectors and discard them.

ICA: Independent Component Analysis

Independent Component Analysis (ICA) is a signal processing technique that was originally devised to solve the *blind source separation problem*. ICA represents features as a linear combination of Independent Components [65]. Independent Components (ICs) have been used to select the most independent (*i.e.*, the most important) features [104].

ICA differs from other methods as it looks for components that are both statistically independent, and yet, non-gaussian (*e.g.*, has non-vanishing high order moments – beyond mean and variance – such as the fourth-order moment, represented by kurtosis). Given a set of observations as a vector of random variables X, ICA estimates the Independent Components S by solving the equation $X = AS$ with A being the so-called *mixing matrix*. ICs represent linear combinations of features expressing maximal independence in the data. We followed the method described in [128] to select the most informative ICs by picking those with highest kurtosis (*i.e.*, the 4^{th} order cumulant). The underlying intuition is that higher is the kurtosis of an IC more "peaked" is its distribution, making it deviate further than what could be expected from central limit theorem (CLT). After selecting the ICs with kurtosis values in the top 80% of the kurtosis distribution, we singled out from each IC that feature, which contributed the highest in the linear combination.

6.2.2 Experiments Creation

We worked both with real and simulated datasets. We concentrated our attention on small bacterial and viral genomes for several reasons: first, the sample of assembled bacterial genomes is sizable enough to satisfy our aims; second, bacterial genomes are not diploid; and last but not least, the in silico experiments can be conducted with an affordable amount of resources (primarily, the computation time).

Half of the experiments have been performed on Sanger like data. Although Sanger sequencing has been replaced by NGS approaches, we consider this experiment of primary importance for the following statistical analysis, especially, if the features should have a universal interpretation. Sanger sequencing is a well-known and stable method, used for more than 20 years, and the tools

used to cope with Sanger data have been tested in a wide variety of situations. Thus, long reads present a useful benchmark in order to assess results. The utility of the long-read analysis is likely to become more relevant, as all available NGS technologies have been increasing the read lengths, steadily approaching Sanger reads in length.

Long Reads

We downloaded from NCBI public `ftp`, data from 21 completed sequencing projects, consisting of reads, quality and ancillary data (paired read information, vector trimming, etc.). This dataset is summarized in Table 6.2.

	Genome	Length	# reads	Avg lgth	tot length	cov
1	`Alcanivorax`	3789834	39044	1080	42177431	11.13
2	`Alteromonas macleodii`	4448980	43878	1007	44209050	9.94
3	`Bacillus anthracis`	5227293	125879	854	107563457	20.58
4	`Bacillus cereus`	5269030	68503	1071	73375574	13.93
5	`Bifidobacterium dentium`	2636367	28240	757	21394408	8.12
6	`Bordetella bronchiseptica`	5339179	55895	946	52909812	9.91
7	`Bbradyrhizobium`	8264687	89675	1018	91346484	11.05
8	`Brucella Suis`	3315173	36275	895	32499069	9.8
9	`Burkholderia mallei`	5742303	101634	1008	102506338	17.85
10	`Candidatus korarchaeum`	1590757	30168	1048	31625328	19.88
11	`Escherichia coli`	5572075	58534	1119	65538509	11.76
12	`Lactobacillus gasseri`	2011295	42477	882	37495317	18.64
13	`Mesoplasma florum`	793224	86566	788	68278119	86.08
14	`Shewanella oneidensis`	4969803	69499	752	52307472	10.53
15	`Staphylococcus apidermidis`	2616530	57997	900	52208201	19.95
16	`Staphylococcus aureus`	2809421	50035	818	40937267	14.57
17	`Thioalkalivibrio`	3464554	28458	940	26766873	7.73
18	`Vibrio cholerae b33`	4154698	30570	1075	32865241	7.91
19	`West nile virus`	11029	3148	937	2952302	267.69
20	`Wolbachia sp`	1267782	26816	981	26332465	20.77
21	`Yersinia pestis biovar`	4681648	73065	989	72291428	15.44

Table 6.2: Summary of the 21 Sanger project downloaded from NCBI.

The organisms' genome lengths varied from $\sim$ 11 Kbp (*West Nile virus*) to $\sim$ 8 Mbp (*bradyrhizobium sp. btai1*). All the 21 datasets have been assembled using 5 different *de novo* assemblers for long reads: CABOG [120], MINIMUS [169], PCAP [62], SUTTA [129], TIGR [170] for a total of 105 assemblies. Only 84 (CABOG 20, MINIMUS 15, PCAP 20, SUTTA 15, TIGR 14) were used in the subsequent analysis. We discarded 21 assemblies for two reasons: the assembler returned with error (missing data) and the assembly was clearly of bad quality (data outlier). Confronted with the first situation, we tried to resolve the problem by further manual interventions, but more often than not, we failed to understand the source of error, while in few other cases, usually the problem was due to bad format conversions (*e.g.*, CABOG was the only assembler unable to parse the ancillary file provided as input for *Staphylococcus aureus* dataset). In the second situation, we noticed that on some datasets (for example, *Bradyrhizobium sp. btai1*) some assemblers produced much worse results than others (TIGR produced 19680 contigs while CABOG 72). Since both PCA and ICA were adversely affected by the presence of such outliers, which we assumed to be due to a wrong format conversion step, we disregarded these data points. All the assemblers have been tested using the default parameters, as provided by the implementers.

Another 20 bacterial organisms were selected to generate 20 simulated coverages (see Table 6.3). We used MetaSim [151] to generate a 12$\times$ coverage composed of paired reads of mean size 800 bp with insert sizes of length 3 Kbp and 10 Kbp (forming respectively a 10$\times$ and a

2× coverage) for each genome. These 20 sets have been assembled using CABOG, MINIMUS and SUTTA with default parameters, while PCAP has been used after relaxing some parameters ("`-d 1000 -l 50 -s 2000`") in order to obtain results comparable to the other three assemblers. We did not use TIGR assembler in order to avoid its poor assembly results, which could not be corrected even after changing various parameters. Of the 80 assemblies produced, 4 failed. The 76 remaining assemblies did not create outliers [1].

			Long Reads		Short Reads	
	Name	Length	read lgth	cov	read lgth	cov
1	`Alcanivorax borkumensis`	3120143	800	12	100	80
2	`Alteromonas macleodii`	4448980	800	12	100	80
3	`Bacillus amyloliquefaciens`	3918589	800	12	100	80
4	`Bacillus cereus`	5269030	800	12	100	80
5	`Bordetella bronchiseptica`	5339179	800	12	100	80
6	`Brucella suis`	3315173	800	12	100	80
7	`Burkholderia mallei NCTC`	5742303	800	12	100	80
8	`Campylobacter jejuni`	1777831	800	12	100	80
9	`Chlamydia trachomatis`	1038842	800	12	100	80
10	`Chlorobium tepidum`	2154946	800	12	100	80
11	`Dehalococcoides`	1413462	800	12	100	80
12	`Geobacter metallireducens`	3997420	800	12	100	80
13	`Mesoplasma florum`	793224	800	12	100	80
14	`Shewanella oneidensis`	5131416	800	12	100	80
15	`Staphylococcus aureus COL`	2813862	800	12	100	80
16	`Staphylococcus aureus JH1`	2906507	800	12	100	80
17	`Staphylococcus epidermidis`	2643840	800	12	100	80
18	`Thioalkalivibrio sulfidophilus`	3464554	800	12	100	80
19	`Wolbachia`	1267782	800	12	100	80
20	`Yersinia pestis`	4600755	800	12	100	80

Table 6.3: Summary of the 20 Reference Genomes used for simulation purpose.

For each assembly we used 11 forensics features (`HIGH_LINKING_CVG` and `HIGH_SPANNING_CVG` have been collapsed in a single feature) and inserted them in a row in the experiment table, which has a row for each observation (*i.e.*, assembly) and a column for each feature. For the PCA analysis we also added two more columns: N50 and number of contigs (`NUM_CONTIG`).

Short Reads

We also performed a similar set of experiments for short reads. *De novo* assemblers for short reads have appeared only very recently and apart from the *multifasta* file containing all the computed contigs, no standard output format is provided. A particularly useful format used by all the Sanger based assemblers (mandatory for *amosvalidate*) is the `afg` format. An `afg` file is a text-based file that contains all the information related to reads, paired reads and contigs (in particular, the layout information, *i.e.*, where a read has been used in generating a consensus). This file is fundamental in order to run *amosvalidate* and hence, to obtain the forensics features.

Velvet [188], SUTTA [129] and, Ray [17] natively produced such files. In order to produce such files with other popular assemblers like ABySS [166] and SOAPdenovo [93], we found no solutions apart from mapping the reads back to the contigs and then use a program provided by the ABySS suite (*abyss2afg*) to obtain the `afg` file. Obviously, layouts created in such a way are unlikely to coincide with real ones. In particular, reads that fall in repeated regions are likely to be wrongly assigned, thus producing a "wrong" layout. However, since this was the only available method to

[1] This can be seen as the first significant difference between analyses involving real and simulated datasets.

obtain the layout files for ABySS and SOAPdenovo, we used these layouts for our analysis. This situation stresses the need for a standard assembly output that is able at the same time to retain all important informations needed for the validation step but also to store all this information in a compressed way.

Another stumbling block, we faced with short reads dataset, involved a lack of a large enough number of genomes that have been assembled, *i.e.*, a paucity of a repository of short reads datasets for genomes. Data loaded on the Short Read Archive is obtained through different pipelines and different protocols, making it really hard to obtain several assemblies from different assemblers. A similar problem concerns the read length. Over the last two years Illumina reads have grown in length from 36 bp to 150 bp, but often assemblers are optimized only for certain ranges of read lengths. Moreover, almost always raw reads were needed to be trimmed and/or filtered to remove contamination, which invariably improved the final results.

Notwithstanding all these practical difficulties, we decided to assemble four real datasets: *Escherichia coli* (SRX000429) composed of paired reads of length 36 bp and insert size of 200 bp, *Chlamydia trachomatis* (ERX012723) composed of paired reads of length 54 bp and insert size of length 250 bp, *Staphylococcus aureus ST239* (ERX012594) composed of paired reads of length 75 bp and insert size of 270 bp and, *Yersinia pestis KIM D27* (SRX048908) composed of reads of length 100 bp and insert size of length 300 bp. Datasets are described in Table 6.4.

	Name	Length	# reads	Avg lgth	cov	ins lgth
1	`Escherichia coli`	4639675	20816447	36	161.52	200
2	`Chlamydia trachomatis`	1042579	8100845	54	419.58	243
3	`Staphylococcus aureus ST239`	2906507	5307429	75.73	138.29	268
4	`Yersinia pestis KIM`	4600755	2311795	100	50.25	300

Table 6.4: Summary of the 4 Illumina projects downloaded from NCBI.

