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

Ph.D. Thesis

# Paired is better: local assembly algorithms for NGS paired reads and applications to RNA-Seq

Candidate:
Francesca Nadalin

Supervisor:
Prof. Alberto Policriti

May 12, 2014

Author's Web Page: sole.dimi.uniud.it/~francesca.nadalin/

Author's e-mail: francesca.nadalin@uniud.it

Author's address:

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

Referees: Prof. Alessandra Carbone, CNRS - Université Pierre et Marie Curie, Paris, France
Prof. Raffaele Calogero, Università di Torino, Italy

| Candidate's sign: | Supervisor's sign: |
|---|---|
| [illegible] | Alberto Policriti |

# Abstract

The analysis of biological sequences is one of the main research areas of Bioinformatics. Sequencing data are the input for almost all types of studies concerning genomic as well as transcriptomic sequences, and sequencing experiments should be conceived specifically for each type of application.

The challenges posed by fundamental biological questions are usually addressed by firstly aligning or assemblying the *reads* produced by new sequencing technologies. Assembly is the first step when a reference sequence is not available. Alignment of genomic reads towards a known genome is fundamental, *e.g.*, to find the differences among organisms of related species, and to detect mutations proper of the so-called "diseases of the genome". Alignment of transcriptomic reads against a reference genome, allows to detect the expressed genes as well as to annotate and quantify alternative transcripts.

In this thesis we overview the approaches proposed in literature for solving the above mentioned problems. In particular, we deeply analyze the sequence assembly problem, with particular emphasys on genome reconstruction, both from a more theoretical point of view and in light of the characteristics of sequencing data produced by state-of-the-art technologies. We also review the main steps in a pipeline for the analysis of the transcriptome, that is, alignment, assembly, and transcripts quantification, with particular emphasys on the opportunities given by RNA-Seq technologies in enhancing precision.

The thesis is divided in two parts, the first one devoted to the study of local assembly methods for Next Generation Sequencing data, the second one concerning the development of tools for alignment of RNA-Seq reads and transcripts quantification. The permanent theme is the use of paired reads in all fields of applications discussed in this thesis. In particular, we emphasyze the benefits of assemblying inserts from paired reads in a wide range of applications, from *de novo* assembly, to the analysis of RNA.

The main contribution of this thesis lies in the introduction of innovative tools, based on well-studied heuristics fine tuned on the data. Software is always tested to specifically assess the correctness of prediction. The aim is to produce robust methods, that, having low false positives rate, produce a certified output characterized by high specificity.

# Contents

# List of Figures

# List of Tables

# Preface

Very often, the challenges arising when addressing the resolution of biological problems are of two types: (i) coping with difficult formalization of under-determined problems, impossible to solve both exactly and completely, and (ii) working with biased data, whose source of errors is sometimes unpredictable. In this thesis we focus on the analysis of biological sequences, a topic including problems that often require to cope with both (i) and (ii).

Methods employed to solve problems related to biological sequence analysis can be grouped in two classes. The first one envisages the definition of a clean mathematical model mimicking the real problem that, in spite of leading to instances that are computationally intractable, have well-known algorithms for its approximate resolution; procedures falling in this class can be identified as *heuristics on the problem*. Procedures belonging to the second class are instead created starting directly from the data and see as goal the understanding of the system behavior that can be inferred from them; such algorithms may be referred to as *heuristics on the data*.

Both methods have their own merits and defects. On the one hand, a model is easy to manipulate and strengthened solving procedures already exist, however, as the model is different from reality, their output is not intended to be a solution of the real problem. On the other hand, a procedure grounded on the data directly addresses the resolution of the real problem, the shortcoming being the possibility of getting unexpected results when input data differ from the dataset the procedure was designed on.

The contribution of this thesis should be intended as a collection of methods and procedures aiming at solving biological problems in practice. We address the design of algorithms that both work well *on* the data and use as much as possible information coming *from* the data.

# Introduction

The application of Mathematics and Computer Science methods to the study of biological problems cover several areas ranging from the design of algorithms on strings to the modeling of biological systems. Even though first approaches towards such biological applications are dated several decades ago, only recently Computer Science became a fundamental support for biological studies, namely, with the advent of new machines able to "read", *i.e.*, *sequence*, the code of life of every organism.

The term *sequencing* refers to a procedure for extracting information from a nucleic acid or from a polypeptide, in the form of sequences of letters (*i.e.*, nucleotides or amino acids), possibly provided with quality values. Pioneer sequencing experiments were performed by Frederick Sanger in the early 1950s with the sequencing of insulin. In the 1970s he started using the DNA polymerase to clone DNA fragments, and in 1977 he proposed the chain-termination method, or capillary sequencing, that remained the unquestioned sequencing technology for nearly 30 years. The throughput of Sanger method was relatively low and expensive, a shortcoming that limited the application of sequencing technologies in terms of both the number of organisms that could be sequenced and the type of analyses that could be performed.

A new concept of sequencing experiment was proposed less than ten years ago with the advent of Next Generation Sequencing (NGS) technologies. The difference with "Sanger method" mainly lie in a massive level of parallelization, which allowed both to reach high througput (up to millions of bases sequenced in a single run) and to decrease sequencing costs. The production of large amounts of affordable data characterized by single-base resolution and high technical reproducibility leaded to several consequences: (i) many labs could afford sequencing experiments, (ii) advanced computing infrastructures are needed to store and to analyze the data, and (ii) many more types of studies can now be done, whereas the classical analyses could now reach higher sensitivity.

The goal of Bioinformatics is not only to apply computational methods to biology. Nowadays, Bioinformatics has a completely pervasive role in all "omics" disciplines (*e.g.*, genomics, transcriptomics, proteomics, *etc.*). Bioinformatic applications to clinical studies are within our grasp and need fast, trustable, and automatized pipelines. For instance, the 1000 Genomes Project [155] is aimed at sequencing a large number of organisms in order to deeply annotate the patterns of genetic variation in human.

Software tools for bioinformatic applications have become the new "microscope" of life scientists: almost all labs need advanced computer infrastructures to store and analyze the enormous amount of data produced by state-of-the-art sequencing technologies. Thus, the bioinformatic community is constantly called to produce cutting edge instruments able to cope with large datasets when classical problems such as alignment and assembly are addressed. Efficiency and memory constraints have stimulated broad research efforts towards optimal data structures engineering and sophisticated algorithms design to minimize the search time (*e.g.*, hash tables and suffix trees) or the RAM memory required (*e.g.*, FM-index and Burrows-Wheeler transform).

Extraordinary advances in cell biology have matured in the last decade thanks to new sequencing technologies. In particular, the central dogma of molecular biology (Watson, 1958) and the simplified notion of eukaryotic gene as an alternation of exons and introns (Gilbert, 1978), underwent several modifications after discoveries moved by the assembly of the first human genome in 2001 [167], thanks to the possibility of studying several phenomena with the high resolution offered by NGS data. Among the most important ones, we mention the phenomenon of alternative splicing, which allows the production of different trascripts (and hence different proteins) from the same gene; the sequencing of several types of RNAs, that do not encode for proteins but instead influence gene expression; the discovery of epigenetic modifications, that have a central role in gene expression but are not encoded at the genomic level, such as post-transcriptional modifications.

The research of new algorithms that employ NGS data to tackle such bioinformatic problems is continuously evolving and contribute to deepen our knowledge of biological processes with a degree of precision that was impossible only ten years ago.

In this thesis we will exploit the potential of assembly not only in the reconstruction of biological sequences, but also as an instrument able to provide a good starting material for downstream analyses concerned with the understanding of the genome and of the transcriptome. In the first part we will overview genome assembly methods and challenges, and propose a new tool for accurate reconstruction of stretches of DNA identified by couples of sequences known to lay at a certain distance (*i.e.*, paired reads). In the second part, we apply and extend the results previously obtained to the analysis of RNA-Seq data. We will focus on both reads alignment against the reference genome, and transcripts quantification using an existing annotation. The starting point for all applications is local assembly of sequencing data. We will extensively show the benefits of such an approach in three main bioinformatic problems: assembly, alignment, and coverage estimation.

## I.1 Relevant problems and computational challenges in genomics and transcriptomics

The study of the structure of genomes is extremely important for a number of problems, such as evolutionary studies, clinical applications, metagenomics, epigenomics, and many others. From the computational point of view, the genome is a string (or a set of strings) in the alphabet $\{A, C, G, T\}$. Retrieving the complete DNA sequence of an organism using only NGS reads, without any information on the genome, is the so-called *de novo* assembly problem.

There are theoretical limitations that make *de novo* assembly problem intractable. First approaches towards the formalization of sequence assembly were based on the definition of genome as the shortest string that explains the set of reads extracted from it. The corresponding combinatorial problem is called Shortest Superstring Problem (SSP) and it easily seen to reduce to the Traveling Salesman Problem, hence to be NP-complete.

Further insights on the first formulation of the assembly problem pointed out the inconsistence between the minimality of the solution of the SSP and the presence of repeats in the genome. In fact, a solution of SSP, by definition, would report only one copy of each (long) repeat, a feature that is not justified from a biological point of view. A proposed variant to the SSP model consists in introducing some constraints [113], namely, requiring each read to be used in the assembly a minimum number of times, thus allowing for repeats to be reported more than once in the solution. Many other variants have been proposed.

The repeated structure of the genome causes the assembly problem to be NP-hard even if repeats are relatively short (*i.e.*, comparable to reads length). Before NGS technologies appeared on the market, read length was about 800 bp and assembly task was easier. A common belief was that a complex genome structure could be disentangled by providing higher coverages, that is, by increasing the redundancy of data. Unfortunately, with shorter NGS reads, the number of repeats that are not spanned by a read increases. This and several other considerations on the relation between read length and repeat size clarified that genome assembly with short reads is harder as compared to assembly with long ones, irrespective of coverage [126].

Several studies discussed the possibility to disambiguate repeats with NGS data, provided that long-range information on the genome sequence is available. In particular, paired read sequencing has found several applications in genome assembly, both in the detection of correctly assembled areas [116, 134, 179], and in the post-processing phase, called *scaffolding*, in which assembled *contigs* are connected to one another [76].

In the literature, several algorithmic approaches have been proposed to solve the assembly problem, most often based on one of the following schemata: overlap-layout-consensus (OLC), de Brujin graph, and greedy. OLC assemblers [6, 26, 51, 116, 153] build an overlap or string graph and compute a solution in terms of hamiltonian path. Approaches based on de Brujin graphs [13, 20, 22, 23, 96, 134, 154, 179] visit the $k$-mer graph built from the reads and find an eulerian path. Finally, greedy techniques [18, 31, 36, 52, 77, 120, 160, 174] employ a local strategy that extends the assembly as long as overlapping reads exist.

Assessing genome assembly correctness has recently attracted research efforts towards the standardization of validation statistics, starting from recent competitions among assembly tools. Nowadays statistical methods exists that evaluate not only assembly contiguity and completeness, but expecially correctness, with [15, 37, 56, 143] or without [136, 170, 171] a reference genome. The continuing debate on the performances of assembly tools, as a consequence, is motivating further efforts in improving software tools and proposing new algorithmic approaches.

For several types of studies strongly based on resequencing (*e.g.*, structural variants calling and reconstruction), fundamental prerequisite is the availability of a reliable reference sequence. To this aim, genome assembly with NGS reads is, always more frequently, supported by long-range information provided by either pair-end reads and mate-pairs [102, 183] (*e.g.*, paired reads with insert size of $\sim 250 \div 600$ bp up to several kbp, respectively) or Third Generation Sequencing data [39]. Already far from the time when assemblers were busy in a race to reach higher N50, the trend now is to concentrate on sequence correctness.

The assembly of the genome is only the first step towards the comprehension of cell functions. Identification of coding regions (*i.e.*, genes) and quantification of their products (*i.e.*, transcripts) pose further bioinformatic challenges that are faced with *ad hoc* algorithms. In eukaryotes, genes are organized in coding parts (*i.e.*, exons) interspersed by non-coding ones (*i.e.*, introns), the latter being removed after transcription. Identification of coding regions is challenging because alternative splicing allows for the production of different transcripts from the same gene.

NGS reads extracted from a transcriptome, also called RNA-Seq reads, are on the basis of numerous downstream analyses focused on genome annotation and transcripts quantification. Issues concerned with the annotation problem are: understanding genes structure, finding all types of expressed transcripts, detecting alternative splicing events, and calling post-transcriptional modifications. From a computational point of view, these problems can all be reconducted to assembly or alignment of RNA-Seq reads.

The reconstruction of the transcriptome *de novo* is performed with tools mainly derived from existing instruments for genome assembly [12,25,150], even though the nature of the problem is intrinsically different [50]. Tools developed from scratch exist as well [45, 132]. The lack of coverage information cannot be used here to disambiguate the assembly and its non-uniformity along the transcriptome requires *ad hoc* designed heuristics. The availability of a reliable reference genome can often ease transcriptome assembly task [49, 97, 118, 162, 165] that, in this case, is based on RNA-Seq reads alignment against the genome to find putative exons.

Accurate RNA-Seq reads alignment against the reference genome is the first step towards reference-guided transcriptome assembly, novel splicing sites detection, and transcripts quantification [173]. As RNA-Seq reads come from transcripts, mapping against the genomic sequence should envisage the presence of large gaps, when a read spans an intron. The importance of RNA-Seq reads alignment for the analysis of the transcriptome stimulated several research efforts [1, 41]. In the literature two main approaches are found: *exon-first* [7, 30, 32, 75, 98, 164, 172, 181], which use unspliced reads as anchors for exons detection and spliced reads for isoforms identification, and *seed-and-extend* [4,8,17,34,99,103,135,176,177], which cut each read in seeds, align them separately, and identify their correct placement using global mapping information.

Quantification is performed employing RNA-Seq reads alignment, whereas the task of decomposing cumulative exons coverage into the expression levels of alternative transcripts is usually addressed with a least square model that minimizes the discrepancy between prediction (*i.e.*, expression levels) and observation (*i.e.*, exons coverage).

Very recently, a new research front emerged, addressing systematic evaluation of software tools for RNA-Seq data analysis [40, 158]. These studies are relevant in a scenario in which, lacking any standardized method to compare software instruments, the choice of the most suitable tool for a specific bioinformatic application is nearly casual or, at most, is based on current trends. To the best of our knowledge, ground truth-independent evaluation metrics have not been proposed yet, but, analogously to GAGE competition for genome assembly evaluation [58], efforts towards unbiased comparison of state-of-the-art tools for the analysis of the transcriptome are laying the foundations for the development of a new research, targeted to RNA-Seq reads alignment, isoforms assembly, and transcriptome quantification assessment.

## I.2 The contribution of this thesis

With *paired read* we denote a couple of nucleotide sequences known to lay at an estimated distance in the genome and with a given orientation. In this thesis we will make extensive use of pair information. The lack of contiguity proper of short NGS reads can be partially accomplished by this type of information. Paired reads are employed both to disambiguate complex structures during assembly and to connect the assembled sequences during the finishing (*i.e.*, scaffolding) phase. Moreover, paired reads can aid accurate reads mapping against a reference, by identifying pair-consistent alignments.

However, *pair information* is weaker than complete long range information (*i.e.*, the knowledge of the entire sequence lying between the two mates of a paired read) and, in some cases, single-base resolution would be extremely helpful. For instance, a single paired read cannot be used alone to assess the correctness of an assembled sequence, nor to connect two separate parts of an assembly. In other words, it is often necessary to provide a threshold amount of evidences, in terms of paired reads, to infer that two sequences are adjacent one another in the genome. Moreover, alignment of RNA-Seq reads against a reference genome is only weakly supported by insert-size information.

In this thesis we deeply exploit the advantages of using pair information for various applications, ranging from genomics to transcriptomics. In Part I we will propose a *de novo* assembly tool, called GapFiller [123], to reconstruct inserts of a paired read library. It is based on a seed-and-extend schema, in which every read is selected as seed and iteratively extended until its mate is possibly found. During assembly, a sequence is extended with a consensus computed on overlapping reads. The search is speeded-up thanks to a hash function that guarantees a low false positive rate [169]. The extension phase employs refined clustering and trimming heuristics to guarantee assembly correctness. We will discuss the advantages of having longer in-silico built sequences in contexts in which sequence correctness is of crucial importance. In particular, we will employ GapFiller output contigs for genome assembly, as in [102, 182], and for structural variants reconstruction.

Part II is devoted to the analysis of RNA-Seq data. In particular, we will introduce two new approaches for RNA-Seq paired reads alignment and transcripts quantification, respectively, both taking advantage of a preprocessing phase that allows transcripts reconstruction from RNA-Seq paired reads [122]. The first pipeline is structured in three steps: (i) inserts are assembled with GapFiller and, for each of them, the layout information is stored (*i.e.*, position and orientation of each read used for the assembly); (ii) inserts are aligned against the reference genome and mapping is refined around splicing junctions; and (iii) reads alignment is retrieved from inserts layout and multiple alignments are either merged or disambiguated using pair information. Such insert-guided alignment procedure is aimed at overcoming possible ambiguities or imprecisions arising from NGS reads alignment around splicing junctions, using neighboring sequence information.