In order to achieve a number of experiments that allowed PCA and ICA to yield statistically significant results, we assembled for each genome different random coverages ranging between 30× and 130×. In order to assess parameters, for each genome, for each coverage and for each assembler, we varied the most important parameters and retained the results with the best trade-offs between N50 and number of contigs. We performed 105 assemblies and kept 82 of them (20 ABySS, 17 Ray, 20 SOAP, 9 SUTTA and, 16 VELVET) after discarding outliers.

The same 20 genomes used to obtain the simulated datasets for Sanger were also used for Illumina. For each of the 20 genomes, we used SimSeq, the read generator used for Assemblathon 1 (`www.assemblathon.org`), to produce an *in silico* 80× coverage formed by paired reads of length 100 bp and insert size of 600 bp. For these experiments we used ABYSS, RAY, SOAP, and VELVET. The most important parameter to set in these assemblers was the *k-mer* size, *i.e.*, the size of the word used to compute overlaps. We noticed that by fixing this parameter to 55 bp all the assemblers were able to achieve good and comparable results. We did not use SUTTA because the publicly available version was mainly designed for ultra-short reads (*i.e.* reads of length 36-55).

We produced two tables, one for the real data and one for the simulated data. For each assembly we computed 10 forensics features with *amosvalidate* (at present, `BREAKPOINT` feature could not be computed since only SUTTA and VELVET return the unused reads). In the PCA analysis, we also added to those features the N50 and the number of contigs.

6.2.3 Results

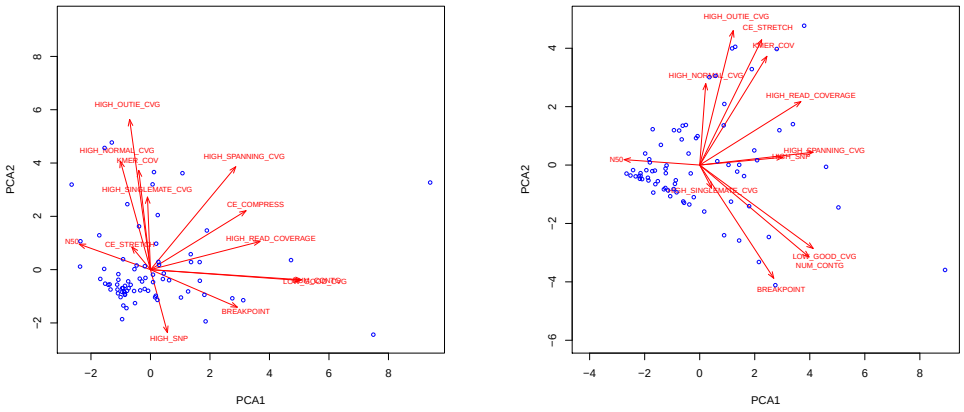
We used PCA in order to extract the most important Principal Components (PCs) and analyzed whether and how the features are correlated. In order to choose how many PCs to keep, we used random matrix theory (RMT) as suggested in [71]. In the following, we describe the results achieved with PCA on the different datasets. Moreover, we present the results achieved with ICA, in particular we show how the FRC can be used on a small subset of features to better describe the

behavior of the different assemblers. To evaluate the results obtained from this analysis, we used the reference genome in order to compute the number of *real* misassemblies by aligning the *de novo* contigs. We used *dnadiff* [143] in order to compute misassembly. When parsing *dnadiff* results, we ignored small differences like SNPs and short indels, and disregarded breakpoints occurring within the first 10 bp of a contig. This kind of analysis gave us the possibility to gauge how FRC represents the relationship between different assemblies/assemblers. In particular we could evaluate if the restriction to the ICA-feature space can improve our capability to predict the assembly quality.

PCA was performed on the extended features space (forensics features plus `N50` and `NUM_CONTIG`) while we restricted the analysis only to the forensics features for ICA. We operated in such way as to understand how two common metrics used to judge assemblies are related to the other features and to gauge the excess-dimensionality of the feature space.

Long Reads Results

We performed PCA on the real as well as on the simulated dataset. In Figures 6.7(a) and 6.7(b), we plotted the first component versus the second in order to have a graphical representation of how the assemblies are separated by the first two PCs.

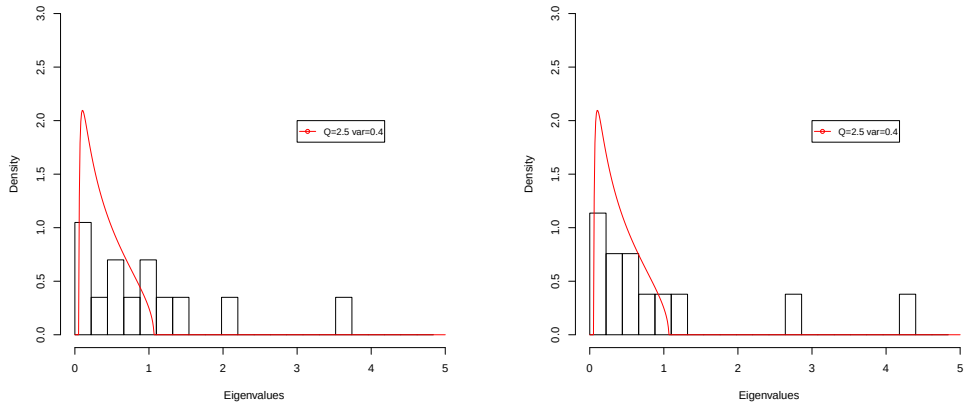


(a) Real datasets

(b) Simulated datasets

Figure 6.7: First PC versus Second PC: Long Reads Datasets. The plots in Figures 6.7(a) and 6.7(b) show the results of plotting the first principal component against the second. The blue dots represent the assemblies.

In the real dataset, 6 PCs are necessary to represent at least 80% of the total variance, while in the simulated dataset only 5 PCs are necessary to represent the same amount of variance. A more careful analysis performed by fitting the Marčenko-Pastur distribution to the empirical density distribution of the eigen-values (see Figure 6.8), showed how to prune the eigen-vectors with eigen-values lower than one. This more precise analysis tells us that we need five and four PCs to fully describe the real and the simulated dataset, respectively. Both these methods also suggest how the feature space (11 forensics features plus N50 and `NUM_CONTIG`) is "over-dimensionated" and what can be eliminated without loss of valuable information. Examining the first eigen-vector (*i.e.*, first PC) of the real dataset closely, we see that the most important features are `LOW_GOOD_CVG` and `NUM_CONTIG`. The other positive contributing features are connected to the presence of areas with no uniform coverage. Surprisingly the acclaimed N50 metric not only lacked a large coefficient, but instead exhibited negative correlations with the others. This result suggests that high N50 values are simply a consequence of mis-assemblies and due to the fact that many assemblers try



(a) Real datasets (b) Simulated datasets

Figure 6.8: Marčenko-Pastur Distribution: Long Reads Datasets. We found the Marčenko-Pastur that best fits the eigen-value distribution. All eigen-vectors with eigen-values under the Marčenko-Pastur function are considered non informative. Figure 6.8(a) shows the results obtained on real long read, conversely Figure 6.8(b) shows the results obtained on simulated long reads.

aggressively to merge as many sequences as possible. In the second component, the main source of variation among assemblies with a large number of features is due to mis-assembled repeats (`HIGH_READ_COVERAGE`, `K_MER_COV`, `HIGH_OUTIE_CVG`, and `HIGH_SPANNING_CVG`), a low number of contigs and SNPs. The first three components account for the 55% of the total variation.

Examining the results from the simulated datasets, we noticed that no `COMPRESSION` feature has been found in any of the assemblies. The first eigen vector of the simulated dataset is similar to the ones obtained from real data, indicating a consistency between the two analyses. Again `LOW_GOOD_CVG` and `NUM_CONTIG` are among the most important features and N50 is again negatively correlated. The second component is similar to the one obtained from real data, as the main source of variation is again between assemblies characterized by repeats assembled in the wrong copy number and assemblies with too few contigs and breakpoints. The first three components account for 70% of the total variance.

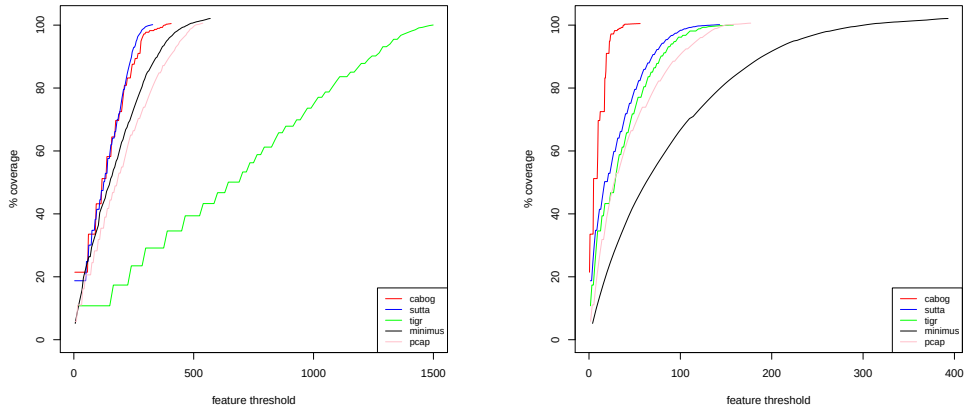
A closer examination reveals that real and simulated PCs are somewhat different. Even though a complete absence of a feature in the simulated dataset (probably a failure of the read simulator to properly simulate the insert variation), we notice several differences: the first "simulated PC" gives non-negligible importance to features like `STRETCH`, `HIGH_SNP`, and `KMER_COV` that have much smaller importance in the first "real PC." A similar situation holds true also for the second PC. The third components are utterly different (see Table 6.5), but not unexpected. We are thus led to conclude that sequence assembly evaluation based on simulated experiments could be misleading, unless genome sequence simulators are further improved.

The principal component analysis (PCA) convinced us that the feature-space is highly over-dimensioned. Therefore we tried to *select* from the feature space the most informative features in order to estimate the performance of different assemblers on a small feature subspace. This analysis, leading to feature selection, was accomplished using another multivariate technique known as Independent Component Analysis (ICA). Following the method proposed in [128], we performed ICA using the *fastICA* algorithm on the forensics features. We extracted the Independent Components (ICs) and selected the most representative feature in each of the ICs with the highest kurtosis value. From the real dataset, we selected the following 6 features: `COMPRESSION`, `HIGH_OUTIE_CVG`, `HIGH_SINGLEMATE_CVG`, `HIGH_READ_COVERAGE`, `KMER_COV`, and `LOW_GOOD_CVG`.