In the second method, similarly to most approaches to the estimation of expression levels of alternative transcripts, we associate each gene a linear system of equations $\mathbf{Mx} = \mathbf{c}$, which puts in relation expression levels ($\mathbf{x}$) with measured exons coverages ($\mathbf{c}$). In order to compute the components of $\mathbf{c}$, we will employ the alignment pipeline described above. The difference with state-of-the-art methods, which are usually based on the resolution of a least square problem that minimizes $||\mathbf{M}\hat{\mathbf{x}} - \mathbf{c}||$, where $\hat{\mathbf{x}}$ is an estimate of $\mathbf{x}$, lie in two aspects. First, each exon is associated a quality value, depending on length, coverage uniformity, and coverage consistency with respect to other exons of the gene. Second, a solution to the linear system is found by solving a minimal sub-system, whose equations are associated to high quality exons, for which a unique solution exists.

Our aim is to provide methods that guarantee high specificity, that is, correctness of predictions. For this reason, we will always test correctness of the proposed tools through experiments on simulated data, but tests on real sequencing data will be performed as well. Moreover, we will compare the performances with state-of-the-art tools for analogous problems.

# 1 Preliminaries

The *cell* is the smallest entity containing information for the development and the functionalities of every living organism, and the *genome* is the collection of all DNA (Dehoxiribonucleic Acid) molecules that are found in every cell. RNA (Ribonucleic Acid) is the basis on which protein synthesis can occur.

The study of both DNA and RNA provides the fundamental material to answer a wide multiplicity of biological questions.

One of the goals of the comparison of the genome of an organism with respect to that of a different (although related) species, is the understanding of evolution through the identification of events involved in the speciation process, that is, of the *acquired mutations*.

Finding the differences in the DNA sequence within cells belonging to the same organism (or to multiple organisms of the same species) is often related to the comprehension of the causes of the so-called "diseases of the genome", which are driven by *somatic mutations*. In this context, crucial information is supplied by the outcomes of the analysis of the RNA content of the cells of interest. The ability of detecting both differences in RNA content among diverse samples, or within the same cells at different conditions, and aberrant DNA-to-RNA processes, is fundamental and require a continuous interchange of the knowledge matured on DNA and RNA.

In the first part of this chapter, an overview of some fundamental concepts in genomics is presented. In the second part, we describe the basic principles and applications of the technologies employed to produce the genomic data, which constitute the row matter for subsequent biological analyses.

## 1.1 DNA, genes, and RNA

DNA is a polymer that, up to mutations, remains almost unchanged during cell life, hence we will refer to *the* genome of an organism as the set of DNA molecules contained in its cells. RNA is produced from DNA and, differently from DNA, the RNA composition widely changes depending on the cell life stage, on the environment, and on the tissue the cell belongs to.

An important distinction among cells is based on the notions of *prokaryote* and *eukariote*, depending on the absence or presence of a nucleus, respectively. Within eukaryotic cells, each part (*i.e.*, *organelle*), is separated from the cytoplasm by a membrane. The organelles containing DNA are the nucleus, the mitochondrion, and the chloroplast. Within the nucleus, the genome is structured into linear or circular DNA fragments named

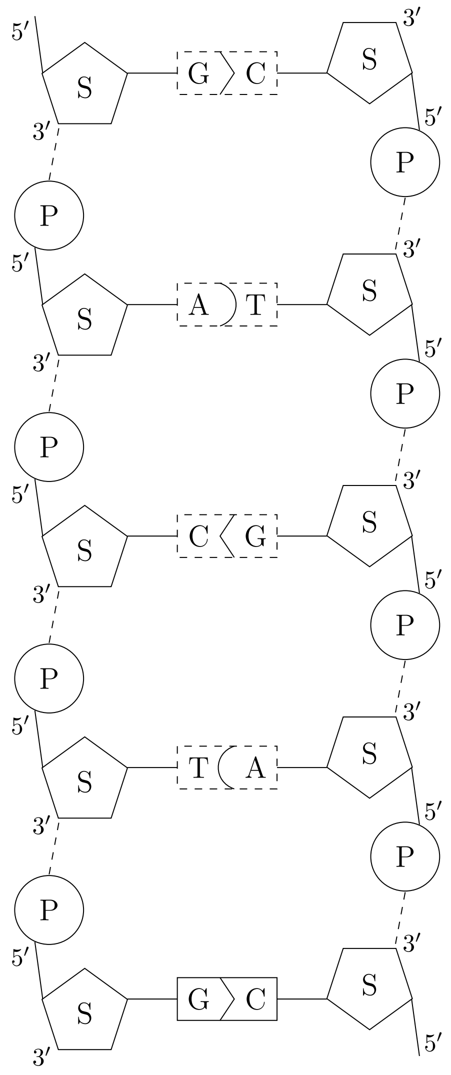


**Figure 1.1:** The DNA structure of the polymer GACTG. The sugar and the phosphate group are represented with S and P, respectively.

*chromosomes*, which are embedded around a specific type of proteins (*i.e.*, histones) to form *chromatine*. Among linear chromosomes, shorter DNA sequences, called *telomers*, are found, and their function is to protect DNA from degradation. *Ribosomes* are the organelles responsible of protein synthesis.

### 1.1.1 The structure of DNA

The DNA, or *deoxiribonucleic acid*, is a macromolecule constituted by two anti-parallel polymers. The two DNA strands owe their names to Watson and Crick, who first theorized the DNA structure by giving it the shape of a double helics [175].

A single DNA strand is composed by a chain of nucleotides, each of them constituted by three elements: a sugar (deoxiribose), a phosphate group, and a nitrogen base (see also Figure 1.1). There are four nitrogen bases: adenine (A), guanine (G), cytosine (C),

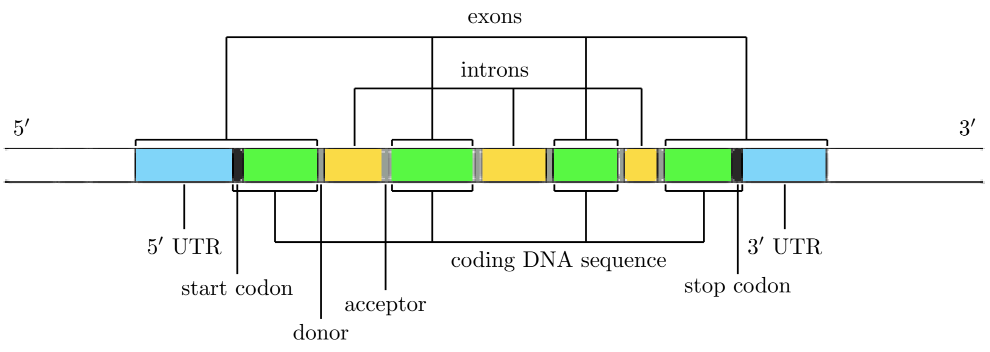


**Figure 1.2:** The structure of an eukaryotic gene.

and tymine (T). Adenine and guanine are called purines, while cytosine and tymine are dubbed pyrimidines.

Each strand is associated a direction: $5' - 3'$ is called *upstream*, whereas $3' - 5'$ is called *downstream*. The two strands stay together thanks to the hydrogen bonds among complementary bases, namely, A-T and G-C. This means that if one strand is known, the other one can be obtained by reading the nucleotide sequence towards the opposite direction and substituting every base with its complementary one. The sequence defined in this way is called the *reverse complement*. The process that builds a complementary DNA sequence from a template is called DNA duplication.

DNA is constituted by both coding and non-coding regions, depending on the fact of encoding for proteins or not, respectively. Within eukaryotic genomes, coding regions are located in correspondence of genes, which, in turn, are constituted by protein coding sequences, called *exons*, interspersed by non-coding ones, called *introns* (see also Figure 1.2).

Exons are the parts of an eukaryotic gene that contain protein coding sequence. At the beginning of the first exon and at the end of the last exon of a gene, respectively, two *untranslated regions* (UTR) are found. UTRs are transcribed into mature RNA but they are not translated into proteins. The parts of exons that are converted into amino acids constitutes the *coding DNA sequence* (*i.e.*, CDS).

The various parts of a gene are separated from one another by means of short oligonucleotides. At the beginning and at the end of the CDS, respectively, a start codon and a stop codon are found. The canonical oligonucleotide pair identifying the boundaries of the CDS is ATG-TAG, but non-canonical pairs can occur as well (*e.g.*, GTG-TGA and TTG-TAA). In correspondence of exon-intron and intron-exon boundaries, within the intronic sequence, *donor* and *acceptor* dinucleotides are found. The canonical pair is GT-AG, but other dinucleotides may occur, *e.g.*, GC-AG and AT-AC.

### 1.1.2 From DNA to RNA: the role of genes

The RNA is a single-stranded molecule constituted by a chain of nucleotides. An RNA nucleotide differs from a DNA one because the deoxiribose is replaced by *ribose*, and the nitrogen base *uracile* (U) is found in place of the tymine (T). Due to its single-stranded structure, RNA is not able to replicate itself and is produced from DNA in correspondence of genes.

In eukaryotic cells, different types of RNA exist [173] and they can be grouped in two main classes: *coding RNA* and *non-coding RNA*. Coding RNA is the type of RNA that is subsequently translated into proteins; it is also called *messenger RNA* (mRNA) and is responsible of carrying pieces of RNA from the nucleus to the ribosomes, where protein synthesis occur. Non-coding RNAs (ncRNAs) are not translated into proteins, instead they have a *functional* role within the cell. Several types of ncRNA exist, some examples are the following ones: *transfer RNA* (tRNA), which carries information, in terms of, single amino acids, from the nucleus to the cytoplasm; *ribosomal RNA* (rRNA), which provides the "instructions" for aminoacids encoding; and *small RNAs*, *i.e.*, short sequences of $\sim 20$ bp that regulate gene expression and are located towards the upstream direction with respect to the gene they refer to.

A single RNA molecule produced from a gene is called a *transcript*. Transcripts are usually produced along the gene towards the *sense strand* (*i.e.*, the $5' - 3'$ direction), that is also known as the coding strand, but *antisense* transcripts can also be produced (*i.e.*, towards the $3' - 5'$ direction). Recent studies demonstrated that high percentages of human genes (*i.e.* approximately three-quarters) have transcription potential [33]. After transcription, the splicing process occurs in order to remove introns and gather the so-called *mature RNA*. The set of all mature RNA molecules produced by a cell is dubbed the *transcriptome*.

The various parts of a gene, and the specific DNA sequences placed outside of it, play an important role in the processes involved in the production of the mature RNA starting from a gene. The presence of the *initiation factor* (or *promoter*), located on the upstream direction, supplies the RNA polymerase the instructions for starting transcription, whereas the *(distal) regulatory elements* control the amount of RNA molecules to be produced [81]. Donor and acceptor dinucleotides are involved in the splicing process and determine the location of the introns to be removed from the pre-RNA, whereas start and stop codons identify the boundaries of the protein coding sequence.

The presence of specific sequences separating the different parts of an eukaryotic gene may suggest *ab initio* prediction methods, that is, use information from the DNA sequence to infer genes loci. However, methods based on the genomic sequence only shown low effectiveness in the context of the genome annotation problem [85]. In fact, fundamental information for this type of analyses is provided by the transcripts produced from each gene [133], as well as the technologies employed to extract data from RNA (see also Section 1.2.3).

A problem that is put aside to the determination of genes structure consists in the estimation of the amount of transcripts produced from a gene, called *expression level*. The transcription process can lead to different amounts of transcripts for a single gene, depending on the tissue and on the environment, and the expression levels of different genes in the same condition can vary too. Indeed, some genes may not be expressed at all, hence reconstructing the complete set of RNA molecules produced at a specific cell life stage and under given environmental conditions is a very complex task. An overview on the transcriptome assembly problem can be found in Section 5.3.

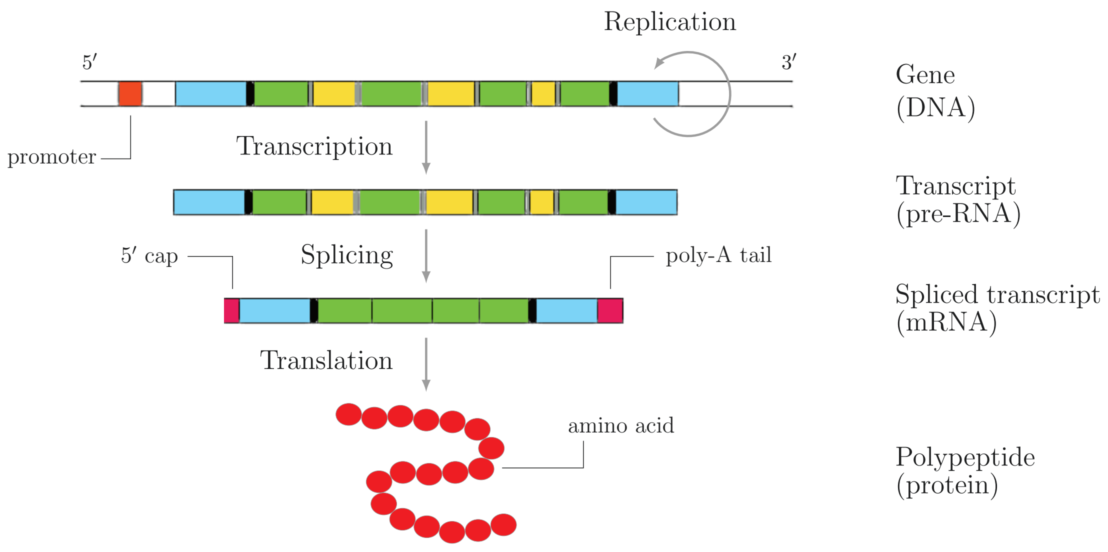


**Figure 1.3:** The central dogma of molecular biology.

### 1.1.3 The central dogma of molecular biology VS the role of alternative splicing in gene expression

The *central dogma of molecular biology* states that three fundamental processes occur within a cell (see Figure 1.3): DNA replication (or duplication), DNA transcription (which is followed by RNA splicing), and RNA translation.

DNA is able to replicate itself through the *DNA duplication* process, made by the enzyme DNA polymerase. Such enzyme traverses a DNA template towards $5'-3'$ direction and creates a new stretch of DNA corresponding to the reverse complement of the template.

RNA can be created from DNA by means of the enzyme RNA polymerase, through the process called *transcription*. Transcription converts stretches of DNA into pre-RNA, through the enzyme RNA polimerase, and adds a $5'$ CAP (a guanine) immediately before the $5'$ UTR. During or after transcription, the splicing process occurs, turning pre-RNA into mature RNA through the removal of introns and eventually adding a poly-A tail after the $3'$ UTR.

Finally, *translation* is the process that synthesizes proteins from mature RNA by converting triples of nucleotides into amino acids.

The central dogma also states that the DNA sequence contains all information for the cell life and functions and that the processes changing the molecular structure (*i.e.*, transcription and translation) are assumed to have *one direction* and that the only possible path is DNA $\rightarrow$ RNA $\rightarrow$ protein.

The variety of functions found in mammals suggested that their gene content should have been sensibly higher than that of simpler organisms, such as bacteria (see Table 1.1). Within the human genome, an amount of roughly $100,000$ genes was estimated in order to explain the multitude of functions carried out by human cells. Since just few years before the completion of the Human Genome Project [62], very little was known about the structure of the genomes and the relation among the processes occurring in the cell. The assembly of the first human genome [167] clearly showed that the estimated number of genes seems is not related with the organisms complexity, hence that a 1-

to-1 correspondence between gene and protein is not possible. This belief was found to be false as now we estimate that the human genome contains less than $25,000$ genes, a discovery suggesting that a more subtle characterization of the processes happening within eukaryotic cells should be provided. Indeed, the complexity of genome structure is so huge that even today the human genome, which is the most studied one, is yet to be completely revealed.

Several studies carried out some doubts on the validity of the central dogma (see [83]) also because it is not able to explain a lot of processes happening within a cell, for instance: (i) RNA-to-DNA conversions are allowed, two examples of such a phenomenon are the replication processes carried out by retroviruses and retrotransposons; (ii) several types of RNA do not encode for proteins and may have, instead, a regulatory function; (iii) some mechanisms are not encoded within the DNA sequence itself, but instead are dependent on epigenomic information. The ENCODE Project [59] played a crucial role in the characterization of transcripts and helped in understanding the human genome *function*.

**Table 1.1:** Genome length and genes content of some eukaryotes

| Organism | Genome length (kbp) | Number of genes[a] |
|---|---:|---:|
| *Homo sapiens*[b] | $3,324,592$ | $20,769$ |
| *Mus musculus*[b] | $3,480,528$ | $23,139$ |
| *Danio rerio*[b] | $1,505,582$ | $26,247$ |
| *Drosophila melanogaster*[b] | $168,737$ | $13,937$ |
| *Arabidopsis thaliana*[c] | $119,146$ | $27,411$ |
| *Picea albies*[d] | $19,600,000$ | $28,354$ |
| *Caenorhabditis elegans*[b] | $103,022$ | $20,532$ |
| *Saccharomyces cerevisiae*[b] | $12,157$ | $6,692$ |

[a] protein coding genes
[b] source: Ensembl [67]
[c] source: TAIR [65]
[d] source: The Norway spruce genome sequence [130]

## Alternative splicing

Given two sequences of pre-RNA transcribed from the same region of DNA, the resulting mature RNA molecules, *i.e.*, after splicing, can differ from one another. More precisely, the splicing process is not limited to the removal of intronic sequences, in fact it can cause exons or part of exons to be removed from the pre-RNA. In such a case, such exonic sequences are not present in the resulting mature RNA. This phenomenon is named *alternative splicing* and can show different *patterns* [112] (see Figure 1.4), which are possibly combined to each other.

Alternative splicing makes it possible to generate different transcripts, and hence, in case of mRNA, to encode for different proteins, starting from the same gene. Each feasible pattern for a given gene is called *gene isoform*, which can be associated to a specific *protein isoform*. Different types of transcripts produced from the same gene are often called *alternative transcripts*.

Within mammalian genomes, a high percentage of genes allow multiple gene isoforms (*e.g.*, $> 90\%$ in human) characterized by high splicing complexity. Splicing patterns are

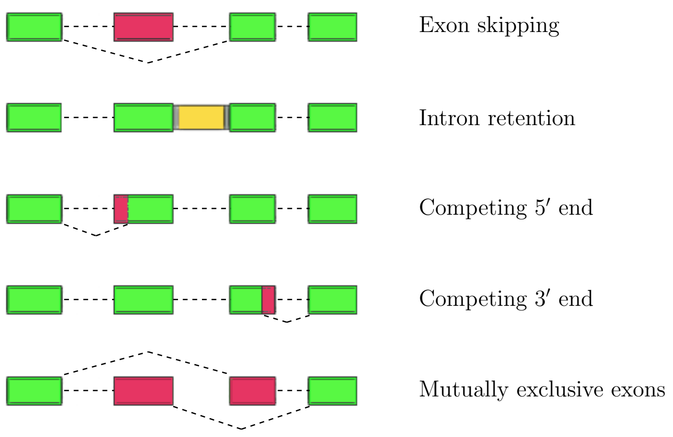


**Figure 1.4:** Alternative splicing patterns

very different, for instance, among animals and plants. The complexity of an organism is more related to the alternative splicing pattern, rather than to the gene content of its genome [129] (see also Table 1.1).

The splicing pattern complexity makes it difficult both to identify the different isoforms that are expressed by a single gene at the same time, and to quantify the expression level of alternative transcripts. Moreover, as already mentioned, different patterns are not mutually exclusive, *e.g.*, exon skipping can be combined with (possibly multiple) events of competing $5'$ or $3'$ ends.

The concept of *exon* becomes ambiguous, because, as stated in Section 1.1.2, the mature RNA is composed by the exons after the removal of introns. With this definition, multiple choices are possible and each isoform has its own exons. A definition by Gerstein and colleagues [42] says that "a gene is a union of genomic sequences encoding a coherent set of potentially overlapping functional products [...] final, functional gene products (rather than intermediate transcripts) should be used to group together entities associated with a single gene". In light of alternative splicing, an *operational gene definition* seems to be the most sensible one. Pesole [133] propose the following definition: "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)".

The above definitions state that the structure of a gene depends on the products of the transcription process, hence the notions of gene and transcript are strongly connected. Such an idea also suggests that the methods employed to analyze RNA are important to study the gene structure, in particular the technologies allowing single-base resolution (*i.e.*, RNA-Seq, see Section 1.2.3) supply a powerful method to predict alternative splicing sites, which would be nearly impracticable using *ab initio* prediction methods.

The ability of a gene to produce different proteins explains the complexity of cell functions and solves the paradox deriving from the central dogma of molecular biology, which states the existence of a 1-to-1 correspondence between genes and proteins. The complexity of the gene structure, due to the presence of alternative splicing patterns, of intronic genes, of paralogs, and of non-canonical splicing dinucleotides, pose several challenges to both gene isoforms prediction and expression levels estimation and require both *ad hoc* computational methods and deeper analysis of the data.

However, the central dogma of molecular biology should not be hopelessly rejected; in fact, it still represents the *main* channel through which the DNA is actually expressed. It has been observed that, for differentially spliced genes, there exists an isoform whose expression level dominates the other ones, at a given condition [33]. Moreover, there are evidences for a gene to have one predominant transcript, that is (almost always) the one that dominates the others [44].

## 1.2 Sequencing: extracting information from DNA and RNA

Sequencing is the process of extracting information from a nucleic acid in the form of sequences of bases. The problem of reading the complete DNA sequence of an organism is a task that strongly depends on the sequencing technology used. Due to technological limits, it is impossible to estract a *single*, *complete* and *exact* genomic sequence from a cell.

A *DNA sequencer* is a machine that reads stretches of DNA and reports its sequence of bases. It allows automatized an parallel sequencing experiments. Each sequence tag outputted by a sequencer is called *read*, and each read contains the following information: an (univocal) ID, a sequence of bases $s$ in the alphabet $\Sigma = \{A, T, G, C\}$, and a quality sequence $q$ containing a value for each base of $s$. Both $s$ and $q$ have the same length $|s|$ and, for each $i = 0, \ldots, |s| - 1$, $q[i]$ is a function of the probability $p_i$ that the base $s[i]$ is incorrect. Depending on the sequencing technology used, the quality value is defined in different ways, for example:

$$q[i] := \begin{cases} -10\log_{10} p_i & \text{(Sanger)} \\ -10\log_{10} \frac{p_i}{1-p_i} & \text{(Illumina Solexa)} \end{cases} \tag{1.2.1}$$

A base within a read differing from the original sequence is called *sequencing error*.

The currently used approach is the Whole Genome Sequencing (WGS) [10], which consists in extracting reads from random portions throughout the whole genome. The opposite approach consists in employing physical maps, *i.e.*, selecting (overlapping) portions on the genome, sequencing each of them separately, and subsequently merging them.

Sequencing can be used to extract information from both DNA and RNA and the data produced are the basic material for a wide plethora of analyses. Different technologies provide different types of data, both in terms of read length, nucleotide accuracy, and error profile.

DNA sequencing approaches can be grouped in two categories: *de novo* sequencing and resequencing. An important application of *de novo* DNA sequencing is *de novo* genome assembly, that is the task of reconstructing the complete DNA sequence of an organism starting from a set of reads, without using any other information. DNA resequencing instead finds applications both in the analyses related to the comparison between the genomics sequences of two organisms of the same species (expecially aimed at the discovery of somatic mutations), or between two different species (mostly concerned with evolutionary studies or reference-guided assembly).

RNA sequencing is employed for reference-guided or *de novo* transcriptome assembly (depending on whether a reference genome is available or not), genome annotation, and gene expression profiling. The latter ones require a reference genome and the methods used for isoforms expression level estimation may also require a (partial) genome annotation, which is possibly refined using information provided by RNA sequencing data.

### 1.2.1 Sequencing technologies

The output of a sequencer is a set of reads obtained from a genome that is usually oversampled. The length of a read, and in general of a nucleotide sequence, is measured in *base pairs* (bp).

The reads produced by sequencers are usually much shorter than the genome length; for instance, bacteria genomes range from ∼2 Mbp to ∼5 Mbp, human genome is ∼3 Gbp, and plant genomes range from ∼150 Mbp (*Arabidopsis thaliana*) to >100 Gbp (*Paris japonica*). Read length, instead, varies depending on the technology used: nowadays it ranges from 75 bp (sequencing by ligation) to an average of 1500 bp (single molecule sequencing). For this reason, oversampling is necessary in order to be able to reconstruct the genome of an organism *de novo*, but also to collect a reasonable number of evidences for the analyses envisaged by resequencing projects [145, 146].

Sequencing technologies differ from one another in terms of read length, throughput, and per-base-cost (see [108, 151]). So far, three families of sequencing technologies exist, each of them showing main traits that signed a significant break with the preceding one. First generation is the so-called traditional sequencing method, whereas Second and Third are commonly addressed as Next Generation Sequencing. Third generation differs from Second generation by means of the ability of sequencing a whole, single molecule in real time.

First Generation Sequencing, also called *capillary sequencing*, was introduced in the 70s by Sanger [144]. This technology is able to output sequences of length up to $\sim 9,000$ bp, they are very expensive to produce and they have very low throughput, because the amount of fragments that can be sequenced in parallel is very low. The high costs demanded by traditional technologies do not only depend on the quantities of reagents, but also on the large amount of human work required for each sequencing experiment (only a number of 96 up to 384 capillaries can be sequenced at the same time) and infrastructures maintenance [151]. The protocol requires first fragmenting the DNA and then cloning the fragments into pools of plasmids, which then undergo cycle sequencing reactions. The final data consist of reads of average length $\sim$ 800 bp having up to 99.999% accuracy [151].

The traditional sequencing technology proposed by Sanger were the only one available for approximately thirty years. In the early 2000s new technologies emerged, based on different chemistry and with the aim of automatizing sequencing experiments in order to make them affordable for a larger community of scientists. These *Next Generation Sequencing* (NGS) technologies are grouped into two categories: Second Generation, and Third Generation.

Second Generation Sequencing technologies have been introduced around 2004 with *pyrosequencing* (Roche 454 [63]), followed by *sequencing by synthesis* in 2006 (Illumina Solexa [71]), and *sequencing by ligation* in 2007 (Applied Biosystems SOLiD [64]). Since the time they were commercialized [108], these technologies showed to be far cheaper and to have much higher throughput compared to Sanger sequencing, however the reads produced are shorter (see also Table 1.2).

Several studies addressing the potentialities, as well as the weaknesses, of NGS technologies have appeared in the last few years [2, 101, 108, 109]. Even though read length is

**Table 1.2:** Read length, throughput, and cost of major sequencing technologies

| Instrument | Read length (bp) | Throughput per run | Run time | Cost ($/Gbp) |
|---|---|---|---|---|
| Sanger 3730xl[a] | $400 \div 900$ | $1.9 \div 84$ Kbp | $20' \div 3$h | $2,400,000$ |
| Roche 454GS FLX4 [a] | 700 | 0.7 Gbp | 24h | $10,000$ |
| Illumina HiSeq 2000[a,b] | up to 150 | 600 Gbp | 11 days | 41 |
| Applied Biosystems SOLiDv4[a] | 50 | 120 Gbp | $7 \div 14$ days (SE-PE) | 130 |
| Pacific Biosciences RS[b] | $1,500$ | 100 Mbp | 2h | $2,000$ |

[a] source: Liu *et al.* [101]
[b] source: Quail *et al.* [139]

shorter compared to that of traditional technologies, NGS became the unquestioned sequencing strategy because it allows single base resolution, high throughput, and extremely lower costs [151]. Resequencing and gene expression profiling find lots of advantages in NGS, while *de novo* assembly is still difficult due to shorter read length, sequencing errors, and PCR artifacts (in the case of 454 and Illumina technologies). However, as the sequencing technologies are continuously improving, accurate and complete *de novo* assembly with short reads has been possible since few years ago even for large genomes [11, 43].

A task to be addressed when dealing with NGS data is to analyze the type and the frequency of sequencing errors and, possibly, to correct the reads in order to reach an ideal $k$-mer distribution (see also Section 2.2.2). It has been shown that the most common sequencing errors are different for each technology, for instance, insertions and deletions are proper of 454 and HeliScope, whereas nucleotide substitution is the type of error occurring in Illumina data [151]. Also, error profile, that is, the error frequency as a function of the position in the read, may vary among two different releases of the same vendor company.

Third Generation Sequencing has been introduced in 2009 [39], it is known as the "True" Single Molecule Sequencing and the main vendor is Pacific Biosciences (PacBio). This technology is able to produce uniformly distributed, very long reads (more than 1000 bp) [21]. The important shortcoming is that these reads are high error-affected (say, nucleotide accuracy is a little less than 85% [27, 140]), although sequencing errors are uniformly distributed along the read. Some approaches have been proposed to correct these long reads, for example by aligning reads from Second Generation Sequencing experiments against them [84].

In this thesis, the algorithms and tools presented have been applied to Illumina sequencing data. Hence, we go into more details with the Illumina sequencing technology in the following section.

### Next Generation Sequencing-by-synthesis: the Illumina protocol

The most widely used technology nowadays is Illumina [71], which provides the so-called *sequencing-by-synthesis* technology. The technology has improved throughout the years, yelding reads up to 300 bp long (whereas the first reads were 25-36 bp long), thus opening the successful application of such a technology to the assembly of extremely repetitive and long genomes, especially when paired reads are provided [70].

The Illumina protocol is structured in three main steps (see also Figure 1.5): (i) library preparation, (ii) amplification, and (iii) sequencing. Step (i) consists in cutting the genome

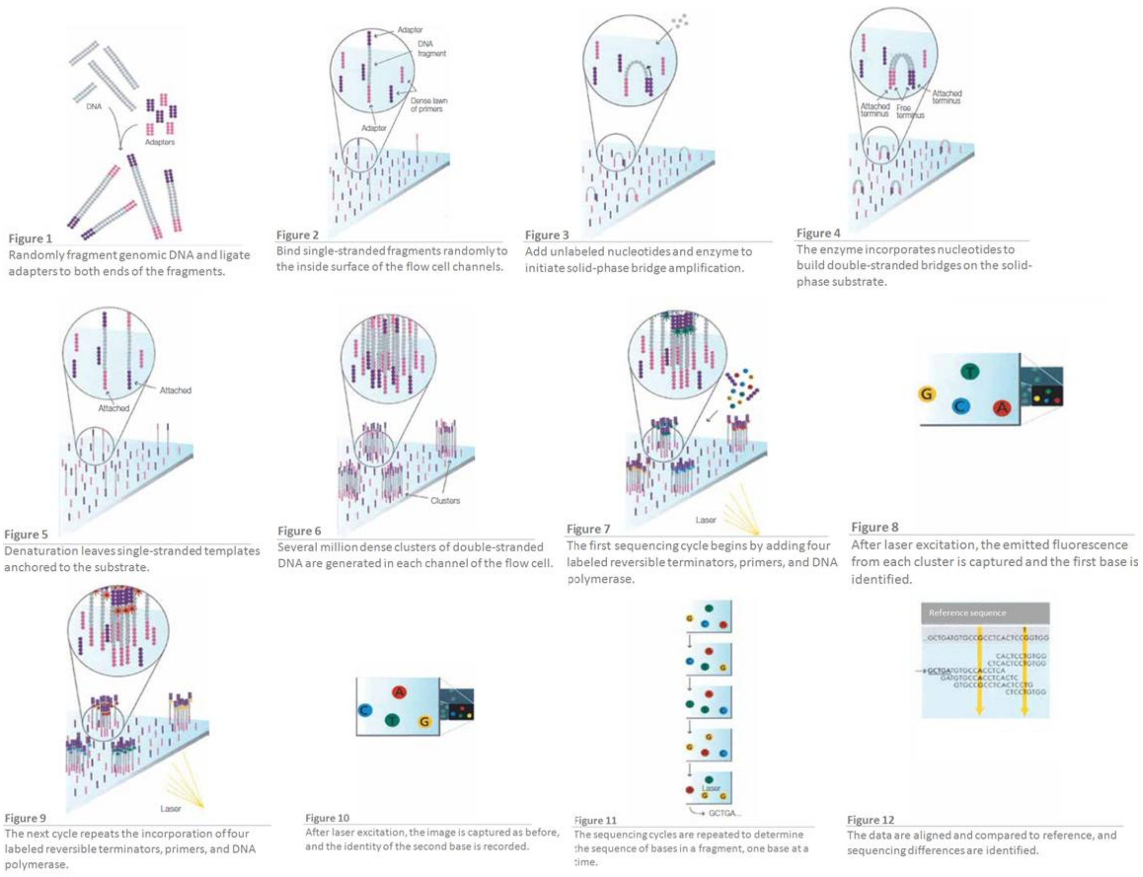


Figure 1
Randomly fragment genomic DNA and ligate adapters to both ends of the fragments.

Figure 2
Bind single-stranded fragments randomly to the inside surface of the flow cell channels.

Figure 3
Add unlabeled nucleotides and enzyme to initiate solid-phase bridge amplification.

Figure 4
The enzyme incorporates nucleotides to build double-stranded bridges on the solid-phase substrate.

Figure 5
Denaturation leaves single-stranded templates anchored to the substrate.

Figure 6
Several million dense clusters of double-stranded DNA are generated in each channel of the flow cell.

Figure 7
The first sequencing cycle begins by adding four labeled reversible terminators, primers, and DNA polymerase.

Figure 8
After laser excitation, the emitted fluorescence from each cluster is captured and the first base is identified.

Figure 9
The next cycle repeats the incorporation of four labeled reversible terminators, primers, and DNA polymerase.

Figure 10
After laser excitation, the image is captured as before, and the identity of the second base is recorded.

Figure 11
The sequencing cycles are repeated to determine the sequence of bases in a fragment, one base at a time.

Figure 12
The data are aligned and compared to reference, and sequencing differences are identified.