In Figure 6.9, we illustrate how the ICA-subspace allows better evaluation of different assem-

	Real			Simulated		
FEATURES	PC1	PC2	PC3	PC1	PC2	PC3
BREAKPOINT	0.29	-0.14	-0.21	0.26	-0.38	-0.04
COMPRESSION	0.32	0.22	0.35	-	-	-
STRETCH	-0.06	0.08	0.27	0.22	0.42	0.12
HIGH_NORMAL_CVG	-0.10	0.40	0.21	0.02	0.2	-0.44
HIGH_OUTIE_CVG	-0.07	0.56	-0.09	0.12	0.46	0.01
HIGH_READ_COVERAGE	0.36	0.10	-0.13	0.36	0.21	-0.19
HIGH_SINGLEMATE_CVG	-0.01	0.27	-0.53	0.04	-0.07	-0.76
HIGH_SNP	0.05	-0.23	-0.13	0.30	0.02	-0.18
HIGH_SPANNING_CVG	0.28	0.38	0.31	0.41	0.04	0.00
KMER_COV	-0.03	0.37	-0.48	0.24	0.37	0.16
LOW_GOOD_CVG	0.50	-0.04	-0.02	0.41	-0.28	0.04
N50	-0.23	0.09	0.20	-0.27	0.01	-0.30
NUM_CONTG	0.50	-0.03	-0.02	0.39	-0.31	0.02
cumulative variation	27%	44%	55%	36%	59%	70%

Table 6.5: Most Informative Principal Components For Long Reads. First three PCs for the two long reads datasets: real long reads, simulated long reads. At the bottom of each component we reported the cumulative variation represented.



(a) FRC computed on all the feature space

(b) FRC computed on the ICA features

Figure 6.9: Figure 6.9(a) shows the FRC-curve for the 5 assemblers on Brucella suiss dataset when using all the feature space. Figure 6.9(b) shows the FRC-curve computed on the ICA-selected features.

blers. Figure 6.9(a) shows the FRC for the assembly of the *Brucella suis* dataset. We see how, rather surprisingly, TIGR now behaves much worse than all other assemblers, while PCAP, MINIMUS, CABOG and SUTTA have comparable performances. It is surprising that TIGR performs worse than MINIMUS, which does not use the important information, available in paired reads. Inspecting these two assemblies closely (Table 6.6), we see how MINIMUS produces a highly fragmented assembly (206 contigs) in comparison to TIGR (69 contigs). Also the number of detected mis-assemblies via contig alignment suggests us that MINIMUS produces the worst-scoring assembly in the group. If we plot the FRC after reducing the space to only the ICA-features (Figure 6.9(b)) we obtain a slightly different picture. Looking only at the most informative features we discovered that CABOG performs better than all the other assemblers, while SUTTA, TIGR and PCAP are more or less equivalent. This picture is concordant with the results showed in Table

6.6, from which we clearly see that MINIMUS is the assembler with the poorest performance.

Assembler	# Ctg	N50 (Kbp)	Max (Kbp)	Errs	# Feat	# corr Feat	# ICA Feat	# corr ICA Feat
cabog	41	265	711	24	375	24	45	18
minimus	205	31	89	44	382	37	208	36
pcap	91	69	194	50	455	57	94	41
sutta	72	93	621	45	261	23	75	22
tigr	69	111	357	31	1281	24	134	20

Table 6.6: Brucella suis assemblies obtained with Long reads have been compared using standard assembly statistics. We reported the assembler employed, the number of contig returned by the assembler, the N50 length, the length of the longest contig and the number of mis-assemblies identified by *dnadiff*. Moreover we reported the number of features returned by *amosvalidate* and the number of such features that overlap with a real mis-assembly. The same data is reported for the ICA-features.

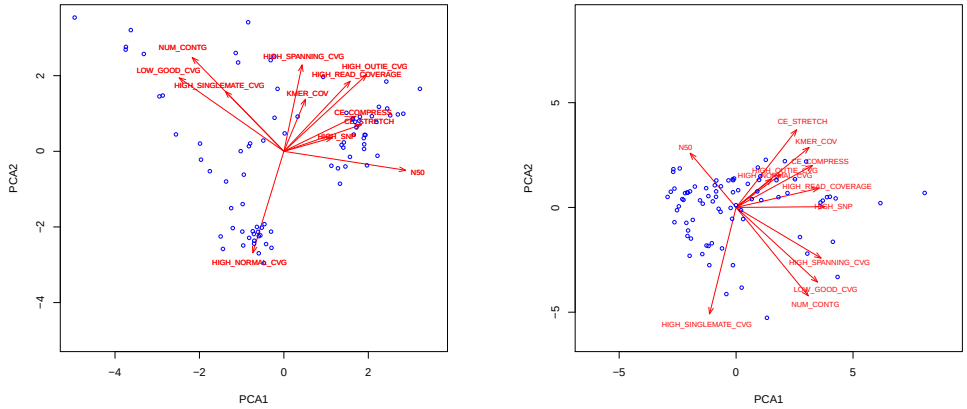
Last four columns of Table 6.6 show how, in general, by reducing the feature space we are able to discard a large number of features (in the TIGR case we pass from 1281 to 134 features) without discarding any significant number of valid features (*i.e.*, features that coincide with real mis-assemblies). This statistics on true discovery suggest that our method does not suffer from a lack of desirable sensitivity. It was noticed in [143] that assembly features have, in general, high sensitivity (higher than 98%) but they lack specificity. We also noticed that the situation remains true even after dimensionality reduction of the feature space. In general this is a consequence of how features arise in two scenarios: features that affect large portions of contigs and assembler-specific features. In the first scenario a feature affects a large portion of a contig when, however, only a relatively small fragment of such contig is a true mis-assembly. The second scenario is much more problematic, we noticed that some assemblers have a particular feature that appears almost in every contig (in the case of *Brucella Suis*, `LOW_GOOD_CVG` appears in almost all TIGR contigs). When this feature is selected by the ICA analysis the specificity is deeply affected (however, the sensitivity remains high). This situation can be avoided by selecting the most representative features for each assembler, but a larger dataset of genomes is necessary in order to successfully apply PCA and ICA.

Short Reads Results

As explained before, the real short read dataset is somewhat different from the simulated ones. In the real dataset, we used only 4 different genomes, sequenced with Illumina producing reads of different lengths. In order to obtain a number of assemblies that allowed PCA and ICA we extracted and assembled with different coverages. We chose to use four different kinds of reads to obtain a set of PCs as general as possible. However it would be preferable to have a larger and more representative datasets to obtain more accurate results. On the other hand, the simulated dataset was obtained by simulating on 20 different organisms at a constant coverage (80×) composed of paired reads of length 100 bp and insert size of 600 bp. The results obtained using this dataset gave us a picture of the state-of-the-art assembly capabilities. However, as seen in the analyses of long reads, PCs obtained through simulated data appear to be far-off from the real ones.

Again, PCA analysis on the real and simulated dataset (Table 6.7) suggested the presence of highly "over-dimensioned" feature space. In order to achieve the 80% of the variance, while we need only 5 components in the real datasets, just 4 are adequate in the simulated ones. Using more sophisticated random matrix theory and the Marčenko-Pastur function, we observed that it is safe to disregard an extra PC with no loss of accuracy in either cases.

As far as the first real PC is concerned, we saw how the `LOW_GOOD_CVG` and N50 are among the most important features. Again, as in the long reads datasets, the two features are negatively correlated. While in the first long read PC most of the features were positively correlated, in the short read case it was no longer true. We saw that compression and extension events (`COMPRESSION`, `STRETCH`) are correlated to mate-pairs problems (`HIGH_OUTIE_CVG`) while the number of contigs is



(a) First and Second Principal Component with real short read Dataset

(b) First and Second Principal Component with simulated short read Dataset

Figure 6.10: First PC versus Second PC: Short Reads Datasets. The plots in Figures 6.10(a) and 6.10(b) show the results of plotting the first principal component against the second. The blue dots represent the assemblies.

	Real			Simulated		
FEATURES	PC1	PC2	PC3	PC1	PC2	PC3
BREAKPOINT	-	-	-	-	-	-
COMPRESSION	-0.28	-0.15	0.24	0.32	0.20	0.33
STRETCH	-0.30	-0.11	0.32	0.2	0.37	0.26
HIGH_NORMAL_CVG	0.12	0.44	-0.09	0.15	0.13	-0.62
HIGH_OUTIE_CVG	-0.32	-0.33	-0.29	0.19	0.15	-0.536
HIGH_READ_COVERAGE	-0.26	-0.30	-0.41	0.35	0.09	-0.01
HIGH_SINGLEMATE_CVG	0.23	-0.26	-0.37	-0.11	-0.50	0.15
HIGH_SNP	-0.19	-0.05	-0.38	0.37	0.00	-0.06
HIGH_SPANNING_CVG	-0.07	-0.38	0.12	0.36	-0.24	-0.16
KMER_COV	-0.08	-0.22	0.47	0.31	0.28	0.28
LOW_GOOD_CVG	0.41	-0.32	0.09	0.34	-0.35	0.09
N50	-0.48	0.08	0.10	-0.19	0.25	0.02
NUM_CONTG	0.36	-0.41	0.12	0.30	-0.42	0.03
cumulative variation	26%	50%	63%	43%	62%	75%

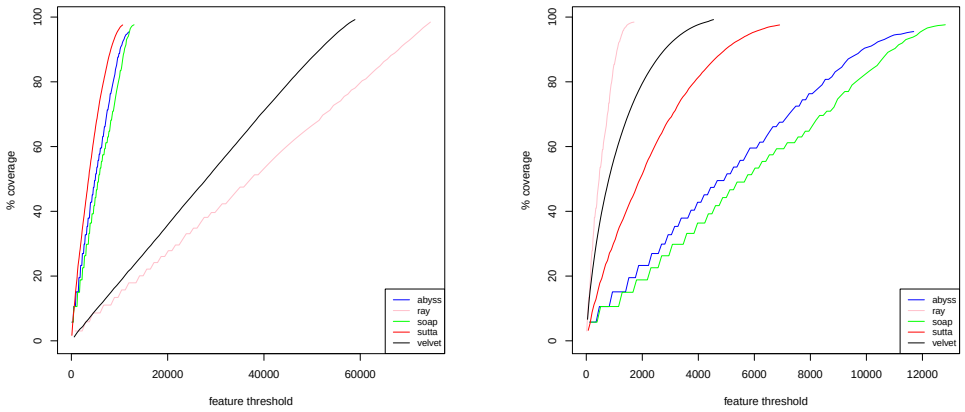
Table 6.7: Most Informative Principal Components For Short Reads. First three PCs for the two short reads datasets: real short reads and, simulated short reads. At the bottom of each component we reported the cumulative variation represented.

positively correlated to areas with low coverage. These effects can be explained in the following way: areas with compression and extension events are likely to contain a large number of mis-oriented reads, while the production of an excess of contigs can be a consequence of a failure in properly estimating the copy number of repeated sequences (thus resulting in a low coverage). The second PC distinguishes assemblies with high `HIGH_NORMAL_CVG`. All the other relevant features are negatively correlated to this one.