**Figure 1.5:** The Illumina sequencing protocol (Source: Illumina [71]).

into smaller pieces, called *fragments*. After fragments of the desired length are selected, they are denaturated and both their tips are linked to an *adapter*, that is a short nucleotide sequence that is attached at the beginning of each fragment. Step (ii) consists in firstly attach the adapters on a solid support, that is provided with sequences that are the reverse complement of the adapters. This is required in order for the fragments to bind with the support. After that, fragments are amplified in order to produce multiple copies of the same fragment. Finally, during step (iii) the fragments are sequenced. A dye is employed to create a reverse complement sequence of the fragment to be read, base-per-base, within the so-called *base-calling* step. In this phase, each read is extended of 1 nucleotide at each cycle. As the fragment has been amplified ($\sim$ 1000 times), each base is created multiple times. The reported base in a fixed position of the read is that occurring the highest number of times, provided with a quality value representing the probability of that base to be correct. Fragments are read from the 5' end of the read and, after each base is read and before reading the next one, the amplified fragments are cleaved. As the residues of the cleavage procedure roll by, the read's quality is espected to decrease while going to 3' direction. Technology improvements allow to continuously increase the precision and hence to reach higher read lengths.

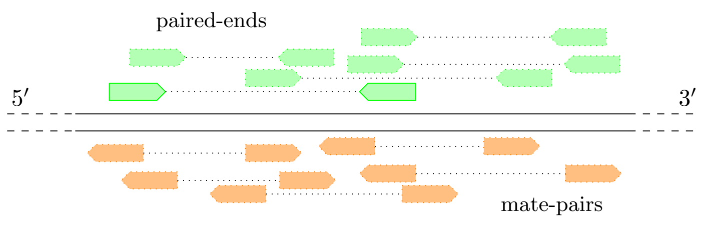


**Figure 1.6:** Pairwise strand of paired-end and mate-pair read libraries.

### 1.2.2 Reads libraries

Beyond single-read sequencing, it is possible to generate pairs of sequences, called *paired reads*, laying at an estimated distance and with known orientation. The process is known as *double-barrel sequencing* and has been introduced yet with Sanger technology. With Illumina sequencing, paired reads libraries are produced by bridging the same fragment, previously attached to adapter sequences on both tips, within the solid support used then for amplification. This method requires each read to "know" the reads that is coupled to it in a pair.

Data produced with double-barrel sequencing are of two types: *paired-ends* and *mate-pairs*. The sequence from which the paired reads are generated is called *insert*, and the distance between the two further ends of the two reads (*mates*) is called *insert size*. Paired-ends have short insert size (*e.g.*, 200-500 bp for Illumina) and reads are inner stranded, whereas mate-pairs have usually much longer insert size (several kbp) and reads are outer stranded (see Figure 1.6).

Paired reads are more expensive to produce with respect to single reads data but their availability is of unquestionable help in various contexts. Information provided by paired reads is useful both in *de novo* assembly (*e.g.*, to detect misassemblies and perform scaffolding) and in resequencing projects (*e.g.*, identify structural variants).

### 1.2.3 RNA-Seq: NGS for transcriptomics

Before the advent of NGS technologies, gene expression was usually analysed with microarrays [147] and Serial Analysis of Gene Expression (SAGE, see [166]). Such methods, however, do not allow to detect new splice junctions or to discover novel genes. Sequencing of Expressed Sequence Tags (ESTs) was the only available method for *de novo* transcriptome reconstruction. RNA molecules are firstly converted to cDNA and then attached to BAC clones to be sequenced. However, the application of ESTs Sequencing was limited by the high costs required [109, 118].

RNA-Seq is the method that uses Next Generation Sequencing to extract information from RNA [131]. The procedure follows three main phases [173]: (i) a population of RNAs is extracted from a cell, (ii) complementary DNA (*i.e.*, cDNA) is produced from RNA molecules through reverse transcription, and finally (iii) cDNA is fragmented, amplified, and sequenced. A different pipeline consists in fragmenting RNA directly and then converting it into cDNA.

Within an RNA-Seq experiment, the ability of reading contiguous stretches of RNA is limited by the technology used (*i.e.*, size selection step), hence large RNA molecules must be fragmented in order to be sequenced. For this reason, data redundancy is needed in order to reconstruct the initial RNA molecules (but also to have a high percentage of covered RNA) and amplification is a required step. Prior amplification, RNA-to-cDNA conversion is needed.

The potentialities of RNA-Seq technologies can be limited by the biases related to fragmentation and cDNA synthesis. cDNA itself can be biased due to some errors occurred during reverse transcription (*e.g.*, in *template switching*, cDNA anneals to a molecule different from the initial RNA template) which can cause chimeric alternative transcripts [28]. The problems caused by cDNA synthesis [173] include non-uniformity of coverage across transcripts and biases in transcript length. Concerning the former point, several studies pointed out that RNA-Seq technologies tend to overrepresent the $5'$ end of transcripts. In [118] Mortazavi and colleagues observed that, by performing fragmentation before reverse transcription, it is possible to reduce the biases related to non-uniform transcripts coverage. Such biases cause incorrect transcriptome assembly as well as inaccurate gene expression profiling.

Gene expression is usually measured by counting the number of reads aligned against a precise transcript, normalized by the transcript length. This metric is called RPKM (Read Per Kilobase per Million) and is defined, for each transcript, as follows:

$$\text{RPKM} = \frac{\text{number of aligned reads} \cdot 10^9}{(\text{transcript length})(\text{total number of reads})}. \tag{1.2.2}$$

An accurate RPKM computation is difficult also because read mapping against the reference genome may be ambiguous (due to the presence of repeated regions and paralogs) or requires to cope errors within the reads (sequencing errors) or within the reference genome (incorrect transcript annotation).

Direct RNA sequencing with Third Generation technologies would not need cDNA synthesis, openinig the possibility of reading a whole RNA molecule contiguously, avoiding the numerous error-affected steps described above.

# I

# Local assembly of paired reads

# 2 Sequence assembly

In genomics, the reconstruction of the original DNA sequence of an organism, or of that of a closely related species, is the first fundamental step for a number of biological analyses. Nowadays several reference genomes of different organisms are publicly available but the Assembly Problem is yet to be a closed chapter.

Some statistical approaches [88] attempted to characterize the relation between read length and genome coverage, trying to find some “conserved quantities” among genomes that should guarantee the existence of a solution, that is, a contiguous sequence representing *the* genome.

The Next Generation Sequencing revolution has allowed to sequence the DNA of large number of organisms and has enabled wide and thorough studies that helped the bioinformatic community to dramatically increase its knowledge on genome functions. In spite of this, some studies concerned with assembly tools evaluation almost invalidated all the efforts trying to formulate a “golden rule” for the optimal input for genome assembly algorithms. In fact, the repeated structure of genomes made the assembly problem nearly intractable at the time NGS reads (*i.e.*, Second Generation) were extremely short and biased, irrespective of coverage [126]; a hurdle that could be almost neglected working with long and accurate data coming from traditional capillary sequencing.

It is broadly accepted that the assembly problem remains unsolved in practice because of different data types and, especially, due to the extremely different lengths and structures of the genomes that the bioinformatics community is asked to analyze. Tools for genome assembly behave in different ways depending on the organism under study; moreover, the tools must be designed to target platform-specific sequence biases, whereas the data structures must be adapted both to the type and to the abundance of the data.

Classical approaches to the *de novo* assembly problem moved from the Shortest Superstring Problem gave birth to several heuristic algorithms for its resolution, mainly based on overlap graphs and $k$-mer graphs; greedy algorithms were proposed too, the most common ones being prefix-trees and seed-and-extend techniques.

Nowadays, advances in sequencing technologies allow to reach much higher read length; moreover, the lack of contiguity of NGS reads is now partly overcome with long-range sequence information, mainly consisting of paired reads libraries, but some efforts towards the integration of Third Generation Sequencing data.

Sequence assembly has found applications also to transcriptomic analyses. The theoretical problem is completely different and the traditional methods and tools applied to genome assembly had to be revised and integrated in order to address a different type of solution, but also to cope with the lack of coverage information.

The absence of a standard for genome assembly stimulated a field of research concerned with assembly evaluation. Starting with works by Phillippy [136] (amosvalidate) and Salzberg [143] (GAGE project), in the last few years substantial efforts have been devoted to the improvement of statistical methods for assembly projects validation, as well as to the definition of a gold standard for the evaluation of genome assembly tools.

It is fair to say that, even though we are not able to perfectly reconstruct the complete genome of an organism, both the ample choice of assembly tools and the sequencing technologies improvements allowed to gain substantial knowledge on genomes' structure. Also, the availability of public databases [66, 68, 72] of biological information matured on former projects, provides scientists with an enormous bulk of searchable data, resulting in the possiblity of finding specific information for the type of analysis to be carried out.

## 2.1 The mathematical formulation of the assembly problem

In bioinformatics, the Assembly Problem (AP) is the task of reconstructing the *genome* $g$ given a set of *reads* extracted from $g$.

Under the simplifying assumptions of identifying a read with a string and of assuming that no mismatches occur between a read and the genomic region from which it has been extracted, the Assembly Problem can be stated as follows:

**Definition 2.1.1 (Assembly Problem (AP))** *Let $\Sigma = \{A, C, G, T\}$ and $S \subseteq \Sigma^*$ be a finite set of substrings of an unknown string $g \in \Sigma^*$. Find $g$.*

Notice that AP, stated in this way, allows to easily construct several solutions satisfying Definition 2.1.1 that are poorly relevant from a biological perspective (*e.g.*, a string obtained from the concatenation of all strings in $S$, kept in any order, is a solution of AP).

Intuitively, finding a *good* solution for AP means being able to select a string $g$ the sequences of $S$ are *likely* to have been extracted from. First, some conditions on $g$ should be posed in order to limit the number of solutions; second, a *unique* solution should be selected from the set of good ones.

In literature a common choice is the introduction of an objective function in order to get an optimization problem from the feasibility problem. For each $s \in \Sigma^*$, we will denote with $|s|$ the length of $s$ and, for each $s' \in \Sigma^*$, we will say that $s'$ is a *substring* of $s$, and we will denote it with $s' \subseteq s$, if and only if $|s'| < |s|$ and there exists $i \in \{0, \ldots, |s| - |s'|\}$ such that $s[i, \ldots, i + |s'| - 1] = s'$. A new formulation for AP is given in Definition 2.1.2.

**Definition 2.1.2 (Shortest Superstring Problem (SSP))** *Let $\Sigma$ be a finite alphabet and $S \subseteq \Sigma^*$ be a finite set of strings on $\Sigma$. Find $\bar{s}$ with the following properties:*

- *$s \subseteq \bar{s}$ for all $s \in S$;*
- *$|\bar{s}|$ is minimal.*

A solution of SSP can be seen as a permutation $\sigma$ of the set $S$. Ambiguities are overcome assuming that, for every couple of consecutive strings in the permutation $\sigma$, the maximum suffix-prefix overlap is taken. It can be shown that SSP is NP-hard [106], but, in spite of this, most of the approaches to solve AP are based on this mathematical formulation.

From a practical point of view, an algorithm for solving AP needs some evidences in order to safely put strings together (*i.e.*, a minimum overlap is required). A further refinement for the formulation of AP consists in adding an input constant $k$ and requiring the suffix-prefix overlap between each couple of consecutive strings in $\sigma$ to be equal to or greater than $k$. We may call this new formulation $k$SSP and its solution, in general, is a set of strings $g_1, \dots, g_n$.

$k$SSP can be seen as the problem of finding paths on a graph. Two possible formulations involve the notions of de Brujin graph and string (or overlap) graph, in which nodes are $k$-mers or reads, respectively. See Section 2.3 for a detailed overview of algorithms based on these two models.

Even though the approach of minimizing the genome length is the most used one, the biological meaning of that choice is not clear. In fact, repeats tend to be over-collapsed by constraining the genome to be a minimum-length sequence that explains the set of reads [113]. A possible way to overcome this limitation is to regard repeats as instances of "representative sequences" and let the solution of AP to use them more than once. The number of feasible solutions can be limited by providing paired reads information.

There are also some theoretical limits on the feasibility of assembly that depend on both the dataset and the genome complexity. These two aspects will be discussed in the following section.

### 2.1.1 The parameters of the assembly: read length and coverage

The ability of reconstructing the genome of an organism, given a set of reads, depends on the amount of overlapping sequences and, thus, on the uniformity of reads sampling along the genome. A minimum length $k$ is required in order to make overlaps detectable and, possibly, unambiguous.

Given an unknown genome $g$ to be assembled from a set of $N$ reads of length $L$, we define the *coverage* as the following value

$$c = \frac{NL}{|g|}. \tag{2.1.1}$$

If a sequencing experiment extracts reads from random positions within the genome, then $c$ should be chosen in order to reach an expected contiguity level, taking into account also the minimum overlap length $k$.

In this section we recall the article by Lander and Waterman [88] who proposed a model for optimal fingerprinting of clones to get a contiguous genomic sequence starting from a physical map. The authors claim that their reasoning applies also to *de novo* genome assembly from shotgun sequencing experiments.

We will assume that every real overlap is also *detectable*, that is, if two reads $r_1$ and $r_2$ have been extracted from the genome at a distance $d(r_1, r_2) \leq L - k$, then any assembly algorithm is able to find the $d(r_1, r_2)$-length overlap among $r_1$ and $r_2$. In this hypotesis, we can model the number of gaps in the assembled sequence as the number of events of finding a new read, scanning the genome towards $5' - 3'$ direction, at a distance greater than $L - k$ from the previous one. Thus, a maximal contiguous sequence of reads overlapping each other in the genome corresponds to an assembled sequence, named *contig*, returned as output by the assembly algorithm.

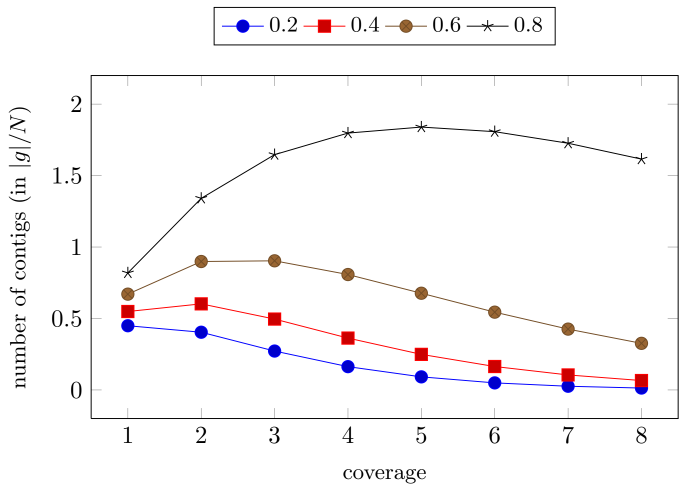


**Figure 2.1:** Number of contigs (in units of $|g|/N$) as a function of genome coverage. Curves are reported for different values of $k/L$.

If the number of bases in the genome in which a read starts is low, then the sequencing experiment can be modeled as a Poisson process in which an event corresponds to an occurrence of a read that does not overlap the previous one. If the number of reads is $N$ and the length of the genome is $|g|$, then the expected distance between the starting points of two consecutive reads within the genome is $\frac{|g|}{N}$. Hence, given a read $r_1$, the probability of finding the consecutive read $r_2$ at a distance $d = d(r_1, r_2) > L - k$, *i.e.*, $r_2$ do not overlap $r_1$, is given by

$$P(d > L - k) = e^{-\frac{N}{|g|}(L-k)}. \tag{2.1.2}$$

In other words, $P(d > L - k)$ represents the probability of finding a gap in the assembled sequence, under the hypothesis that overlaps among the reads are all detectable. By defining a boolean variable $X$ representing the event that a gap occurs ($X = 1$) or not ($X = 0$), the expected number of gaps is given by

$$E(X) = N\, P(d > L - k) = N\, e^{-\frac{N}{|g|}(L-k)}, \tag{2.1.3}$$

which can be rewritten as

$$E(X) = \frac{|g|}{L}\, c\, e^{-c(1-\frac{k}{L})} \tag{2.1.4}$$

and represents the expected number of sequences of the assembly (minus 1). In order to avoid misassemblies, a minimum overlap $k$ is required. Equation (2.1.4) tells that, by increasing $k$, the number of available overlapping reads decreases and therefore a higher value for $c$ is needed in order to reach the desired contiguity level.

Figure 2.1 depicts the expected number of gaps $E(X)$ as a function of coverage and read length, respectively. The graph suggests that results obtainable with long reads are also reachable with short ones, provided that coverage has been properly increased.

### 2.1.2 The repeated structure of the genome

Although the considerations stated above are reasonable in the context of filling gaps among sequences within physical maps or *de novo* assembly with long reads, several strong assumptions are not satisfied for most sequencing experiments, and for *de novo* assembly projects related to them, that are nowadays performed. The reasons of previous considerations are the following ones:

(i) the ratio $N/|g|$ cannot be considered small, especially for NGS experiments; if this hypothesis does not hold, then the Poisson distribution is not applicable;

(ii) the assumption that every real overlap is detectable is not always true, due to the presence of both sequencing errors and repeats, and also because the value of $k$, which is constrained to $k < L$, can lead to ambiguities especially for large and highly repetitive genomes.

The consequences of the above considerations are that, with short and biased NGS reads, a theoretical model for the optimal sequencing experiment for *de novo* genome assembly does not fit with the data. Even without taking into account biases in the data and non-uniformity of coverage, the presence of repeated sequences was shown to have a crucial role in the existence of a solution for the assembly problem when read length does not allow to span the repeats.

Nagarajan and Pop [126] proposed some theoretical results showing how the computational complexity of assemblying a genome relates with read length, coverage, and overlap length. In the following we will denote with $o(r_1, r_2)$ the length of the maximal overlapping string between $r_1$ and $r_2$.

Given a set of strings $S$ and the overlap threshold $k$, the authors model AP as a combinatorial problem on the *string graph* $SG^k(G)$, whose definition is reported below.

**Definition 2.1.3 (String graph $SG^k(S)$)** *Given a set of string $S$ and $k \in \mathbb{N}$, let $G = (V, E, w)$ be the labeled directed graph defined as:*

- $V = S$;
- *there exist a directed edge* $(r_1, r_2) \in E$ *if and only if* $r_1, r_2 \in V$ *and* $o(r_1, r_2) \geq k$;
- *each edge* $(r_1, r_2) \in E$ *is assigned a weight* $w(r_1, r_2) = |r_1| - o(r_1, r_2)$.

*The graph G is called* overlap graph. *For each path* $P : r_1 \to r_n$, *we will denote with* $w(P)$ *the sum of the weights of edges of* $P$, *plus* $|r_n|$. *The* string graph $SG^k(S)$ *is the graph defined as:*

- $V(SG^k(S)) = \{r \in V \,|\, \forall r' \in V \setminus \{r\}\ r' \nsubseteq r\}$, *i.e., reads contained within other reads are discarded;*
- $E(SG^k(S)) = \{(r, r') \in E \,|\, r, r' \in V(SG^k(S)) \wedge \forall P : r \rightsquigarrow r', P \neq (r, r') \Rightarrow w(P) > |r| + |r'| - k\}$, *i.e., transitive edges are eliminated.*

Thus, the string graph $SG^k(S)$ contains only non-redundant information, in terms of both reads and edges (*i.e.*, those supported by a path are deleted from the graph, see also Figure 2.2).

The problem of finding a solution to AP based on the string graph $SG^k(S)$ is modeled with a generalization of the Traveling Salesman Problem (TSP) to arbitrary graphs, not necessarily complete. The problem definition is as follows.

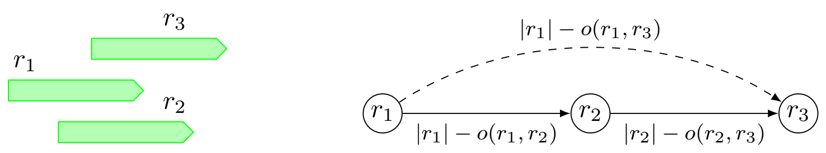


**Figure 2.2:** Identification of a transitive edge (dashed arrow). In this example, $(r_1, r_3)$ is a transitive edge because $|r_2| < o(r_1, r_2) + o(r_2, r_3)$.

**Definition 2.1.4 (Generalized TSP))** *Given a labelled graph $G = (V, E, w)$, find a path $P$ satisfing the following conditions:*

- *$P$ visits every node in $V$ at least once;*
- *$w(P)$ is minimal.*

The authors provide a polynomial reduction from the SSP, which proves the NP-hardness of the generalized TSP. They also consider the relation between read length and repeat length, and between minimum overlap and read length as well. A *repeat* is defined here as a maximal sequence of length at least $k$ that occurs multiple times throughout the genome. In this meaning, a repeat is a *sequence* repeat and not a *biological* repeat and represents a substring that can cause overlap ambiguities among the reads.

They deduce that a solution of AP, stated as a generalized TSP, cannot be found in polynomial time, unless P=NP, if either one of the following two conditions is satisfied:

- $R_{\max} < 2L - 1$;
- $L - k \geq 4$,

where $R_{\max}$ is the maximum length of a repeat. The two aforementioned results provide a negative information for the assembly problem in practice: if we do not consider neither sequencing errors nor objective functions (which reduce the solutions space), we still come up with a theoretical problem that turns out to be intractable. The above results hold for both long and short reads, but it is intriguing that the number of repeats increases as the value of $k$ decreases. In some sense, by increasing coverage, one can partially compensate for the difficulty of assemblying a genome because $k$ can be set to a higher value. Unfortunately, the need of taking into account also *approximate* overlaps, due to the presence of sequencing errors within NGS reads, introduces much more ambiguity which can be overcome only requiring longer sequence information (*e.g.*, using paired reads). Biological repeats are not identical one another, but they still cannot be easily disambiguated.

## 2.2 Input data preparation and preprocessing

The assembly of the sequencing data is often the first step for a number of biological analyses. As described in previous section, before being able to start an assembly experiment, it is necessary to decide the type of the data to be produced as well as their amount. Moreover, a preprocessing step is useful most of the time, even necessary in some cases, to prepare the data in order to get the best from a genome assembly pipeline. In the following we describe the basics of the steps to be taken when a *de novo* assembly project is to be performed.

### 2.2.1 Design of sequencing experiments

When sequencing the genome of an unknown organism, it is usually possible to estimate its length in advance (see [121]). This fact allows to decide (at least in theory) the amount of reads of the input dataset that should guarantee a given coverage $c$ of the genome. With an ideal sequencing experiment we would end up with a set of reads $R$ such that every base pair of $G$ is covered by exactly $c$ reads in $R$. In practice a weaker request is made, *i.e.*, a good sequencing experiment is such that coverage is approximately uniform, resulting in a small fraction of uncovered bases and in at most few events of highly covered ones. Unfortunately, NGS technologies are not able to produce an adequately uniform coverage [35].

Coverage uniformity, other than allowing to assemble a high percentage of the genomic sequence, is useful in detecting repeated regions. A biological repeat is a fraction of the genome that replicated itself throughout the genome during the evolution. Biological repeats can be viewed as *approximate* sequence repeats. If a read $r \in R$ comes from a repeated sequence, then we expect to find it, up to some mismatches, within another copy of the same sequence occurring in the genome. Reads coming from an occurrence of a repeat should align against other occurrences of the same repeat. As a consequence, we expect to find higher coverages in correspondence of repeats with respect to the expected one. Hence, uniform read sampling allows to better identifying repeats because a higher coverage can be interpreted as a signal related to the genome structure, rather than to the artifacts introduced in the data by the sequencing experiment.

The availability of paired read information allows to both check the correctness of the assembled sequence. Indeed, the estimated insert size can be used, for instance, to infer a draft genome assembly quality by aligning the paired reads against it and then checking the distance between the aligned mates, as well as their relative orientation. If the sequencing technology chosen produces short reads, then the PI should take into account the fact that repeats would be difficult to disambiguate. Hence, the possibility of paired read sequencing should be encouraged, if possible.

### 2.2.2 Data cleaning

Before reads are provided as input for bioinformatics tools, and hence used for biological analyses, it is important to clean the dataset. Illumina workflow comprises an initial filtering of the data outputted by the sequencer, that is intended to remove bad quality reads, and also provides summary statistics that give an overview of the experiment outcome. After sequencing, possibly with raw filtering provided by the vendor, one or more of the following steps is often performed: filtering, trimming, and/or error correction.

When raw data are given, the first step is to discard low-quality reads (quality filtering) and remove reads belonging to other genomes or organelles (contamination filtering). Due to their error profile, Illumina reads can be also trimmed, that is, a read 3' end is cut if the quality of its bases is below a threshold. These procedures can be done using, for example, a specific option of the tool ERNE [169]. Filtering (plus trimming) is an important preprocessing step for genome assembly, however, there exists also assembly tools that use quality information to improve the results (*e.g.*, Phrap [31], QSRA [18]).

Several tools are available to perform error correction on Illumina data using the so-called "read spectrum". The idea is to firstly detect low-frequency $k$-mers (within the reads) and conveniently change few bases so that a minimum frequency is reached. Some of the most recent tools are QUAKE [80], Hammer [114], and ALLPATHS [43].

An important way to assess the global quality of a dataset, instead, is to plot the reads' *k-mers distribution*. This procedure can be done before or after cleaning the dataset. The idea is that if the genome has been sequenced tens of times, then two peaks are expected in the distribution: one in correspondence of the expected coverage and one in correspondence of coverage 1. $k$-mers that cause this second peak are likely to be sequencing errors. As a rule of thumb, a low number of $k$-mers occurring only few times suggests that the dataset has a good global quality. A tool that performs $k$-mers distribution is Jellyfish [107].

## 2.3 Classical algorithmic approaches to the assembly problem

Heuristic approaches are mostly employed to tackle the difficult task of assemblying a genome. Exact methods exists as well, and they are based on branch-and-bound schemata that allow to explore the entire search space [127], but their application is restricted to the assembly of bacteria genomes and they cannot cope with the task of assemblying larger genomes. The set of *contig* sequences returned in output by an assembler is usually finished by possibly connecting the sequences to each other to form *scaffolds*. To this aim, mate pairs are commonly employed and sometimes the sequence between two consecutive contigs is also filled using *ad hoc* tools.

In literature both graph-based and greedy approaches are found, often specifically designed to handle read length and error profile of data produced by different sequencing technologies. For instance, overlap graph-based assemblers are particularly suited for relatively long reads (*e.g.*, those produced by Roche 454), having been inherited from early tools for the assembly of the data coming from traditional sequencing technologies [9]. In contrast, approaches based on de Brujin graphs found application to short sequence tags, such as the reads produced by the earliest Illumina sequencers appeared on the market (*e.g.*, Solexa). The idea of considering $k$-mers instead of reads turned out to be successful because a minimum overlap length at a fixed position, actually covering a high percentage of the read, was not required anymore, thus allowing to capture more information from the common sequences but, in contrast, hindering the task of finding a *read-coherent* solution. However, in the recent years, the improvements of sequencing technologies allowed to reach higher read length and thus extended the field of application of tools based on the overlap-layout-consensus paradigm. Greedy approaches, instead, look for the reads that better extend the current partial assembly. Seed-and-extend techniques usually keep extending a sequence, called *seed*, by selecting the best read (*i.e.*, exhibiting the highest similarity) among the overlapping ones. In this case, suffix-prefix and/or prefix-suffix overlaps are searched in the set of available reads, with respect to the current sequence. A more sophisticated approach consists in considering a set of overlapping reads. These reads are either clustered together, resulting in a hybrid method between overlap-layout-consensus and seed-and-extend, or used to validate the single-read extension. A greedy algorithm allows to save memory because overlaps are computed on-the-fly, but, for the same reason, the performances are usually slower compared to the search phase performed by assembly tools based on the overlap-layout-consensus approach.

In the following we provide a brief description of the most common algorithmic ap-

```
r1  GACCTACA
r2   ACCTACAA
r3    CCTACAAG
r4     CTACAAGT
r5      TACAAGTT
r6       ACAAGTTA
r7        CAAGTTAG
r8      TACAAGTC
r9       ACAAGTCC
r10       CAAGTCCG
```

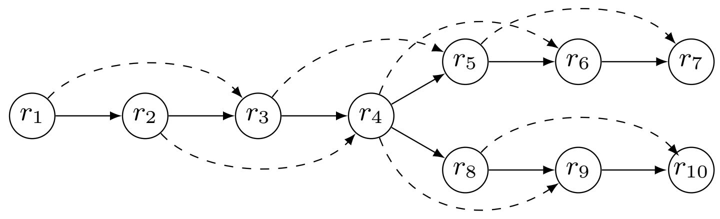


**Figure 2.3:** An example of overlap graph. The minimum overlap length is $k = 6$ and transitive edges are represented by dashed arrows.

proaches to AP mentioned above, as well as some reference tools. See also [117,138,168,180] for some overviews on the available *de novo* assemblers for NGS data.

### 2.3.1 Overlap-layout-consensus (OLC)

This technique is based either on the *overlap graph* or *string graph* (see Definition 2.1.3) built with the reads and pairwise overlaps among them. Each node refers to a read, possibly associated to a strand, and each edge represents a suffix-prefix overlap, or read inclusion (see also Figure 2.3).

The graph is built once all overlaps are computed, hence all-against-all comparison is required. The consequence is that the computational time demanded at this step grows quadratically with the total number of reads. In order to speed up the computation, reads are often indexed by $k$-mers and stored within a hash table. Overlaps are then searched among couples of reads sharing a $k$-mer in a specific position. This strategy should allow to find overlaps more quickly. It is worthwile to note that, for highly repetitive genomes, the time complexity remains high as the number of reads sharing the same $k$-mer can be large.

The resulting overlap graph, and the output of the assembly algorithm as well, depends on the value of $k$. Indeed, a large $k$ is more specific, but may prevent the detection of overlaps affected by few mismatches; in contrast, a small $k$ is more sensitive, but can produce a non-negligible amount of false positives. Moreover, the minimum overlap length required among two sequences must be adapted to both read error profile and read length. For this reason, the tools based on the OLC strategy are often specifically designed for either 454 data (*e.g.*, Newbler [110], CABOG [116]) or short reads coming from Illumina or SOLiD platforms (*e.g.*, EDENA [51]).

The OLC paradigm envisages the following three main steps: (i) the computation of all (or almost all) pairwise overlaps among the reads, (ii) the identification of chimeric reads and/or incorrect paths, and (iii) the graph simplification through path compression and transitive reduction. Steps (ii) and (iii) can be performed in reverse order and possibly re-iterated on further smaller and simpler graphs.

The overlap computation is the most demanding step in terms of memory and time consuption. For this reason, several techniques aimed at reducing the computational requirements has been proposed: CABOG [116] tries to keep the size of the graph as small as possible by keeping only the best overlaps, EDENA [51] uses only exact matches, FM-assembler [152] (and its revised implementation SGA [153]) stores overlaps using the FM-index, and PE-Assembler [6] computes overlaps considering only $k$-mers whose frequency in the reads is similar to the expected one.

Once the graph is built, the layout is analyzed in order to remove errors and eliminate redundancy. In order to remove incorrect information in the graph, either chimeric reads are discarded [110] or incorrect paths are identified [51]. Repeats are often handled using coverage information, but there exist also tools able to improve assembly quality using the location of repeated sequences within the genome (*e.g.*, MIRA [26]).

The graph can be simplified by looking at its structure. For example, maximal paths without bifurcations suggest the presence of subsequences that can be unambiguously assembled, the so-called *linear contigs* or *unitigs*. Minimus (AMOS) [157] is an example of a tool that, after the removal of redundant reads and the transitive reduction, compresses each maximal linear path into a single node. Unitigs can also feed a new overlap graph, built by computing all pairwise overlaps between them. In this case, graph simplification can be done in this last graph, possibly using other information (*e.g.*, paired reads alignment) and removing redundant edges.

Let us note that the obtained graph strongly depends on the minimum percentage of identity between overlapping sequences. The presence of sequencing errors in the reads, thus the need of detecting *approximate* overlaps, requires an extension of the classical transitive reduction algorithm for overlap graphs [121]. Moreover, if the graph is simplified by removing redundant nodes (*i.e.*, reads included within others), the coverage information employed for recognizing repeats is partially lost. Thus, heuristics for repeats disambiguation, that assume coverage uniformity along the genome, should be properly revised in order to take into account also deleted reads. To make things worse, those heuristics are intrinsically biased as NGS technologies often provide non-uniformly distributed reads [117]. At this stage, some tools can take advantage from information deriving from paired reads, if available, in order to simplify the overlap graph, for instance, retaining only paired read-coherent paths. However, few tools use paired reads yet during the construction of the graph, like PE-assembler [6].

The final graph, obtained after error correction, nodes compression, and transitive reduction, represents a solution of AP [117], that is, a minimal number of paths that explain the set of input reads.

### 2.3.2 de Brujin graph

Assembly tools based on de Brujin graphs are particularly suited for very short reads. At the time when Illumina Solexa platforms were commercialized, read length was 36bp. Providing such short sequences as input for a tool based on OLC schemata would result into highly fragmented assemblies, unless the reconstruction of short (*e.g.*, bacterial) genomes is addressed. Moreover, the higher coverages required to compensate for short read length results in a huge amount of reads, which makes all-against-all overlaps computation impracticable.

In a de Brujin graph, nodes are defined by $k$-mers and edges are $(k-1)$-length overlaps between $k$-mers. In theory, given an alphabet $\Sigma$, the de Brujin problem consists in finding a shortest superstring that contains each $k$-mer. The graph for the de Brujin Problem can be defined in two different ways: the first one consists in placing a node for each $k$-mer and an edge whenever a $(k-1)$-length overlap between two $k$-mers occurs; the second one states that $k$-mers correspond to edges and $(k-1)$-mers correspond to nodes (connections

```
r1 GACCTACG
r2  ACCTACGG
r3   CCTACGGT
r4    CTACGGTA
r5   CCTTCGGT
r6    CTTCGGTA
r7   CCTTCGGC
r8    CTTCGGCC
```

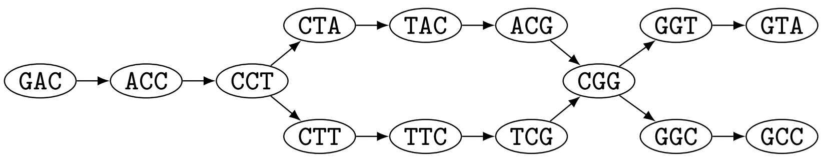


**Figure 2.4:** An example of de Brujin graph with $k = 3$. Only read-coherent edges are reported.

between edges identify $(k-2)$-length overlaps and a $k$-mer is the concatenation of nodes connected to each other). In the first case, the solution to the de Brujin Problem is a Hamiltonian path, whereas in the second formulation it corresponds to an Eulerian path.