As expected, the PCs resulting from the simulated dataset differed to some degree from the ones obtained from real datasets. Also in this case N50 is negatively correlated and its coefficient is not among the maximal ones (like in the long read case). The first component is similar to the first component of the simulated long read dataset. In the second component the main source of

variation between assemblies could be explained by a low number of contigs and regions covered only by unpaired reads as well as a large number of compression expansion events and mate pairs in different contigs.

Using ICA we extracted two feature subsets: one for the real data and the other for the simulated data. As before, we considered ICs that account for 80% of the kurtosis distribution. The ICA-space for the real dataset is formed by 6 features: `COMPRESSION`, `LOW_GOOD_CVG`, `KMER_COV`, `HIGH_SPANNING_CVG`, `HIGH_OUTIE_CVG`, and `CE_STRETCH`.



(a) FRC computed on all the feature space on a real dataset

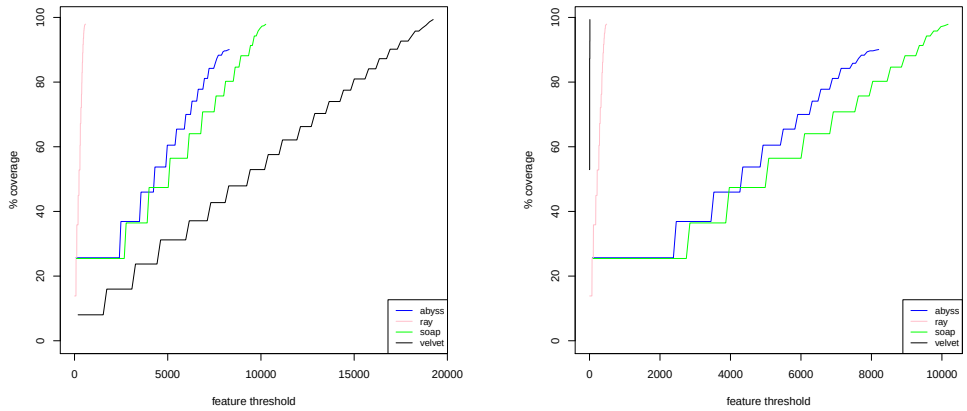
(b) FRC computed on the ICA features space on a real dataset

Figure 6.11: Feature Response Curve and ICA features: Real Short Reads. Figure 6.11(a) show the FRC-curve for the 5 assemblers on E. coli real dataset (read length 36 bp, insert size 200 bp and coverage 130×) when using all the feature space. Figure 6.11(b) shows the FRC-curve computed on the ICA-selected features.

In Figure 6.11(a) we drew the FRC for the *E. coli* dataset composed of paired reads of length 36 bp that form a 130× coverage of the sequenced genome. From this picture we can clearly see how SUTTA, ABySS and SOAPdenovo outperform RAY and VELVET. This situation is in contrast with the analysis presented in Table 6.8, where we clearly see that RAY is the assembler generating very few mis-assemblies along with ABySS, SUTTA and SOAP all behaving similarly. VELVET has much larger number of mis-assemblies, suggesting that the long contigs that it produces are often a consequence of incorrect choices. If we reduce to the ICA-subspace (Fig. 6.11(b)) the picture changes drastically but some problems still remain: Ray, as one would expect, becomes the best assembler, but it is now surprisingly closely followed by VELVET. Moreover, ABySS becomes one of the worst assemblers. This situation is probably a consequence of the way in which features have been computed: ABySS and SOAP provide no facility but to map reads back to the contigs in order to build a layout [2]. This approach clearly skews our empirical analysis. Nonetheless, we can see how the reduced ICA space is able to highlight the good performances of RAY. The last four columns of Table 6.8 show that ICA-features significantly reduce the number of features to be considered (even if this time the reduction is not as impressive as the one obtained with long reads) without noticeably affecting the number of real features (with the only exception of VELVET). This picture motivates us again to highlight the need for assemblers to provide read layouts that could ensure a meaningful evaluation

In the short read case, we explored the ICA-features also for the simulated dataset too. We decided to proceed in these analysis with this dataset in order to avoid the bias produced by the small dataset of real short reads. However, due to the differences between the real and simulated PCs

[2] it is not clear if the other de Bruijn based assemblers build a real (Sanger-like) layout or if they make use of heuristics



(a) FRC computed on all the feature space on a simulated dataset

(b) FRC computed on the ICA features space on a simulated dataset

Figure 6.12: Feature Response Curve and ICA features: Simulated Short Reads. Figure 6.12(a) show the FRC-curve for 4 assemblers on Brucella suis simulated dataset (read length 100 bp, insert size 600 bp and coverage 80×) when using all the feature space. Figure 6.12(b) shows the FRC-curve computed on the ICA-selected features.

Assembler	# Ctg	N50	Max (Kbp)	Errs	# Feat	# corr Feat	# ICA Feat	# corr ICA Feat
abyss	113	97	268	11	11804	119	11475	105
ray	194	58	140	17	74565	52	1701	30
soap	125	109	267	62	12254	174	12053	140
sutta	690	11	41	56	7949	140	5528	114
velvet	65	142	428	136	2156	26	131	2

Table 6.8: Assembly Comparison Real Short Reads: E. coli 130×. E. coli assemblies obtained with short real reads have been compared using standard assembly statistics. We reported the assembler employed, the number of contig returned by the assembler, the N50 length, the length of the longest contig and the number of mis-assemblies identified by *dnadiff*. Moreover we reported the number of features returned by *amosvalidate* and the number of such features that overlap with a real mis-assembly. The same data is reported for the ICA-features.

Assembler	# Ctg	N50	Max (Kbp)	Errs	# Feat	# corr Feat	# ICA Feat	# corr ICA Feat
abyss	20	301	850	2	8250	67	8174	63
ray	27	261	459	1	590	5	486	2
soap	30	299	843	15	10142	112	10057	108
velvet	23	663	1010	22	13547	149	11	1

Table 6.9: Assembly Comparison Simulated Short Reads: Brucella suis. Brucella suis assemblies obtained with short simulated reads have been compared using standard assembly statistics. We reported the assembler employed, the number of contig returned by the assembler, the N50 length, the length of the longest contig and the number of mis-assemblies identified by *dnadiff*. Moreover we reported the number of features returned by *amosvalidate* and the number of such features that overlap with a real mis-assembly. The same data is reported for the ICA-features.

with long reads we consider the following analysis of doubtful value. We again selected 6 features: namely, `HIGH_READ_COVERAGE`, `HIGH_SNP`, `HIGH_NORMAL_CVG`,`HIGH_SPANNING_CVG`, `KMER_COV`, and `STRETCH`. Figure 6.12 demonstrates the differences between FRC curve computed on all the feature space (Fig. 6.12(a)) and on the ICA-space (Fig. 6.12(b)). As before, we observed a similar anoma-

lous behaviour: VELVET is the worst assembly when all the features are taken into account, while it is the best assembler when only the ICA-features are counted (Ray, ABySS and SOAPdenovo do not show any significant variation). These pictures are in contrast with the data summarized in Table 6.9 where, again, we can see that RAY and ABySS are the assemblers less affected by mis-assemblies, while VELVET contains as many as 23 mis-assemblies. A closer scrutiny explained that VELVET has a large number of `HIGH_SINGLEMATE_CVG` (that are clear witnesses of a mis-assembled region) that are not taken into account in the ICA-space. This is a clear bias that affects the ICA analysis but it is difficult to estimate how much this depends on the read simulator or on the in-vitro generated layout.

6.3 Conclusion

Validation and evaluation of data and results are mandatory steps in *de novo* assembly. In this Chapter we presented two methods able to enhance (filtering pipeline) and evaluate (16mer-Counter) the information available in the reads. In particular we illustrated how quality information can be used to improve our datasets and how we can extract from reads themselves useful information like genome size and heterozygosity levels. In the second part we presented a multivariate study of assembly forensics features. The aim of this study was to understand the relationships among different features and how they correlate among them. Moreover we were able to gauge N50 performances in describing assembly quality, reaching the conclusion that such feature badly describes correctness. As a side effect of our analysis we were able also to highlight the lack of available read simulator to effectively reproduce real experiments. This result, in particular, casts a shadow over several assembler evaluation totally based on simulated data (*i.e.* assemblathon first edition).

However, a lot of work need do be done to better understand how features can be improved. Reducing the Feature-space through Independent Component Analysis does not solve the lack of specificity of forensics features. It is not clear if this is a consequence of the currently designed features and if new features can circumvent this problem. We identified as a major stumbling block in obtaining reliable results, the lack of several NGS-based assemblers to produce an assembly-layout. Moreover, it is not obvious how a subset of informative features can be learned, so that a global optimization formulation of the sequence assembly problem (in terms of few score and penalty functions involving these features) would lead to higher fidelity.

Conclusions

The recent Next Generation Sequencing outbreak reshaped our view and perception of Genomics. NGS sequencers are able to produce a large amount of data at a constantly dropping cost. While, on the one hand, this allowed us to sequence and resequence a large amount of new species and of individuals within a population, on the other hand, new and old problems jeopardize the possibility to effectively use all the produced information. New sequencing technologies have reduced costs and increased throughput. However, they have sacrificed read's length and accuracy by allowing more single nucleotide (base-calling) and indel (*e.g.*, due to homopolymer) errors.

In this dissertation we focused our attention on two of the most studied and pressing problems of today's genomics: short read alignment and *de novo* assembly. In particular we showed how, more often than not, standard complexity measures fail in reliably correctly describing tools' performances and problems' complexity. In our opinion, a deep understanding of problems' complexity and, perhaps precedent, an improved characterization of them, is of primary importance in order to find correct solutions and overcame approximated and heuristic approaches.