When using de Brujin graphs for solving the Assembly Problem in practice, given the value of $k$, the existence of all $k$-mers in the set of reads is not guaranteed. In this case, the graph is defined by picking only occurring $k$-mers and related overlaps (see also Figure 2.4).

A definition of AP that both uses combinatorial properties of graphs and takes into account practical issues, can be found in [126]. The idea is that each read must be used at least once and the solution of the assembly problem must not "break" paths corresponding to reads. This problem is called de Bruijn Superpath Problem and is NP-hard, but, in some cases, a solution can be found in polynomial time (depending on the relation between the size of a maximal repeat and read length).

Several problems arise when using this method: as for OLC approach, DNA double-stranded nature complicates the graph definition and requires either to use two nodes for each sequence and then reconcile the assembly on a unique strand, or to add constraints on the edges that can be used to enter into a node and exit from it. The presence of repeats generates cycles in the graph, hence the information provided by the graph should be integrated by reads sequences or paired reads in order to disambiguate the "right" path. Finally, sequencing errors may introduce $k$-mers or overlaps that do not exist in the genome, hence the graph should be pruned in correspondence of unsafe paths (*e.g.*, *spurs* and *bubbles*) [117].

Various approaches based on de Brujin graphs are found in literature. The first proposed assembler in this category was Euler [134], designed for Sanger reads and then adapted for short NGS sequences (Euler-SR [22]). It first preprocesses the dataset using the so-called *spectral alignment* (see Section 2.2.2) in order to discard or correct the reads. Once the graph is built, several operations are performed. First, short repeats are resolved using spanning reads: in practice, a repeats collapse can be found in the graph as a sub-path common to several paths,a situation that can be resolved by splitting the repeat into the paths it belongs to. Second, a similar approach is used for long repeats spanned by paired reads inserts. Sequencing errors are treated using information on the error profile, that is, the part of the read more subsceptible of being incorrect (the 3' end for most platforms). In Euler, this collection of heuristics is implemented for two graphs built using two values of $k$. The final assembly is obtained by taking paths with edges representing both sensitive (small $k$) and specific (large $k$) overlaps.

Velvet [179] is a widely used assembler for short reads; differently from Euler, it does not perform read filtering, instead it iteratively eliminates spurs from the graph, considerably reducing its size. This reduction is done without any spectral alignment, but using a weight for each $k$-mer, corresponding to the number of reads in which it occurs. Low frequency $k$-mers are not added to the graph. The graph is further simplified by reducing each linear path into a single node that corresponds to a unitig. Then, bubbles (*i.e.*, branches that converge after few nodes) are resolved by taking the most supported path, the other one is deleted and the reads supporting it are re-aligned. Unsafe connections are finally removed by looking at low supported paths. Long reads information, if available, is used to connect nodes in the graph. In order to perform scaffolding, mate pairs are used to identify couples of long (unique) contigs. These pairs are filled with short (repeated) contigs in order to fill the gap among the long (unique) ones.

ABySS [154] is an assembler particularly suitable for working with short reads and large amounts of data, as it allows a distributed computation. Each node of the graph represents both a $k$-mer and its reverse complement. Similarly to Euler and Velvet, ABySS simplifies the graph by removing low covered paths and resolving bubbles. Contigs are built using only highly supported paths. Mate pairs information is used only in a second stage and serves both to disambiguate repeats and to merge contigs.

ALLPATHS [20] uses the read spectrum to perform error correction: "trusted" $k$-mers are those exhibiting high frequency and high base quality (provided by the reads). These $k$-mers are employed for error correction. First, perfect overlaps are computed in order to get unitigs. Then the graph is built and spurs are eliminated. Once the "unipaths" (*i.e.*, paths corresponding to unitigs) are corrected, they are used as seeds to build the assembly. The algorithm works on subgraphs and finds overlaps independently on each of them. The result is that ambiguities are preserved. After assembly, small disconnected components, as well as paths not supported by paired reads, are eliminated.

SOAPdenovo [96] is an assembler that firstly corrects reads using $k$-mer frequencies. The implemented algorithm is similar to Euler and Velvet's, but it requires less memory. First, SOAPdenovo builds a de Brujin graph, then it removes small incorrect paths, and resolves bubbles in a Velvet-like fashion. It also uses paired read information to get trusted contigs, mapping all reads against the assembly (even those not used for building the contigs), and removing transitively reducible edges.

Ray [13] is a parallel assembler that takes as input a mixture of read libraries (both short and long reads). Meraculous [23] is designed for paired reads libraries; differently from other assemblers based on de Brujin graphs, it does not perform explicit reads correction, but, instead, it uses information provided by paired reads.

### 2.3.3 Greedy

The greedy methods employed by assembly algorithms sensibly changed when turning from the assembly of bacterial genomes with Sanger reads to the assembly of large eucharyotic genomes with NGS reads. The *greedy* term refers to the fact that decisions are taken locally and no information on the global contig being constructed is used. The first greedy assembler found in literature is probably TIGR Assembler [160], which was proposed by Sutton and colleagues yet in 1995. A similar approach was the one used in Phusion [120]. In brief, the method consists in computing exact sequence matches between all reads pairs, overlaps are then assessed in a Smith-Waterman fashion [156], and possibly the sequences are joined. The difference with the OLC paradigm lies on the fact that overlaps are considered starting from the highest scoring one, where the score of a sequence depends

| | |
|---|---|
| $k_1$ | AAT |
| $k_2$ | AAG |
| $k_3$ | TAT |
| $k_4$ | TCA |
| $k_5$ | TCG |
| $k_6$ | CTA |

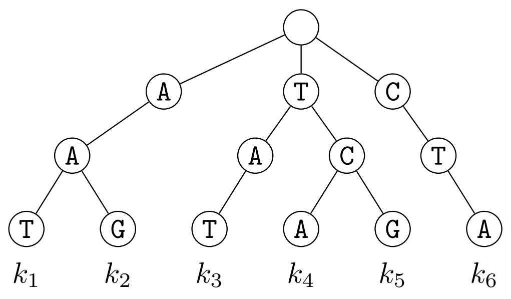


**Figure 2.5:** An example of prefix tree of 3-mers.

on the number of occurrences in the dataset. This procedure is repeated iteratively until no more merging is possible. Such a method cannot disambiguate genome repeats, thus it can easily generate misassemblies.

A common technique adopted by greedy assemblers for NGS data is the so-called *seed-and-extend*. The basic implementation of a seed-and-extend assembler consists in repeatedly picking up a seed (it can be either a read or a previously assembled contig) and extending it using the best overlapping read. The standard method is that implemented in SSAKE [174]. A read is attached to the current contig only if the overlap score is above a threshold and then it is removed from the set of available reads. The procedure stops when either no more overlaps are found or ambiguities (*i.e.*, multiple extensions) occur. Hence, this method requires the computation of all, or almost all, the overlaps between the seed's tips and all the available reads. The procedure described above has the advantage of being relatively quick, as it does not necessarily require all-against-all comparisons, but it is still unable to detect repeats and thus it should be refined in order to perform "safer" extensions.

The basic seed-and-extend technique can be modified by considering more than a read at each extension step. The approach followed by Jeck and colleagues [77] in VCAKE envisages the computation of a consensus string through a "base calling" step in which the highest represented base is selected, in the set of overlapping reads, at each position. Threshold parameters should be set in order to avoid adding a new base if the probability of being incorrect is high. A different approach that uses information from multiple overlapping reads is that implemented in SHARCGS [36]. Every time a read $r$ perfectly matching the current contig $C$ is detected, it undergoes a validation phase. The set of read is searched in order to extract all reads matching the overlap between $C$ and $r$. If all reads found in this phase also match both $C$ and $r$, then $C$ is extended with $r$.

Phrap [31] implements a greedy algorithm that massively uses quality values (produced by Phred, the so-called *Phred scores*). It performs assembly by picking the highest quality sequence portions, instead of computing a consensus sequence. Another method that employs quality values to improve the assembly is QSRA [18].

A different approach is that implemented in SHORTY [52]. The tool is designed to handle SOLiD data and the key factor of the algorithm is the usage of paired read information. The idea is to identify overlapping reads by looking at matching $k$-mers. In particular, the authors define a search procedure that make use of the *left-mers* and *right-mers* (*i.e.*, $k$-mers belonging to the $5'$ and the $3'$ mates, respectively) belonging to candidate reads and that overlap the seed read and its mate, respectively.

The choice of suitable data structures to store the reads and, especially, to efficiently search for overlaps, is an important task concerning the design of seed-and-extend algorithms. Indeed, differently from OLC methods, no information on pairwise reads overlaps is stored and, thus, they must be re-computed every time.

Reads are commonly indexed and stored within a hash table. Suffix-prefix overlaps require efficient search of putative overlapping reads by looking at their prefix. For this reason, sometimes a good choice can be that of storing fixed-length read prefixes into *prefix trees*, or *tries*. The currently adopted definition of prefix tree is that given by Knuth in 1973 [82]. A prefix tree is a labelled tree in which the label of the root is empty whereas the label of every other node is a letter (see also Figure 2.5). The labelling is done in such a way that each distinct root-leaf path identifies a prefix, and vice-versa. Several greedy assemblers use prefix-trees, *e.g.*, SSAKE [174], VCAKE [77], SHARCGS [36], SHORTY [52], and QSRA [18].

## 2.4 Judging the results: methods for sequence assembly evaluation

Many challenges are related to the comprehension of the genome of an organism: finding structural variants, identifying genes, and characterizing the transcriptome. These tasks should be addressed in order to answer important biological questions related to evolution and disease understanding. For these reasons, being able to both obtain a reliable reference genome and assess its correctness are mandatory skills for the bioinformatics community. The availability of assembly validation methods would also constitute a starting point for the definition of independent and standard metrics to evaluate assembly tools.

Genome assembly evaluation is a quite new research subject that criticizes the achievements of the last 10 years, with the aim of reconstructing a genome in a more careful way, now that more and more genomes are being sequenced every day. The main research lines focus on the production of benchmark datasets and on the standardization of assembly evaluation statistics.

Although several tools for genome assembly are available and several projects based on such tools have been carried out, a very lively discussion on how to validate new assemblies and, in general, on how to estimate assemblers' output recently started. All assembly tools are based on a small number of algorithms and differ from one another only in matter of details that, very often, relate to how they deal with errors, inconsistencies, and ambiguities. As a consequence, an increasing number of studies is now being published aiming, on the one hand, at evaluating *de novo* assemblers and assemblies and, on the other hand, at criticising the results achieved so far.

Assemblathon first [37] and second [15] editions, dnGASP [56], and GAGE [143] try to assess the performances of existing tools triggering an assembly evaluation competition among several bioinformatics groups. Even though these competitions succeeded in giving a fairly complete overview of assemblers potentialities, they are almost always based on specific (often already sequenced) genomes or on simulated data, leaving open the question of whether the same tools would have had the same performances when run on different datasets (*i.e.*, different genomes or real reads). In fact, evaluation on simulated datasets is inherently biased by the capabilities of the read simulator to faithfully reproduce error schemata [170].

Recently proposed assemblies carried out using NGS data only (usually Illumina reads), are at the center of a lively debate. Alkan in [2] criticised two of the major late NGS achievements: the assemblies of the Han Chinese and Yoruban individuals [96], both sequenced with Illumina reads. For example, Alkan identified 420.2 Mbp of missing repeated sequences from the Yoruban assembly and estimated that in both assemblies almost 16% of the genome was missing.

More than four years after the so-called NGS revolution started, it is extremely clear that *de novo* assembly needs extensive and standardized validation steps. NGS breakthrough allowed to sequence a number of new species and individuals thought to be impossible only few years ago. While, on the one hand, an increasing number of people keeps sequencing and assemblying genomes using available assemblers and short reads, on the other one, day after day, a larger community criticises and casts doubts on assembly achievements.

### 2.4.1 Statistics for genome assembly evaluation

The resulting collection of sequences representing the assembly of an organism should be evaluated with respect to three parameters: (i) *contiguity*, *i.e.*, the result of the ability of putting sequences together, (ii) *completeness*, *i.e.*, the fraction of genome that the tool successfully reconstructed, and (iii) *correctness*, *i.e.*, the degree of accuracy of the assembled sequences with respect to the real genome. It is clear that statistics related to different evaluation parameters can be conflicting. For instance, high assembly contiguity can be reached without concerning about the accuracy of the assembled sequences, but correctness is likely to collapse if such an approach is adopted.

A widely used, often inflated, measure to assess assembly contiguity is N50. Given a set of contigs $c_1, \ldots, c_n$ sorted by non-increasing length, N50 is the minimum $k$ such that $\sum_{i=1}^{k} |c_i|$ is at least 50% of the cumulative contigs length. NG50 is defined similarly, with the difference that the percentage is computed with respect to the estimated genome length.

Other widely used statistics are the total assembly length, the number of contigs, and the mean contig length. These measures are concerned with contiguity and completeness only, but, being easy to compute, they are often used alone to hurriedly decide which result is the best among the output of different assembly tools, telling nothing about contigs correctness. Recently, some studies started to criticise the way in which the evaluation of assemblies and assemblers is carried out [128, 170].

Correctness estimation, if a reference genome is absent, usually requires the reads to be mapped against the assembly. Assembly quality can be retrieved by computing reads coverage (see (2.1.1)) and checking if it is similar to the expected one. Suspicious regions are those exhibiting aberrant coverage profiles: lower coverage may indicate a misassembly or repeat expansion, whereas higher coverage is a signal of repeat collapse. If paired reads libraries are provided, further additional information can be collected. For instance, a region with a high number of correctly aligned reads with their mates being unmapped, is an indication that a misassembly may have occurred; paired reads mapped at unexpected distances (*i.e.*, different from the expected insert size) or with wrong orientation witness instead putative sequence expansions/compressions or inversions, respectively. If the reference genome is available, more accurate statistics can be computed, allowing to pointwise check for the presence of mismatches, insertions, deletions, repeats expansions and compressions [87].

Even though several features can be computed, deciding which is the best one among various assemblies is a non-trivial question to answer. This is mainly due to the difficulty of establishing a trade-off between completeness, contiguity, and correctness, but also to the fact that some measures can be related one another, resulting in a biased judgment of the ability (or incapability) of a tool in solving some problematic situations. Recently some research efforts have been devoted to the design of methods aimed at either (i) standardizing genome assembly validation, providing tools able to automatize the computation of the features commonly used to detect misassemblies (*i.e.*, amosvalidate [136]), and (ii) simplifying the assembly evaluation, by reducing the statistics to those actually retaining all information [170]. It is important to note that both methods are reference-independent, thus both of them can be employed to evaluate assemblies of newly sequenced organisms. The attempt to minimize the number of significant features is useful in the context of assembly tools comparison, which effectiveness was tested on datasets already evaluated on NGS assembly competitions [171].

# 3 GapFiller: a local assembly tool for paired reads libraries

The recent Next Generation Sequencing (NGS) breakthrough and the consequent tremendous increase in data production, have been accompanied by the appearance of a multitude of pipelines able to *assemble* the (relatively) short sequences (*i.e.*, reads) produced by state-of-the-art sequencers.

In the last two years more than 20 new *assemblers* (see [117] for an up-to-date overview) have been proposed, more than doubling in size the population of the assemblers designed for *long* Sanger reads. Despite the practical and theoretical problems involved in assembling complex genomes using only short sequences [126], several *de novo* assembly projects based exclusively on NGS data have started. Among the most popular ones we mention the Panda genome project [94], the assembly of specific human Individuals [96] (Han Chinese and Yoruban), and several other species [54].

In this chapter we go back to basics and introduce GapFiller, a tool for local assembly of paired reads. Our main goal is to obtain a set of correct in-silico built sequences that can hence be used for further analyses. In the following we present the general idea of the algorithm, a detailed description of the implemented heuristics, and the data structures employed to speed up the assembly phase.

## 3.1 Enhance assembly quality using insert information

The lack of contiguity proper of short NGS reads can be partially accomplished by insert size information, when paired reads libraries are available (see also Section 1.2.2). Paired reads are employed both to disambiguate complex structures during assembly (*e.g.*, Euler [134], CABOG [116], and Velvet [179]) and to connect the assembled sequences during the finishing (*i.e.*, *scaffolding*) phase [76]. Using paired reads directly provides information on the estimated distance between the two mates, but both the exact insert size and the complete (*i.e.*, ungapped) insert sequence are unknown. Due to the weakness of this type of information a single paired read cannot be used alone to assess the correctness of the assembled sequence, nor to connect two separate parts of an assembly. In other words, it is often necessary to provide a threshold amount of evidences, in terms of paired reads, to infer that two sequences are adjacent one another in the genome.

In literature, a handfold of recently proposed techniques successfully employ inserts assembly to either decrease assembly problem complexity [183] or increase assembly contiguity and correctness [102].

The assembler MaSuRCA [183] is aimed at reconstructing a genome starting from a mixture of data (Illumina, 454, and Sanger reads) resulting in a highly heterogeneous input. The assembly of short Illumina reads, together with pair information, allows to get slightly longer sequences, that can hence be used to feed a de Brujin graph-based tool, together with original reads produced by other technologies. The dataset complexity reduction allowed to cope with reads coming from the sequencing of the loblolly pine [70], whose genome length is estimated to be $\sim$ 22 Gbp.

The pipeline ARF-PE [102] is based on external assembly and alignment tools (*i.e.*, Velvet [179] and SOAP2 [95]) and provides a set of insert starting from a paired reads library. Reads are first assembled and inserts are identified by aligning paired reads against the contigs. Then, the obtained inserts are used as input for an assembly software.

### 3.1.1 Close the gap between paired reads with a seed-and-extend technique

Our proposed tool, GapFiller [123], is a local *de novo* assembler. It takes paired reads as input and returns a set of contigs, corresponding to the assembled inserts.

Closing the gap within paired reads is a strategy already used by software packages like SHERA [142] and FLASH [104]. However, these tools are able to work with “overlapping libraries” only, that is, libraries whose insert size is shorter than twice the read length.

The innovative feature of GapFiller is the possibility to produce a highly reliable output that is certified correct. Indeed, we label as *trusted* only those contigs for which there are evidences that the gap between the two mates has been filled and we will demonstrate that false positives are extremely rare. Moreover, we will show that the sequences produced generate a genome coverage consisting of evenly distributed long contigs. Hence, contigs assembled with GapFiller find application in numerous downstream analyses, *e.g.*, improve a whole genome assembly and reconstruct structural variants (see Section 3.5). In Part II we will further discuss the applicability of our method, with particular emphasys on the employment of contigs layout information.

Our method is based on a *seed-and-extend* schema (see also Section 2.3.3) aimed at closing the gap between the two mates of a paired read. Similarly to other seed-and-extend based tools like SSAKE [174], SHARCGS [36], QSRA [18], and TAIPAN [149], GapFiller selects one read and tries to extend it using reads that overlap for a significant region. The main drawback of seed-and-extend assemblers is their inherent incapability to cope with complex (*i.e.*, repetitive) genomes. We employ a refined extension phase that (i) considers more than a read at each extension step, and (ii) identifies problematic positions in the consensus sequence and treat them in a different way in order not to introduce errors. The assembler TAIPAN [149] is implemented to stop its extension phase in presence of a repeat, but, like all other full-fledged assemblers, it is not designed to output certified contigs.

The seed-and-extend technique employed in GapFiller is realized by computing and analysing all—or almost all—the overlaps between the seed tips and the available reads. The reads used for an extension are those with the highest alignment score. The computation bottleneck of seed-and-extend assemblers lies in their capability to quickly cope with all the alignment scores to be determined.

GapFiller begins by storing all *useful* reads in a memory efficient data structure that allows to readily compute overlaps between the contig under construction and the remaining available reads. In a second phase each seed read (possibly belonging to a new set of paired reads) is selected one after the other and used to start an extension phase. Such phase halts when a stop condition is reached. Depending on the stop condition, the contig produced is labelled as *trusted* or *non-trusted*.

## 3.2 The algorithm

In this section we will describe the hashing technique employed for overlaps search, the steps performed during the extension phase, and the stop criteria that make the extension halt. High output accuracy is reached thanks to both the refined extension phase, which computes the consensus in two stages, and the stop criteria, which assigns the *trusted* attribute only to contigs that are likely to be correct inserts.

### 3.2.1 The hashing technique

Let $\Sigma$ be an alphabet and $\Sigma^*$ be the set of the words from $\Sigma$. For every $S \in \Sigma^*$ we will denote with $|S|$ the number of characters of $S$ 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 $l = 1$, and we will simply write $S[p]$.

In order to quickly identify overlaps between the contig under construction and the reads' tips, we use an approach closely related to the one presented in [169] based on an Hamming-aware hash function. The idea is that, by representing a string of length $l$ as a base-$|\Sigma|$ number, expensive char-by-char comparison can be often replaced by fast integer (or bit-string) comparison. However, for practical values of $l$, the integers to be compared would not fit in a memory word. For this reason, as in the classical Karp-Rabin exact string matching algorithm [79], we can work with numbers modulo $q$ considering equality modulo $q$ only as an indication (*i.e.*, a necessary condition) that pairs of strings may be the same. In other words, we operate with the strings *fingerprints*. Policriti *et al.* proposed in [137] an extension of the approach by Karp and Rabin, introducing a technique to deal with mismatches, based on the idea of replacing simple fingerprints comparison with a more articulated test. In particular they noticed that, by choosing $q$ to be a Mersenne (prime, when possible) number (*i.e.*, $q = 2^w - 1$, for some $w \in \mathbb{N}$), to check whether two strings align against each other at a small Hamming distance can be implemented in average linear time.

Given a string $S \in \Sigma^*$ and its base-$|\Sigma|$ numerical representation $s \in \mathbb{N}$, let us define the hash function $f_H \colon \Sigma^* \to \{0, \ldots, q-1\}$ as

$$S \quad \mapsto \quad f_h(S) := s \mod q, \tag{3.2.1}$$

where $q$ is a number 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 our context, the use of $f_H$ significantly reduces the size of the set employed in the search of the overlapping reads. Every read $r$, as well as its reverse-complement, is indexed by the fingerprint of a substring of length $b$, starting at a fixed position $x$ in $r$ (see also Figure 3.1).

Formally, given a set of reads $\mathcal{R}$, a sequence $S$, a maximum allowed Hamming distance $k$, the set $\mathcal{Z}(k, q)$ of the *witnesses* (*i.e.*, the Hamming sphere of radius $k$ around $S$, see [137]

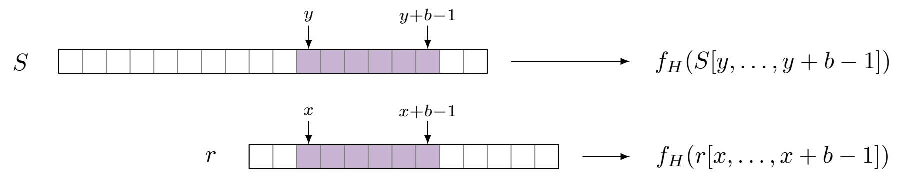


**Figure 3.1:** Fingerprint computation on $b$-length substrings. When looking for overlaps between $S$ and $r$, the fingerprints are computed on the substrings $r[x,\ldots,x+b-1]$ and $S[y,\ldots,y+b-1]$, respectively, where $x$ and $b$ are set before the contig's extension phase. We require an (almost) exact $b$-length match between $r$ and $S$ in order to include $r$ in the set of putative overlapping reads, by setting $f_H(r[x,\ldots,x+b-1]) = f_H(S[y,\ldots,y+b-1])$. Using such a method, the suffix-prefix overlaps that can be detected are those of length $l \geq x+b$.

for more details), a fixed value $b$ for the length of the substring on which the fingerprint is computed in $r$, and two positions $x$ and $y$, the following set

$$\mathcal{R}(S,x,y) := \{r \in \mathcal{R} \mid (f_H(r[x,\ldots,x+b-1]) - f_H(S[y,\ldots,y+b-1])) \mod q \in \mathcal{Z}(k,q)\} \tag{3.2.2}$$

contains at least all the reads $r \in \mathcal{R}$ such that the hamming distance between $r[x,\ldots,x+b-1]$ and $S[y,\ldots,y+b-1]$ is not greater than $k$. False positives can be present but, as showed in [137], their amount is limited. On this ground the search for reads overlapping $S$ can be restricted to those belonging to $\mathcal{R}(S,x,y)$, for some $x,y \in \mathbb{Z}$.

As far as GapFiller is concerned, we set $k=0$ as default, meaning that we search for *exact* $b$-length substrings in the reads (*i.e.*, $r[x,\ldots,x+b-1] = S[y,\ldots,y+b-1]$, for some $x$ and $y$). As a consequence, better quality output will be obtained if we select a position $x$ in $r$ such that the average base quality is expected to be the highest possible. This point will be further discussed in Section 3.3.1.

### 3.2.2 The assembly phase: contig extension

In the contig extension phase, each read is selected in a loop and 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 string, which is subsequently used to perform further extensions. The procedure continues while enough overlapping reads exist and the consensus string is *highly representative* of the clustered reads. If either one of the previous two conditions is not met, the extension phase stops, the current sequence is returned in output, and the loop continues.

Before the extension phase some parameters are set: the minimum overlap length $L$ and the maximum shift $\Delta$: an overlap between the current contig suffix and the read prefix is considered only if the overlap length $l$ belongs to the interval $[L, L+\Delta]$.

GapFiller builds a *cluster* every time a contig is to be extended with the overlapping reads. In particular, GapFiller uses only those reads aligning against the contig suffix with at most $\delta$ mismatches (where $\delta := \delta(l)$ is a function of the overlap length $l$) and requires at least $m$ reads in order to compute a consensus string. Notice that $b \leq L \leq l$ holds, hence suffix-prefix overlaps might occur with more than $k=0$ mismatches (see Section 3.2.1).

Let $\mathcal{R}$ be the set of input reads for GapFiller and $r_0 \in \mathcal{R}$ be a seed read. At step $i=0$ the current sequence is initialized with the seed $S_0 := r_0$. Denoting by $S_i$ the current

**Step 1.** The overlap length is at least $L$ and at most $L + \Delta$ for every read $r_i$, $i = 1, \ldots, 4$. The number of mismatches (highlighted in red) with $S_i$'s suffix does not exceed $\delta = 2$.

**Step 2.** The consensus string is computed for every position covered by at least $m = 2$ reads. The characters rounded in gray and red lay on low-represented and non-represented positions, respectively. In presence of ambiguities (*i.e.* characters with the same representation rate) we choose the one belonging to the first read encountered, from left to right.

**Step 3.** Reads with mismatches in correspondence of low-represented positions are discarded, hence they do not contribute to reach the threshold $m$ to compute a new consensus string. In our example read $r_4$'s tail is cut in the non-represented position, regardless on whether it matches the consensus string or not.

**Step 4.** The reads still alive after Step 3 are used to compute the final consensus string $C_{new}$. Since there are $2 \geq m$ reads exceeding $S_i$'s right end, $C_{new}$ is computed, is attached to $S_i$, and the extended contig $S_{i+1}$ is obtained.

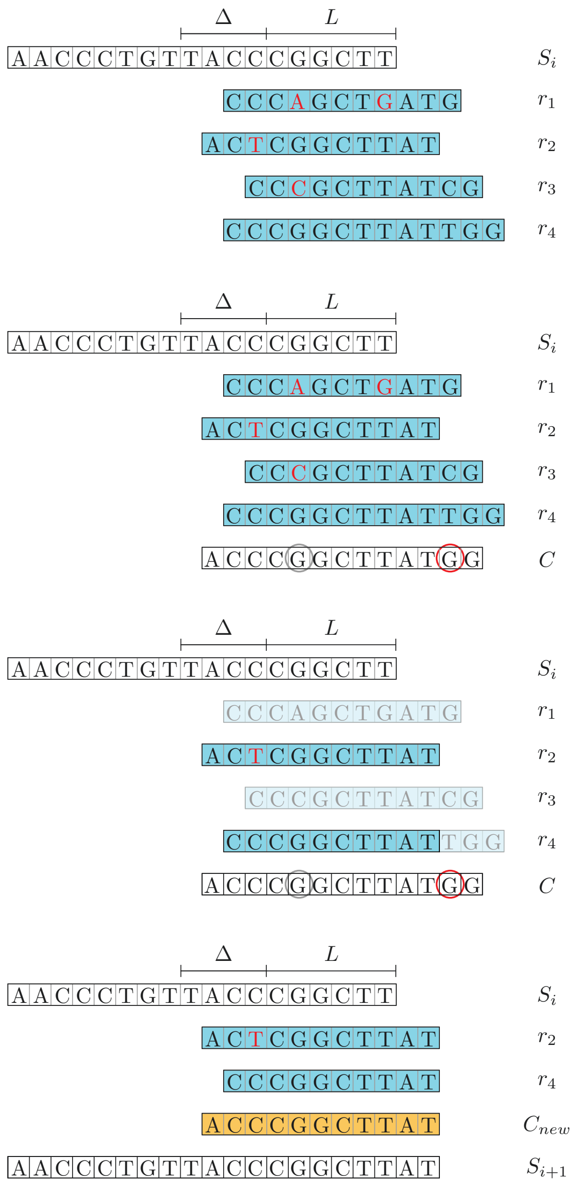


**Figure 3.2:** GapFiller extension phase. An example with $\delta = 2$, $m = 2$, $T_1 = 0.3$ and $T_2 = 0.5$.

contig at the generic $i$-th step of the algorithm, the procedure to build $S_{i+1}$ is described in the following.

## Step 1. Overlapping reads selection

Reads are selected according to their similarity with the current contig $S_i$ (see Figure 3.2, Step 1). At this point, every read overlapping $S_i$ for $l \in [L, L+\Delta]$ characters with at most $\delta$ mismatches is selected.

Let us denote with $\mathcal{R}(S_i, l)$ the set of the putative overlapping reads with respect to the $l$-suffix of $S_i$, selected by their fingerprint values (see (3.2.2), with $y = |S| - l + x$, for some values of $x \in \{0, \dots, l - b\}$). For every fixed value of $l$, the set of the reads overlapping the $l$-suffix of $S_i$ with at most $\delta$ mismatches is defined as

$$\hat{\mathcal{R}}(S_i, l) := \{r \in \mathcal{R}(S_i, l) \colon d_H(r[0, \dots, l-1], S_i[|S_i| - l, \dots, |S_i| - 1]) \leq \delta\} \tag{3.2.3}$$

where $d_H \colon \Sigma^l \times \Sigma^l \to \mathbb{R}^+$ is the Hamming distance. The set of all the overlapping reads will be denoted by

$$\hat{\mathcal{R}}(S_i) := \bigcup_{l=L}^{L+\Delta} \hat{\mathcal{R}}(S_i, l). \tag{3.2.4}$$

Given a read $r \in \hat{\mathcal{R}}(S_i, l)$, we define its starting and ending positions as

$$I(r) := |S_i| - l \qquad F(r) := I(r) + |r| - 1. \tag{3.2.5}$$

$I(r)$ and $F(r)$ represent the position of the read $r$ with respect to the current contig $S_i$, therefore we set $I(S_i) = 0$. For instance, in the case depicted in Figure 3.2, we have $\hat{\mathcal{R}}(S_i, 8) = \{r_1, r_4\}$ and $\hat{\mathcal{R}}(S_i) = \{r_1, r_2, r_3, r_4\}$, $I(r_1) = 10$ and $F(r_1) = 20$.

**Step 2. Reads clustering and consensus string computation**

This phase consists of the computation of the consensus string obtained from the set of reads $\hat{\mathcal{R}}(S_i)$ (see (3.2.4)). Notice that, in order to compute reliable extensions, we require the number of reads to be at least $m$, a parameter that may depend on the dataset used. 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 procedure stops. Otherwise, the starting and ending positions of the consensus string $C$ with respect to $S_i$ can be computed, thanks to (3.2.5). In practice, we let the consensus string start from the leftmost reads, *i.e.*, those covering the longest suffix of $S_i$ (see, for instance, the read $r_2$ in Figure 3.2) and end at the rightmost position in which the number of reads is at least $m$. More precisely, the starting and ending positions of $C$ are defined as

$$\begin{aligned} I(C) &:= \min\{I(r) \colon r \in \hat{\mathcal{R}}(S_i)\}; \\ F(C) &:= \max\{F(r) \colon r \in \hat{\mathcal{R}}(S_i) \wedge |\{r' \in \hat{\mathcal{R}}(S_i) \colon F(r') \geq F(r)\}| \geq m\}, \end{aligned}$$

respectively. If $F(C) > |S_i| - 1$ then the procedure continues, otherwise it stops as $S_i$ cannot be further extended. Looking at Figure 3.1 we have $I(C) = 9$ and $F(C) = 21$ and the procedure continues since $F(C) > F(S_{i+1}) = 17$.

More precisely, for every position $j$, GapFiller selects the most occurring character in the reads $\hat{\mathcal{R}}(S_i)$ covering $j$, and the majority consensus string $C$ is computed (see Figure 3.2, Step 2). For every $X \in \Sigma$ and for every $j = I(C), \dots, F(C)$ we define the number of occurrences of the character $X$ in position $j$ with respect to $S_i$ as

$$\sigma(X, j) := |\{r \in \hat{\mathcal{R}}(S_i) \colon I(r) \leq j \leq F(r) \wedge r[j - I(r)] = X\}|. \tag{3.2.6}$$

The consensus string $C$ is defined, for every $j = I(C), \dots, F(C)$, by setting $C[j - I(C)]$ equal to the highest occurring character, *i.e.*, the $X \in \Sigma$ with the highest number of occurrences in position $j$

$$C[j - I(C)] := \arg\max_{X \in \Sigma} \sigma(X, j).$$

Loosely speaking, the character selected on a particular position of the consensus string is the most occurring character in the reads on that position. Thus $\sigma(C[j - I(C)], j)$ is the number of occurrences of character $C[j - I(C)]$ on position $j$.