The first part of this dissertation was focused on the alignment problem. The main contribution presented in this part is the short read aligner *rNA* (randomized Numerical Aligner). rNA is an hash-based aligner currently used by several research groups in the world. rNA's unique feature is its capability to solve the *best-k mismatch* problem without using heuristics affecting its sensitivity. We showed how rNA, thanks to a refined *Hamming-aware* hash function and to a well studied implementation, is able to correctly align more reads than most of the available aligners, yet requiring a feasible amount of time and space. Moreover, a distributed implementation of rNA allows to boost tool's performances by aligning reads over a cluster of tightly connected machines.

A common feature of all software designed for NGS is the necessity to constantly upgrade them in order, on the one hand, to deal with the constantly larger amount of data produced and, on the other hand, to manage new kinds of input data. Even though rNA is a complete and stable software package we are constantly improving it in a number of ways. Currently we are studying new solutions (the integration and the use of q-grams being on the top of the list) to further speed up our algorithm, moreover, we are studying the possibility to adapt rNA to the alignment of bisulfite treated DNA, for a possible use in the determination of the so-called methilation maps.

Even though string alignment is a well known and studied problem, the NGS revolution obliged the Computer Science community to redesign algorithms and to propose new tools to solve this problem. In a (re-)sequencing project, alignment is the basic operation of all downstream analyses like SNP calling, structural variation identification, and gene annotation, to name only the most important. All these analyses, and many others, require tools able to align reads in a fast and correct way. The vast majority of current tools satisfy the first requirement sacrificing the second. rNA, instead, is able to achieve high throughput without affecting its capability to correctly align reads, moreover, in order to meet the always increasing request of high alignment throughput, we are considering the possibility to integrate compression methods in the rNA algorithm (*e.g.* integrating Hamming-aware function and FM-indexes).

The second part of this dissertation was focused on the *de novo* assembly problem. In this part we discussed the theoretical and practical problems related to the reconstruction of genomes with short sequences. In particular we showed the weak points of (current) complexity analyses, of assembling algorithms/heuristics, and of validation methods. The resulting picture is in some way dusky: not only often assemblers are based on a small set of algorithmic ideas and differ among one another only for the collection of implemented heuristics but, moreover, there is no clear and accepted way to validate the results produced by these tools. This is particularly troublesome if we

consider that, encouraged by low costs, an increasing number of projects already started aiming at assembling new organisms using only NGS data.

Despite is commonly accepted that an high coverage is able to overcame problems introduced by short reads and short inserts, we showed that available assemblers are not able to take advantage of coverages higher than 50× and, moreover, we showed how an hold and expensive 8× Sanger coverage allows to achieves better results. This fact raises the question if exchange read length for coverage is worth the deal. Several published assemblies, and several ongoing projects, are trying to solve this problem using hybrid datasets (*i.e.*, datasets composed by reads generated from several different technologies), but there is no general agreement on what technologies use and in which proportions data should be generated. At the same time, several *de novo* assembler are designed to take advantages from this hybrid datasets (*e.g.*, Ray and SUTTA). We think that this approach, coupled with the advent of single molecule sequencing and with improved validation method could help to overcome many present problems.

In the meantime, in order to overcame some of the problems of this field, we proposed two methods (eRGA and GapFiller) aiming at resolving *de novo* assembly under simplified hypothesis. *eRGA* solves the *de novo* assembly problem in presence of a reference genome belonging to an organism closely related to the sequenced one. *GapFiller*, instead, works in a simplified setting trying "only" to close the gap between two paired reads, thus "locally" assembling a small portion of the genome. We showed how both methods allow to partially solve *de novo* assembly problem and, in different ways, to improve results achievable with standard available assemblers. GapFiller, in particular, is still in its embryonic stage and has yet to achieve its full potential. We showed how this tool is able to report as output a set of certified correct contigs, however we still have to show how those certified contigs can be used to improve assembly of complex genome and/or to reconstruct structural variations.

As far as the validation problems is concerned, we concentrated our efforts on the study of the *de novo* validation techniques used so far. We started from the assumption that no widely accepted solution exists and that most of the metrics used in the last ten years emphasized only contig size while poorly capturing an overall "assembly quality". From this point of view we proposed new instruments able, on the one hand, to enhance data being provided as input and, on the other hand, to evaluate *de novo* assembler's results. We believe of primary importance the study performed on the so called *forensics features* and on their correlation. Such a study not only demonstrated once and for all the bad performances in predicting assembly quality of the most (ab)used metric (N50), but it also showed the importance of carefully analyse features in order to design improved assemblers. We believe that an instrument like the *Feature Response Curve* (FRC) coupled with a deep understanding of assembly features could be the key for a more accepted and quality-driven assemblies/assemblers evaluation. Despite its potentialities, FRC needs several enhancements. In particular, FRC is now suited only for small (*i.e.*, bacterial) genomes: we are now designing and implementing a software able to extract a set of features and to plot the resulting FRC on genomes of larger sizes.

Sanger sequencing analysis (based on a technology that remained substantially unchanged in the last 10 years) required more than 10 years to became standard and globally accepted. NGS-revolution has just started: until now the community has been more interested on technology improvements (read's length, error per base, *etc.*) than on standardizing procedures and analyses. However, having in mind the final goal of Personal Genomics, widely accepted procedures and tools will soon become a priority. All the contributions presented in this dissertation aim at achieving such widely accepted standards (*e.g.*, rNA, GapFiller), providing, at least, the elements to start profitable discussions on how to improve current techniques and how to choose focus points (*e.g.*, forensics features).

Bibliography

[1] `http://www.clcbio.com/files/usermanuals/CLC_Genomics_Workbench_User_Manual.pdf`. pp. 405–406.

[2] The VIGNA/VIGNE Consortium Project Site.

[3] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *J. of Discrete Algorithms*, 2(1):53–86, 2004.

[4] K. Abrahamson. Generalized string matching. *SIAM Journal on Computing*, 16(6):1039–1051, 1987.

[5] Roger P. Alexander, Gang Fang, Joel Rozowsky, Michael Snyder, and Mark B. Gerstein. Annotating non-coding regions of the genome. *Nature Reviews Genetics*, 11(8):559–571, July 2010.

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

[7] S. F. Altschul, T. L. Madden, A. A. Schäffer, and *et al.* Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic acids research*, 25(17):3389–3402, 1997.

[8] A. Amir, M. Lewenstein, and E. Porat. Faster algorithms for string matching with k mismatches. *Journal of Algorithms*, 50:257–275, 2004.

[9] Apache Hadopp. http://hadoop.apache.org/.

[10] A. Apostolico. The myriad virtues of sub-word trees. *Combinatorics on Words*, 112:85–96, 1985.

[11] S. Batzoglou, D. B. Jaffe, K. Stanley, and *et al.* Arachne: A whole-genome shotgun assembler. *Genome Res.*, 12(1):177–189, January 2002.

[12] D. R. Bentley, S. Balasubramanian, H. P. Swerdlow, and *et al.* Accurate whole human genome sequencing using reversible terminator chemistry. *Nature*, 456(7218):53–9, November 2008.

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

[14] Jinbo Bi, K. Bennett, Mark Embrechts, C. Breneman, and Minghu Song. Dimensionality reduction via sparse support vector machines. *The Journal of Machine Learning Research*, 3(7-8):1229–1243, October 2003.

[15] A. Blumer, J. Blumer, D. Haussler, and *et al.* The smallest automation recognizing the subwords of a text. *Theoretical Computer Science*, 40:31–55, 1985.

[16] A. Blumer, J. Blumer, D. Haussler, and *et al.* Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM*, 34(3):578–595, July 1987.

[17] S. Boisvert, F. Laviolette, and J. Corbeil. Ray: simultaneous assembly of reads from a mix of high-throughput sequencing technologies. *J Comput Biol*, 17(11):1519–1533, Nov 2010.

[18] C. Boutsidis, M.W. Mahoney, and P. Drineas. Unsupervised feature selection for principal components analysis. In *Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 61–69. ACM, 2008.

[19] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.

[20] D. W. Bryant, W. K. Wong, and T. C. Mockler. Qsra: a quality-value guided de novo short read assembler. *BMC Bioinformatics*, 10:69, 2009.

[21] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. (124), 1994.

[22] J. Butler, I. MacCallum, M. Kleber, and *et al.* Allpaths: de novo assembly of whole-genome shotgun microreads. *Genome Res*, 18(5):810–820, May 2008.

[23] 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. IEEE, 1993.

[24] Alberto Casagrande, Cristian Del Fabbro, Simone Scalabrin, and Alberto Policriti. Gam: Genomic assemblies merger: A graph based method to integrate different assemblies. *Bioinformatics and Biomedicine, IEEE International Conference on*, 0:321–326, 2009.

[25] Federica Cattonaro, Alberto Policriti, and Francesco Vezzi. Enhanced reference guided assembly. In *2010 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 77–80. IEEE, December 2010.

[26] C L C Assembly Cell. Clc - white paper. *White Paper*, 2010.

[27] Eugene Y Chan. Advances in sequencing technology. *Mutation research*, 573(1-2):13–40, June 2005.

[28] W. I. Chang and T. G. Marr. Approximate string matching and local similarity. In *Proc. 5th Ann. Symp. on Combinatorial Pattern Matching*, pages 259–273, 1994.

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

[30] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, McGraw-Hill Book Company, 2nd edition, 2001.

[31] Richard Cronn, Aaron Liston, Matthew Parks, and *et al.* Multiplex sequencing of plant chloroplast genomes using solexa sequencing-by-synthesis technology. *Nucleic acids research*, 36(19):e122, November 2008.

[32] M. David, M. Dzamba, D. Lister, and *et al.* SHRiMP2: Sensitive yet practical short read mapping. *Bioinformatics (Oxford, England)*, 27(7):1011–2, April 2011.

[33] N. G. de Bruijn. A Combinatorial Problem. *Koninklijke Nederlandse Akademie v. Wetenschappen*, 49:758–764, 1946.

[34] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[35] J. C. Dohm, C. Lottaz, T. Borodina, and H. Himmelbauer. Sharcgs, a fast and highly accurate short-read assembly algorithm for de novo genomic sequencing. *Genome Res*, 17(11):1697–1706, Nov 2007.

[36] Bas E Dutilh, Martijn a Huynen, and Marc Strous. Increasing the coverage of a metapopulation consensus genome by iterative read mapping and assembly. *Bioinformatics (Oxford, England)*, 25(21):2878–81, November 2009.

[37] 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.

[38] John Eid, Adrian Fehr, Jeremy Gray, Khai Luong, John Lyle, Geoff Otto, Paul Peluso, David Rank, Primo Baybayan, Brad Bettman, and Others. Real-Time DNA Sequencing from Single Polymerase Molecules. *Science*, 323(5910):133, 2009.

[39] B. Ewing and P. Green. Base-calling of automated sequencer traces using phred. II. Error probabilities. *Genome research*, 8(3):186–194, March 1998.

[40] P. Ferragina and G. Manzini. Opportunistic data structures with applications. page 390, 2000.

[41] Paolo Ferragina. String algorithms and data structures. *CoRR abs/0801.2378*, 2008.

[42] Benjamin A. Flusberg, Dale R. Webster, Jessica H. Lee, Kevin J. Travers, Eric C. Olivares, Tyson A. Clark, Jonas Korlach, and Stephen W. Turner. Direct detection of DNA methylation during single-molecule, real-time sequencing. *Nature methods*, 7(6):461–465, June 2010.

[43] Z. Galil and R. Giancarlo. Improved string matching with k mismatches. *SIGACT News*, 17(4):52–54, 1986.

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

[45] S. Gnerre, I. MacCallum, D. Przybylski, and *et al.* High-quality draft assemblies of mammalian genomes from massively parallel sequence data. *Proceedings of the National Academy of Sciences*, pages 1–6, December 2010.

[46] Sante Gnerre, Eric S Lander, Kerstin Lindblad-Toh, and David B Jaffe. Assisted assembly: how to improve a de novo genome assembly by using related species. *Genome biology*, 10(8):R88, January 2009.

[47] S. M. D. Goldberg, J. Johnson, D. Busam, and *et al.* A sanger/pyrosequencing hybrid approach for the generation of high-quality draft assemblies of marine microbial genomes. *Proceedings of the National Academy of Sciences of the United States of America*, 103(30):11240–5, July 2006.

[48] O. Gotoh. An improved algorithm for matching biological sequences. *Journal of molecular biology*, 162(3):705–8, December 1982.

[49] A.J.F. Griffiths, J.H. Miller, D. T. Suzuki, R. C. Lewontin, and W. M. Gelbart. *An Introduction to Genetic Analysis*. W.H. Freeman and Company, 1993.

[50] W. Gropp, E. Lusk, and A. Skjellum. Using MPI: portable parallel programming with the message passing interface. 1999.

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

[52] D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology.* Cambridge University Press, 1997. Chapter 3.

[53] R. W. HAMMING. Error detecting and error correcting codes. *BELL SYSTEM TECHNICAL JOURNAL*, 29(2):147–160, 1950.

[54] D. Hernandez, P. François, L. Farinelli, and *et al.* De novo bacterial genome sequencing: millions of very short reads assembled on a desktop computer. *Genome*, 18:802–809, 2008.

[55] Nils Homer, Barry Merriman, and Stanley F Nelson. Bfast: an alignment tool for large scale genome resequencing. *PloS one*, 4(11):e7767, January 2009.

[56] W. K. Hon, T. W. Lam, K. Sadakane, and *et al.* A space and time efficient algorithm for constructing compressed suffix arrays. *Algorithmica*, 48(1):23–36, March 2007.

[57] D. S. Horner, G. Pavesi, T. Castrignano, and *et al.* Bioinformatics approaches for genomics and post genomics applications of next-generation sequencing. *Brief Bioinform*, pages bbp046+, 2009.

[58] M. S. Hossain, N. Azimi, and S. Skiena. Crystallizing short-read assemblies around seeds. *BMC Bioinformatics*, 10 Suppl 1:S16, 2009.

[59] S Howorka, S Cheley, and H Bayley. Sequence-specific detection of individual DNA strands using engineered nanopores. *Nature biotechnology*, 19(7):636–9, July 2001.

[60] Sanwen Huang, Ruiqiang Li, Zhonghua Zhang, and *et al.* The genome of the cucumber, cucumis sativus l. *Nature genetics*, 41(12):1275–81, December 2009.

[61] X. Huang, J. Wang, S. Aluru, S. P. Yang, and L. Hillier. Pcap: A whole-genome assembly program. *Genome Res.*, 13(9):2164–2170, 2003.

[62] Xiaoqiu Huang and Shiaw-Pyng Yang. Generating a genome assembly with pcap. *Curr Protoc Bioinformatics*, Chapter 11:Unit11.3, Oct 2005.

[63] T. N. D. Huynh, W.-K. Hon, T.-W. Lam, and W.-K. Sung. Approximate string matching using compressed suffix arrays. *Theor. Comput. Sci.*, 352(1):240–249, 2006.

[64] A Hyvärinen, J Karhunen, and O Erkki. *Independent Component Analysis.* John Wiley & Sons, first edition, 2001.

[65] A. Hyvärinen and Erkki Oja. A fast fixed-point algorithm for independent component analysis. *Neural computation*, 9(7):1483–1492, 1997.

[66] R. M. Idury and M. S. Waterman. A new algorithm for dna sequence assembly. *J Comput Biol*, 2(2):291–306, 1995.

[67] I.F. Imam and Haleh Vafaie. An empirical comparison between global and greedy-like search for feature selection. In *Proceedings of the Florida AI Research Symposium (FLAIRS-94), Pensacola Beach, FL*, pages 66–70. Citeseer, 1994.

[68] S. Inenaga, H. Hoshino, A. Shinohara, and *et al.* On-line construction of compact directed acyclic word graphs. In *Discrete Applied Mathematics*, volume 146, pages 156–179. Springer, March 2005.

[69] O. Jaillon, J. M. Aury, B. Noel, A. Policriti, and *et al.* The grapevine genome sequence suggests ancestral hexaploidization in major angiosperm phyla. *Nature*, 449(7161):463–7, September 2007.

[70] W. R. Jeck, J. A. Reinhardt, D. A..Baltrus, and *et al.* Extending assembly of short dna sequences to handle error. *Bioinformatics*, 23(21):2942–2944, Nov 2007.

[71] I.M. Johnstone. High dimensional statistical inference and random matrices. *Arxiv preprint math/0611589*, 2006.

[72] P. Jokinen and E. Ukkonen. Two algorithms for approximate string matching in static texts. In *Proc. 2nd Ann. Symp. on Mathematical Foundations of Computer Science*, volume 520, pages 240–248, 1991.

[73] I. Jolliffe. *Principal Component Analysis, Second Edition*. Wiley Online Library, 2002.

[74] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Automata, Languages and Programming, 30th International Colloquium, ICALP 2003*, volume 2719 of *Lecture Notes in Computer Science*, pages 943–955, 2003.

[75] R. Karp and M. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Develop.*, 31(2):249–260, 1987.

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

[77] David R Kelley, Michael C Schatz, and Steven L Salzberg. Quake: quality-aware detection and correction of sequencing errors. *Genome biology*, 11(11):R116, November 2010.

[78] Carl Kingsford, Michael C Schatz, and Mihai Pop. Assembly complexity of prokaryotic genomes using short reads. *BMC Bioinformatics*, 11:21, 2010.

[79] L.B. Kish. End of moore's law: thermal (noise) death of integration in micro and nano electronics. *Physics Letters A*, 305(3-4):144–149, 2002.

[80] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[81] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3(2-4):143–156, June 2005.

[82] Stefan Kurtz, Apurva Narechania, Joshua C Stein, and Doreen Ware. A new method to compute k-mer frequencies and its application to annotate large repetitive plant genomes. *BMC genomics*, 9:517, January 2008.

[83] G. M. Landau and U. Vishkin. Efficient string matching in the presence of errors. In *Proceedings of the 26th IEEE Symposium on Foundations of Computer Science*, pages 126–136, 1985.

[84] G. M. Landau and U. Vishkin. Efficient string matching with k mismatches. *Theoretical Computer Science*, 43:239–249, 1986.

[85] E S Lander, L M Linton, B Birren, C Nusbaum, M C Zody, J Baldwin, K Devon, K Dewar, M Doyle, W FitzHugh, R Funke, D Gage, K Harris, a Heaford, J Howland, L Kann, J Lehoczky, R LeVine, P McEwan, K McKernan, J Meldrim, J P Mesirov, C Miranda, W Morris, J Naylor, C Raymond, M Rosetti, R Santos, a Sheridan, C Sougnez, N Stange-Thomann, N Stojanovic, a Subramanian, D Wyman, J Rogers, J Sulston, R Ainscough, S Beck, D Bentley, J Burton, C Clee, N Carter, a Coulson, R Deadman, P Deloukas, a Dunham, I Dunham, R Durbin, L French, D Grafham, S Gregory, T Hubbard, S Humphray, a Hunt, M Jones, C Lloyd, a McMurray, L Matthews, S Mercer, S Milne, J C Mullikin, a Mungall, R Plumb, M Ross, R Shownkeen, S Sims, R H Waterston, R K Wilson, L W Hillier, J D McPherson, M a Marra, E R Mardis, L a Fulton, a T Chinwalla, K H Pepin, W R Gish, S L Chissoe, M C Wendl, K D Delehaunty, T L Miner, a Delehaunty, J B Kramer,

L L Cook, R S Fulton, D L Johnson, P J Minx, S W Clifton, T Hawkins, E Branscomb, P Predki, P Richardson, S Wenning, T Slezak, N Doggett, J F Cheng, a Olsen, S Lucas, C Elkin, E Uberbacher, M Frazier, R a Gibbs, D M Muzny, S E Scherer, J B Bouck, E J Sodergren, K C Worley, C M Rives, J H Gorrell, M L Metzker, S L Naylor, R S Kucherlapati, D L Nelson, G M Weinstock, Y Sakaki, a Fujiyama, M Hattori, T Yada, a Toyoda, T Itoh, C Kawagoe, H Watanabe, Y Totoki, T Taylor, J Weissenbach, R Heilig, W Saurin, F Artiguenave, P Brottier, T Bruls, E Pelletier, C Robert, P Wincker, D R Smith, L Doucette-Stamm, M Rubenfield, K Weinstock, H M Lee, J Dubois, a Rosenthal, M Platzer, G Nyakatura, S Taudien, a Rump, H Yang, J Yu, J Wang, G Huang, J Gu, L Hood, L Rowen, a Madan, S Qin, R W Davis, N a Federspiel, a P Abola, M J Proctor, R M Myers, J Schmutz, M Dickson, J Grimwood, D R Cox, M V Olson, R Kaul, N Shimizu, K Kawasaki, S Minoshima, G a Evans, M Athanasiou, R Schultz, B a Roe, F Chen, H Pan, J Ramser, H Lehrach, R Reinhardt, W R McCombie, M de la Bastide, N Dedhia, H Blöcker, K Hornischer, G Nordsiek, R Agarwala, L Aravind, J a Bailey, a Bateman, S Batzoglou, E Birney, P Bork, D G Brown, C B Burge, L Cerutti, H C Chen, D Church, M Clamp, R R Copley, T Doerks, S R Eddy, E E Eichler, T S Furey, J Galagan, J G Gilbert, C Harmon, Y Hayashizaki, D Haussler, H Hermjakob, K Hokamp, W Jang, L S Johnson, T a Jones, S Kasif, a Kaspryzk, S Kennedy, W J Kent, P Kitts, E V Koonin, I Korf, D Kulp, D Lancet, T M Lowe, a McLysaght, T Mikkelsen, J V Moran, N Mulder, V J Pollara, C P Ponting, G Schuler, J Schultz, G Slater, a F Smit, E Stupka, J Szustakowski, D Thierry-Mieg, J Thierry-Mieg, L Wagner, J Wallis, R Wheeler, a Williams, Y I Wolf, K H Wolfe, S P Yang, R F Yeh, F Collins, M S Guyer, J Peterson, a Felsenfeld, K a Wetterstrand, a Patrinos, M J Morgan, P de Jong, J J Catanese, K Osoegawa, H Shizuya, S Choi, Y J Chen, and J Szustakowki. Initial sequencing and analysis of the human genome. *Nature*, 409(6822):860–921, February 2001.

[86] E S Lander and M S Waterman. Genomic mapping by fingerprinting random clones: a mathematical analysis. *Genomics*, 2(3):231–9, April 1988.

[87] B. Langmead, K. D. Hansen, and J. T. Leek. Cloud-scale RNA-sequencing differential expression analysis with Myrna. *Genome biology*, 11(8):R83, January 2010.

[88] B. Langmead, M. C. Schatz, J. Lin, M. Pop, and S. L. Salzberg. Searching for SNPs with cloud computing. *Genome biology*, 10(11):R134, January 2009.

[89] B. Langmead, C. Trapnell, M. Pop, and S. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(3):R25, 2009.

[90] S. Lee, F. Hormozdiari, C. Alkan, and M. Brudno. Modil: detecting small indels from clone-end sequencing with mixtures of distributions. *Nat Methods*, 6(7):473–474, Jul 2009.

[91] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

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

[93] H. Li and R. Durbin. Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinformatics*, 26(5):589, 2010.

[94] H. Li, B. Handsaker, A. Wysoker, and *et al.* The sequence alignment/map format and samtools. *Bioinformatics*, 25(16):2078–2079, Aug 2009.

[95] H. Li, J. Ruan, and R. Durbin. Mapping short dna sequencing reads and calling variants using mapping quality scores. *Genome research*, 18(11):1851, 2008.

[96] R. Li, W. Fan, H. Zhu, and *et al.* The sequence and de novo assembly of the giant panda genome. *Nature*, 463(7279):311–317, December 2009.

[97] R. Li, Y. Li, X. Fang, H. Yang, J. Wang, and K. Kristiansen. Snp detection for massively parallel whole-genome resequencing. *Genome Research*, 19(6):1124–1132, May 2009.

[98] R. Li, Y. Li, K. Kristiansen, and J. Wang. Soap: short oligonucleotide alignment program. *Bioinformatics (Oxford, England)*, 24(5):713–4, March 2008.

[99] R. Li, C. Yu, Y. Li, T. W. Lam, and *et al.* Soap2: an improved ultrafast tool for short read alignment. *Bioinformatics (Oxford, England)*, 25(15):1966–7, August 2009.

[100] R. Li, H. Zhu, J. Ruan, and *et al.* De novo assembly of human genomes with massively parallel short read sequencing. *Genome research*, 20(2):265–72, February 2010.

[101] 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 computational biology*, 1(4):e43, September 2005.

[102] H. Lin, Z. Zhang, M. Q. Zhang, B. Ma, and M. Li. ZOOM! Zillions of oligos mapped. *Bioinformatics (Oxford, England)*, 24(21):2431–7, November 2008.

[103] Yong Lin, Jian Li, Hui Shen, Lei Zhang, C.J. Papasian, and H.W. Deng. Comparative Studies of de novo Assembly Tools for Next-generation Sequencing Technologies. *Bioinformatics*, pages 1–7, 2011.

[104] J. Liu, G. Pearlson, A. Windemuth, G. Ruano, N.I. Perrone-Bizzozero, and V. Calhoun. Combining fMRI and SNP data to investigate connections between brain function and genetics using parallel ICA. *Human brain mapping*, 30(1):241–255, 2009.

[105] Z. Liu, X. Chen, J. Borneman, and T. Jiang. A fast algorithm for approximate string matching on gene sequences. In *Proc. 16th Ann. Symp. on Combinatorial Pattern Matching*, volume 3537 of *Lecture Notes in Computer Science*, pages 79–90, 2005.

[106] Haiping Lu, K.N. Plataniotis, and A.N. Venetsanopoulos. A survey of multilinear subspace learning for tensor data. *Pattern Recognition*, 44(65):1540–1551, 2011.

[107] B. Ma, J. Tromp, and M. Li. Patternhunter: faster and more sensitive homology search. *Bioinformatics (Oxford, England)*, 18(3):440–5, March 2002.

[108] I. MacCallum, D. Przybylski, S. Gnerre, and J. Burton. Allpaths 2: small genomes assembled accurately and with high continuity from short paired reads. *Genome*, 2009.

[109] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. In *SODA '90: Proc. 1st Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 319–327, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[110] Guillaume Marçais and Carl Kingsford. A fast, lock-free approach for efficient parallel counting ofoccurrences of k-mers. *Bioinformatics (Oxford, England)*, pages 764–770, January 2011.

[111] 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 Heen Ho, 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):376–80, September 2005.

[112] A. M. Maxam and W. Gilbert. A new method for sequencing DNA. *Proceedings of the National Academy of Sciences of the United States of America*, 74(2):560–564, February 1977.

[113] E. McGreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23:262–272, 1976.

[114] P. Medvedev, M. Stanciu, and M. Brudno. Computational methods for discovering structural variation with next-generation sequencing. *Nat Methods*, 6(11 Suppl):S13–S20, Nov 2009.

[115] Paul Medvedev, Eric Scott, Boyko Kakaradov, and Pavel Pevzner. Error correction of high-throughput sequencing datasets with non-uniform coverage. *Bioinformatics*, 27:137–141, 2011.

[116] Alexander Mellmann, Dag Harmsen, Craig a. Cummings, Emily B. Zentz, Shana R. Leopold, Alain Rico, Karola Prior, Rafael Szczepanowski, Yongmei Ji, Wenlan Zhang, Stephen F. McLaughlin, John K. Henkhaus, Benjamin Leopold, Martina Bielaszewska, Rita Prager, Pius M. Brzoska, Richard L. Moore, Simone Guenther, Jonathan M. Rothberg, and Helge Karch. Prospective Genomic Characterization of the German Enterohemorrhagic Escherichia coli O104:H4 Outbreak by Rapid Next Generation Sequencing Technology. *PLoS ONE*, 6(7):e22751, July 2011.

[117] Fabian Menges, Giuseppe Narzisi, and Bud Mishra. TOTAL RECALLER : Improved Accuracy and Performance via Integrated Alignment & Base-Calling. *Bioinformatics (Oxford, England)*, pages 1–8, 2011.

[118] J. R. Miller, A. L. Delcher, S. Koren, and *et al.* Aggressive assembly of pyrosequencing reads with mates. *Bioinformatics*, 24(24):2818–2824, Dec 2008.

[119] J. R. Miller, S. Koren, and G. Sutton. Assembly algorithms for next-generation sequencing data. *Genomics*, 95(6):315–327, Jun 2010.

[120] Jason R Miller, Arthur L Delcher, Sergey Koren, Eli Venter, Brian P Walenz, and et Al. Aggressive assembly of pyrosequencing reads with mates. *Bioinformatics (Oxford, England)*, 24(24):2818–24, December 2008.

[121] I. Munro, V. Raman, and S.S. Rao. Space efficient suffix trees. In *Foundations of Software Technology and Theoretical Computer Science*, pages 1053–1053. Springer, 2004.

[122] D. J. Munroe and T. J. R. Harris. Third-generation sequencing fireworks at marco island. *Nature biotechnology*, 28(5):426–8, May 2010.

[123] R. Muth and U. Manber. Approximate multiple string search. In *Proc. 7th Ann. Symp. on Combinatorial Pattern Matching*, pages 75–86, Laguna Beach, CA, 1996.

[124] E. W. Myers. The fragment assembly string graph. *Bioinformatics (Oxford, England)*, 21 Suppl 2:ii79–85, September 2005.

[125] E. W. Myers, G. G. Sutton, A. L. Delcher, and *et al.* A whole-genome assembly of drosophila. *Science*, 287(5461):2196–2204, Mar 2000.

[126] F. Nadalin, F. Vezzi, and A. Policriti. GapFiller: a Preprocessing Step for the De Novo Assembly Problem. In *NETTAB2011: Clinical Bioinformatics*, 2011.

[127] N. Nagarajan and M. Pop. Parametric complexity of sequence assembly: theory and applications to next generation sequencing. *J Comput Biol*, 16(7):897–908, Jul 2009.

[128] Layan Imad Nahlawi and Parvin Mousavi. Single nucleotide polymorphism selection using independent component analysis. *Annual International Conference of the IEEE Engineering in Medicine and Biology Society. IEEE Engineering in Medicine and Biology Society. Conference*, 2010:6186–9, January 2010.

[129] Giuseppe Narzisi and Bud Mishra. Scoring-and-unfolding trimmed tree assembler: Concepts, constructs and comparisons. *Bioinformatics*, Nov 2010.

[130] Giuseppe Narzisi and Bud Mishra. Comparing De Novo Genome Assembly: The Long and Short of It. *PLoS ONE*, 6(4):e19175, April 2011.

[131] Jurgen Nijkamp, Wynand Winterbach, Marcel van den Broek, and *et al.* Integrating genome assemblies with maia. *Bioinformatics (Oxford, England)*, 26(18):i433–i439, September 2010.

[132] Z. Ning, A. J. Cox, and J. C. Mullikin. Ssaha: a fast search method for large dna databases. *Genome Res*, 11(10):1725–1729, Oct 2001.

[133] Pavel A. P. and Mark J. C. Short read fragment assembly of bacterial genomes. *Genome Res*, 18(2):324–330, Feb 2008.

[134] K. Paszkiewicz and D. J. Studholme. De novo assembly of short sequence reads. *Briefings in bioinformatics*, 44(November 2009), August 2010.

[135] pBWA. pbwa - parallel burrows-wheeler aligner. unpublished.

[136] Graziano Pesole. What is a gene? An updated operational definition. *Gene*, 417(1-2):1–4, July 2008.

[137] P. A. Pevzner. 1-tuple dna sequencing: computer analysis. *J Biomol Struct Dyn*, 7(1):63–73, Aug 1989.

[138] P. A. Pevzner, M. J. Chaisson, and D. Brinza. De novo fragment assembly with short mate-paired reads: Does the read length matter? *Genome Res*, 19(2):336–346, Feb 2009.

[139] P. A. Pevzner, P. A. Pevzner, H. Tang, and G. Tesler. De novo repeat classification and fragment assembly. *Genome research*, 14(9):1786–1796, September 2004.

[140] P. A. Pevzner and H. Tang. Fragment assembly with double-barreled data. *Bioinformatics*, 17(suppl_1):S225–233, June 2001.

[141] P. A. Pevzner, H. Tang, and M. Chaisson. Fragment assembly with short reads. *Bioinformatics*, 20:2069–2074, 2004.

[142] P. A. Pevzner, H. Tang, and M. S. Waterman. An eulerian path approach to dna fragment assembly. *Proc Natl Acad Sci U S A*, 98(17):9748–9753, Aug 2001.

[143] Adam M Phillippy, Michael C Schatz, and Mihai Pop. Genome assembly forensics: finding the elusive mis-assembly. *Genome biology*, 9(3):R55, January 2008.

[144] A. Policriti, A. Tomescu, and F. Vezzi. A randomized numerical aligner (rna). *Language and Automata Theory and Applications*, pages 512–523, 2010.

[145] Mihai Pop, Adam Phillippy, Arthur L. Delcher, and Steven L. Salzberg. Comparative genome assembly. *Briefings in bioinformatics*, 5(3):237–248, September 2004.

[146] Mithun Prasad, Arcot Sowmya, and Inge Koch. Efficient feature selection based on independent component analysis. In *Intelligent Sensors, Sensor Networks and Information Processing Conference, 2004. Proceedings of the 2004*, pages 427–432. IEEE, 2004.

[147] K. R. Rasmussen, J. Stoye, and E. W. Myers. Efficient q-gram filters for finding all -matches over a given length. *Journal of Computational Biology*, 13:296–308, 2005.

[148] D.J. Rees, L.H. Husselmann, and J. M. Celton. De novo genome sequencing of the apple scab (venturia inaequalis) genome, using illumina sequencing technology. Plant & Animal Genomes XVII Conference Abstract.

[149] J. A. Reinhardt, D. A. Baltrus, M. T. Nishimura, and *et al.* De novo assembly using low-coverage short read sequence data from the rice pathogen pseudomonas syringae pv. oryzae. *Genome research*, 19(2):294–305, February 2009.

[150] JA Reinhardt, DA Baltrus, MT Nishimura, and WR. Efficient de novo assembly of bacterial genomes using low coverage short read sequencing. *Genome*, 2008.

[151] Daniel C Richter, Felix Ott, Alexander F Auch, Ramona Schmid, and Daniel H Huson. MetaSim: a sequencing simulator for genomics and metagenomics. *PloS one*, 3(10):e3373, January 2008.

[152] Daniel C Richter, Stephan C Schuster, and Daniel H Huson. OSLay: optimal syntenic layout of unfinished assemblies. *Bioinformatics (Oxford, England)*, 23(13):1573–9, July 2007.

[153] Anna I Rissman, Bob Mau, Bryan S Biehl, Aaron E Darling, Jeremy D Glasner, and Nicole T Perna. Reordering contigs of draft genomes using the Mauve aligner. *Bioinformatics (Oxford, England)*, 25(16):2071–3, August 2009.

[154] S. Rodrigue, A. C. Materna, S. C. Timberlake, and *et al.* Unlocking short read sequencing for metagenomics. *PLoS ONE*, 5(7):e11840, July 2010.

[155] 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.

[156] S. M. Rumble, P. Lacroute, A. V. Dalca, and *et al.* SHRiMP: accurate mapping of short color-space reads. 2009.

[157] L. Salmela, J. Tarhio, and P. Kalsi. Approximate boyer-moore string matching for small alphabets. *Algorithmica*, 2009.

[158] 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.

[159] F. Sanger, A. R. Coulson, B. G. Barrell, A. J. Smith, and B. A. Roe. Cloning in single-stranded bacteriophage as an aid to rapid dna sequencing. *J Mol Biol*, 143(2):161–178, Oct 1980.

[160] Eric E Schadt, Steve Turner, and Andrew Kasarskis. A Window into Third Generation Sequencing. *Human molecular genetics*, 19(2):227–240, September 2010.

[161] M. C. Schatz. Cloudburst: highly sensitive read mapping with mapreduce. *Bioinformatics (Oxford, England)*, 25(11):1363–9, June 2009.

[162] M. C. Schatz, A. L. Delcher, and S. L. Salzberg. Assembly of large genomes using second-generation sequencing. *Genome Res*, 20(9):1165–1173, Sep 2010.

[163] D. C. Schwartz, X. Li, L. I. Hernandez, S. P. Ramnarain, E. J. Huff, and Y. K. Wang. Ordered restriction maps of Saccharomyces cerevisiae chromosomes constructed by optical mapping. *Science (New York, N.Y.)*, 262(5130):110–114, October 1993.

[164] Colin A. M. Semple. *Assembling a View of the Human Genome*, pages 93–117. John Wiley & Sons, Ltd, 2003.

[165] J. T. Simpson and R. Durbin. Efficient construction of an assembly string graph using the fm-index. *Bioinformatics*, 26(12):i367–i373, Jun 2010.

[166] J. T Simpson, K. Wong, S. D. Jackman, and *et al.* Abyss: A parallel assembler for short read sequence data. *Genome Res*, 19(6):1117–1123, Jun 2009.

[167] A. D. Smith, Z. Xuan, and M. Q. Zhang. Using quality scores and longer reads improves accuracy of solexa read mapping. *BMC bioinformatics*, 9:128, January 2008.

[168] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–7, March 1981.

[169] Daniel D Sommer, Arthur L Delcher, Steven L Salzberg, and Mihai Pop. Minimus: a fast, lightweight genome assembler. *BMC bioinformatics*, 8:64, January 2007.

[170] G.G. Sutton, Owen White, M.D. Adams, and A.R. Kerlavage. TIGR Assembler: A new tool for assembling large shotgun sequencing projects. *Genome Science and Technology*, 1(1):9–19, 1995.

[171] Dan S. Tawfik and Andrew D. Griffiths. Man-made cell-like compartments for molecular evolution . *Nature Biotechnology*, 16(7):652–656, July 1998.

[172] G A Tuskan, S Difazio, S Jansson, and *et al.* The genome of black cottonwood, populus trichocarpa (torr. & gray). *Science (New York, N.Y.)*, 313(5793):1596–604, September 2006.

[173] E. Ukkonen. Approximate string matching over suffix trees. In *Proc. 4th Ann. Symp. on Combinatorial Pattern Matching*, pages 228–242, 1993.

[174] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[175] J. C. Venter, M. D. Adams, E. W. Myers, and *et al.* The sequence of the human genome. *Science*, 291(5507):1304–1351, Feb 2001.

[176] F. Vezzi, F. Cattonaro, and Po. e-rga: enhanced reference guided assembly of complex genomes. *EMBnet Journal*, 17:46–54, 2011.

[177] F. Vezzi, C. Del Fabbro, A. Tomescu, and A. Policriti. rna: a fast and accurate short reads numerical aligner. Accepted for publication by Bioinformatics Oxford Journal.

[178] Francesco Vezzi, Cristian Del Fabbro, Alexandru I Tomescu, and Alberto Policriti. rNA : a Fast and Accurate Short Reads Numerical Aligner. *Bioinformatics (Oxford, England)*, 2011.

[179] P. Kerr Wall, Jim L. Mack, Andre Chanderbali, Abdelali Barakat, Erik Wolcott, Haiying Liang, Lena Landherr, Lynn Tomsho, Yi Hu, John Carlson, Hong Ma, Stephan Schuster, Douglas Soltis, Pamela Soltis, Naomi Altman, and Claude dePamphilis. Comparison of next generation sequencing technologies for transcriptome characterization. *BMC Genomics*, 10(1):347+, August 2009.

[180] L. Wang and T. Jiang. On the complexity of multiple sequence alignment. *Journal of computational biology : a journal of computational molecular cell biology*, 1(4):337–48, January 1994.

[181] R. L. Warren. SSAKE 3.0 improved speed, accuracy and contiguity. *Pacific Symposium*, pages 570–570, 2007.

[182] R. L. Warren, G. G. Sutton, S. J. M. Jones, and R. A. Holt. Assembling millions of short dna sequences using ssake. *Bioinformatics*, 23(4):500–501, Feb 2007.

[183] D. Weese, A.K. Emde, T. Rausch, A. Döring, and K. Reinert. Razers a fast read mapping with sensitivity control. *Genome research*, 19(9):1646, 2009.

[184] P. Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th IEEE symposium on switching and automa theory*, pages 1–11, 1973.

[185] J. Wilhelm. Real-time PCR-based method for the estimation of genome sizes. *Nucleic Acids Research*, 31(10):56e–56, May 2003.

[186] J.C. Wooley and Y. Ye. Metagenomics: Facts and artifacts, and computational challenges. *J. Comput. Sci. & Technol*, 25(1), 2010.

[187] D. R. Zerbino. *Genome assembly and comparison using de Bruijn graphs*. PhD thesis, 2009.

[188] D. R. Zerbino and E. Birney. Velvet: algorithms for de novo short read assembly using de bruijn graphs. *Genome Res*, 18(5):821–829, May 2008.

[189] Wenyu Zhang, Jiajia Chen, Yang Yang, Yifei Tang, Jing Shang, and Bairong Shen. A practical comparison of de novo genome assembly software tools for next-generation sequencing technologies. *PloS one*, 6(3):e17915, January 2011.

[190] Fangqing Zhao, Fanggeng Zhao, Tao Li, and Donald a Bryant. A new pheromone trail-based genetic algorithm for comparative genome assembly. *Nucleic acids research*, 36(10):3455–62, June 2008.

[191] Aleksey V Zimin, Douglas R Smith, Granger Sutton, and James a Yorke. Assembly reconciliation. *Bioinformatics (Oxford, England)*, 24(1):42–5, January 2008.

[192] R. Zimmermann. Efficient VLSI Implementation of Modulo $(2^n \pm 1)$ Addition and Multiplication. In *IEEE Symposium on Computer Arithmetic*, pages 158–167, 1999.