### Step 3. Consensus-based reads selection

At this point we need to check whether (i) a read $r$ is highly representative of the consensus $C$ and (ii) the extension is "safe". Every character of the consensus string is assigned a flag indicating how it is representative of the reads from which it is built. We simply define the *representation rate* of the position $j$ as

$$\pi(j) := \frac{\sigma(C[j - I(C)], j)}{|\{r \in \hat{\mathcal{R}}(S_i) : I(r) \leq j \leq F(r)\}|}, \tag{3.2.7}$$

where $\sigma(\cdot)$ is defined in (3.2.6). We set two threshold values $T_1$ and $T_2$ such that $0.25 \leq T_1 < T_2 < 1$ (notice that $\pi(j) \in [0.25, 1]$ as $|\Sigma| = 4$) and we distinguish three types of positions in the consensus string:

$$\begin{aligned} j \text{ is } \textit{non-represented} &\Leftrightarrow \pi(j) \leq T_1 \\ j \text{ is } \textit{low-represented} &\Leftrightarrow T_1 < \pi(j) \leq T_2 \\ j \text{ is } \textit{high-represented} &\Leftrightarrow \pi(j) > T_2. \end{aligned}$$

The reads used to build the consensus $C$ are filtered and trimmed, depending on the presence of low-represented and non-represented positions, respectively. The idea is that on low-represented positions we need a minimum percentage of reads matching the consensus string, and that on non-represented positions the extension in considered to be unsafe, thus it should be halted. Reads differing from $C$ in correspondence of low-represented positions are discarded and the remaining ones are also trimmed if a non-represented position occurs (see Figure 3.2, Step 3).

In practice, we do not consider a read $r$ if it does not match the consensus string on a low-represented position, *i.e.*, $r[j - I(r)] \neq C[j - I(C)]$, for some $j$ such that $\pi(j) \leq T_2$. Clearly, this applies to non-represented positions as well. Then, we trim every read overlapping any non-represented position of $C$. More precisely, if $j_{not}$ is the first non-represented position occurring in $r$ (*i.e.* $\pi(j_{not}) \leq T_1$), we consider $r[0, \ldots, j_{not} - I(r) - 1]$ instead of $r$.

After unsafe reads are discarded and the remaining ones are trimmed, a new set of reads, that can be denoted by $\hat{\mathcal{R}}_{new}(S_i)$, is finally obtained (see Figure 3.2, Step 3). Every read in $\hat{\mathcal{R}}_{new}(S_i)$ is both matching the consensus string $C$ on each low-represented position and not covering any non-represented one. Using this mechanism we take into account only the most representative reads and do not extend the contig with a consensus character when its representation rate is too low.

### Step 4. Final consensus string computation and contig update

After previous step, the new set of overlapping reads $\hat{\mathcal{R}}_{new}(S_i)$ is obtained. A new consensus string $C_{new}$ is computed, considering only the reads obtained at Step 3, and possibly the current contig is extended (see Figure 3.2, Step 4). The extension is done only if the number of reads is at least $m$ and $F(C_{new}) > |S_i| - 1$ holds. In this case, a new contig $S_{i+1}$ is built

$$S_{i+1} := S_i[0, \ldots, I(C_{new}) - 1].C_{new}$$

and the $(i + 2)$-th extension phase restarts from $S_{i+1}$. Otherwise the algorithm stops and the contig $S_i$ is returned.

### 3.2.3 Stop criteria and contigs labelling

The algorithm described in Section 3.2.2 may potentially extend a contig for an arbitrarily large number of times, without checking any "global" property of the current sequence. With our method the extension phase halts if at least one of the following conditions is met: (i) the available overlapping reads for the consensus $C$ are less than $m$; (ii) the available overlapping reads for the new consensus $C_{new}$ are less than $m$; (iii) contig length exceeds the maximum length; (iv) the seed-mate has been found.

Let $S_i$ be the contig obtained at the $i$-th step, starting from the seed read $r_0$. Criterion (i) applies when the consensus string $C$ does not exceed the current contig. This means that there are no more than $m-1$ overlapping reads, or that they are too short. In such a case, the contig produced is labelled as `NO_MORE_EXTENSION`.

Criterion (ii) applies when the consensus may have been produced as 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 this case, either too many reads are discarded (due to the presence of low-represented positions) or a significant trimming of them has been performed (as some non-represented positions occur far before the end of the consensus). In such a situation, the extension is halted and the contig is labelled as `REPEAT_FOUND`.

Criterion (iii) is satisfied as $|S_{i+1}| > L_{\max}$, 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 could not find the seed-mate. 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, labelled as `LENGTH_EXCEED`, is returned.

Criterion (iv) 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 $p \in \{0, \dots, |S_i| - |\tilde{r}_0|\}$ is checked to see whether the following condition is satisfied

$$d_H(S_i[p, \dots, p + |\tilde{r}_0| - 1], \tilde{r}_0) \leq M, \tag{3.2.8}$$

where $M$ is the maximum number of mismatches allowed between $\tilde{r}_o$ and $S_i$. Inequality (3.2.8) 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 performed on-the-fly and hence the positions already checked at the $i$-th step will not be re-checked. The *mate-check* criterion is used as a guarantee of correctness of the whole contig. This is in contrast to previous criteria, which are used to detect and prevent errors introduced in the extension phase. From this point of view, criteria (i) and (ii) can be seen as strictly *local*, since no information collected during previous steps is used. In this last case the contig returned is labelled as `MATE_FOUND`.

## 3.3 Implementation

In this section we will take a closer look at the data structures designed for our algorithm and at their implementation. GapFiller *core* is the module working during the extension phase. At this point, we assume that the set $\mathcal{R}$ has already been trimmed and possibly filtered.

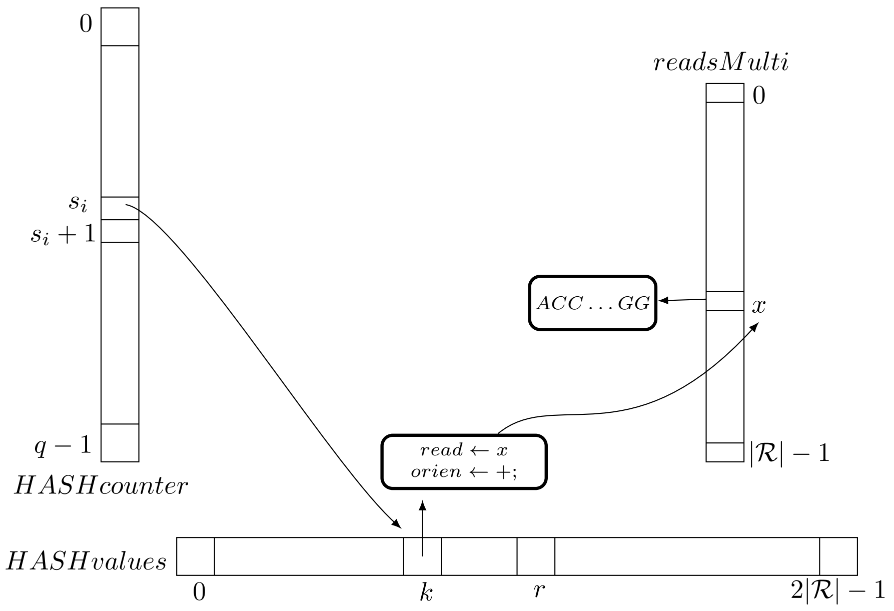


**Figure 3.3:** GapFiller data structures. The data structure used for GapFiller implementation is composed of three arrays: *HASHcounter*, whose length depends on the parameter $q$ used to compute the fingerprints; *Reads* and *HASHvalues*, whose lengths depend on the number of reads in $\mathcal{R}$. *HASHvalues* is divided in blocks, each of them corresponding to a fingerprint value; each *HASHvalues*' entry contains a pointer to an element of *Reads* and a boolean value indicating whether the read has been reverse-complemented or not.

All the data must be stored in the main memory, thus a careful engineering of the data structures is required. In the hypothesis to work with Illumina reads, we can also assume the following [151]: (i) the base quality profile decreases while going towards the end of the read, and (ii) sequencing errors in the form of INDELs are extremely rare. Hence, we decide to compute the read fingerprint on the position that should exhibit the lowest error-rate, and to restrict the search to reads that differ from one another only in terms of sequence mismatches.

## 3.3.1 Data structures

The basic idea is to pre-compute as much as possible of the useful information on the reads, in order to speed up the computation of the 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 string $C$ (see Figure 3.2, Step 1) we are always interested in obtaining overlaps between *suffixes* of $S_i$ and *prefixes* of reads belonging to $\mathcal{R}$.

In order to compute overlaps, GapFiller employs a hashing schema based on the one implemented in ERNE [169]. In particular, a data structure similar to the one proposed in [137] is built (see also Figure 3.3).

The basic idea behind GapFiller is the possibility to obtain in a fast and efficient way the set of reads whose prefixes overlap a suffix of the partial contig under construction. Therefore we used the ERNE hash function to find reads that are likely to overlap a suffix of $S_i$. Those reads are subsequently checked to see if they actually overlap $S_i$ or not.

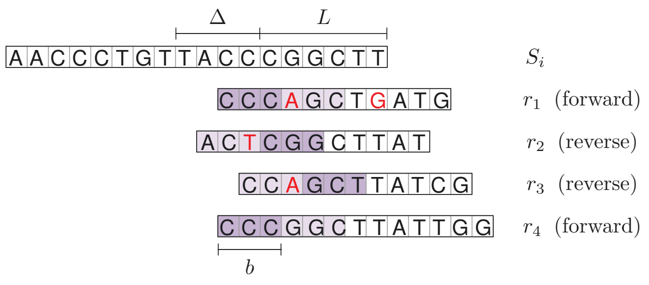


**Figure 3.4:** Reads selection by fingerprint values. The substring on which the fingerprint is computed must belong to the reads' $L$-prefix in order to be independent on the overlap length $l$. The fingerprint is computed on the leftmost substring of length $b$ for forward-stranded reads, and on the rightmost $b$-length substring for reverse-complemented ones.

Overlaps between reads and the the current contig can take place on both strands, thus reads must be stored together with their reverse complement. With the goal to save as much memory as possible, reads are represented as arrays of integers, so that a base needs 2 bits instead of 8 (A$\rightarrow$00, C$\rightarrow$01, G$\rightarrow$10, T$\rightarrow$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 the 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 reads 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. $readsMulti$: this array stores the reads and other useful information, like paired read location, paired read order (first or second in a pair), and read status (used, not used, *etc.*).

The memory requirements for GapFiller depends on the implementation. A detailed description is reported in Section 3.3.4.

## 3.3.2 Fingerprints computation

Each read fingerprint is computed on a precise substring of length $b$ (see (3.2.2)). As pointed out in Section 3.2.1, the fingerprint of $r \in \mathcal{R}$ should be computed on the position $x$ such that the (expected) average base quality is as high as possible and the substring $r[x, \ldots, x+b-1]$ falls into the contig suffix, independently on the overlap length $l$. For these two reasons, having the Illumina error-profile in mind, we choose $x = 0$ if $r$ is considered on its forward strand, $x = L - b$ if $r$ has been reverse-complemented (see Figure 3.4).

In order to compute the overlaps between the current contig $S_i$ and the reads, one has to compute the fingerprints of the substrings of length $b$ starting from $y$, for every $y \in \{|S_i| - L - \Delta, \ldots, |S_i| - L\}$ if original-stranded reads are searched, and for every $y \in \{|S_i| - \Delta - b, \ldots, |S_i| - b\}$ if reverse-complemented ones are to be extracted. Let us indicate with $s_y$ the fingerprint computed from $S_i[y, \ldots, y + b - 1]$ (see Figure 3.4). GapFiller uses this number to retrieve reads whose $l$-length prefix ($l = |S_i| - y$ for forward-stranded reads, $l = |S_i| - y + L - b$ for reverse-complemented ones) is likely to match a substring of $S_i$ close to the sequence end. In particular GapFiller accesses all $HASHvalues$ positions between

```
class Reads {
    vector <uint8_t> read;
    vector <uint8_t> readRC;
    uint16_t         information;
};

typedef struct {
    uint32_t id : 31, strand : 1;
} readOriented;

class Hash {
    vector <Reads>        readsMulti;
    vector <readOriented> HASHvalues;
    vector <uint32_t>     HASHcounter;
};
```

$HASHcounter[s_y]$ and $HASHcounter[s_y + 1]$ and, subsequently, accesses *readsMulti* to identify the set of candidate overlapping sequences $\mathcal{R}(S_i, l)$ (in Figure 3.3 GapFiller scans all positions between $k$ and $r - 1$ of $HASHvalues$). Finally, the set $\mathcal{R}(S_i, l)$ is used to compute $\hat{\mathcal{R}}(S_i, l)$, 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.

### 3.3.3 The program

GapFiller is written in C++ and the source code can be downloaded from [61]. It has been tested in Linux Operating Systems (CentOS and Ubuntu).

The program accepts both sequences (*i.e.*, FASTA and FASTQ format) and alignment (*i.e.*, SAM and BAM) files. GapFiller output consists of two main files, a FASTA file containing contigs sequences, and, possibly, a binary file containing contigs layout (*i.e.*, information on reads placement and orientation within each contig, see Chapter 4).

### 3.3.4 Memory requirements

In this section we provide a detailed estimate of the physical memory required by the algorithm, in the hypothesis that it is run on a 32-bit architecture.

The core of the implementation is a hash table, of parameter $k$, that stores a dataset $\mathcal{R}$ of reads of length $|r|$. The data structure is implemented with `class Hash`.

The sizes of `readsMulti`, `HASHvalues`, and `HASHcounter` are $|\mathcal{R}|$, $2|\mathcal{R}|$, and $2^{2k}$, respectively. During the algorithm, a single object of `class Hash` is used, and the additional RAM memory required by the assembly phase is nearly negligible (each contig is processed one at-a-time and then deleted).

Consider `readsMulti` first, whose size in bytes depends on that of `class Reads`. Members `read` and `readRC` are employed to store the numerical representation of read sequences in the original strand and in their reverse-complement, respectively. More precisely, each component of `read` (of `readRC`) is filled with 4 characters belonging to the original (reverse-complemented) sequence. Thus, the number of elements of `read` is given by $\lceil |r|/4 \rceil$.

In the worst case, the memory occupied by `read` is $4\lceil\frac{\lceil|r|/4\rceil}{4}\rceil$. The same reasoning applies to `readRC`. Taking into account the $12 + 12 + 4 = 28$ additional bytes required by `class vector` and `information`, the memory required to store an object of `class Reads` is given by $8\lceil\frac{\lceil|r|/4\rceil}{4}\rceil + 28$ that, multiplied by $|\mathcal{R}|$, provides the RAM memory used by `readsMulti`.

Concerning `HASHvalues`, consider that `readOriented` occupies 4 bytes, hence the overall memory is $8|\mathcal{R}|$. Finally, `HASHcounter` is stored in $4 \cdot 2^{2k}$ bytes.

Summing up all contributions, we obtain the following estimate for the total memory required by an object of `class Hash`

$$4|\mathcal{R}|\left(2\left\lceil\frac{\lceil|r|/4\rceil}{4}\right\rceil + 9\right) + 4 \cdot 2^{2k}.$$

For instance, the RAM memory required for a dataset of 100 bp-long reads, using $k = 15$, is approximately $100|\mathcal{R}| + 4 \cdot 2^{30}$ bytes.

## 3.4 Results on simulated data and analysis of correctness

GapFiller outputs a set of labelled contigs. The label describes the level of reliability of the sequence, in particular we divide GapFiller 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 seed-mate and hence we have no way to estimate their correctness.

We decided to perform experiments on both simulated and real data. Despite being aware that results on simulated datasets are strongly connected with the ability of read simulators to successfully reproduce realistic error schemata [170], we are also conscious that they are the only way to precisely estimate the reliability of assembled reads. In contrast, experiments on real datasets are necessary in order to test the applicability of our tool.

We simulated NGS experiments on five bacterial genomes, producing four coverages for each of them, in order to show how GapFiller performances scale at different coverages. Moreover, in order to test correctness, we aligned each output contig against a precise region of the reference, as seed reads' coordinates and orientation are known.

The experiments on real datasets were performed on public data, for which the results obtained by various assemblers are public as well. In this case, we first checked the correctness of GapFiller output contigs and then used them as input for an assembler for long reads.

### 3.4.1 Generation of simulated libraries

The reference genomes used for simulated experiments were downloaded from NCBI website [72] and we used SimSeq, the reads simulator employed in Assemblathon 1 [37], to generate paired reads coverages. More specifically, we performed our experiments on five bacterial genomes (see Table 3.1).

We generated a library (see Table 3.2) constituted by 100 bp-length paired reads, with insert size $600 \pm 200$ bp, using error profiles provided by SimSeq for reads 1 and

**Table 3.1:** Reference genomes for simulated datasets.

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

**Table 3.2:** Simulated paired reads libraries.

| Read length (bp) | Insert size (bp) | coverage 1 | coverage 2 | coverage 3 | coverage 4 |
|---|---|---|---|---|---|
| 100 | $600 \pm 200$ | $30\times$ | $50\times$ | $70\times$ | $90\times$ |

2, respectively. In particular, we obtained 20 simulated datasets generating, for each organism, four paired-ends coverages: $30\times$, $50\times$, $70\times$, and $90\times$. The reasons behind this choice lie on the fact that, on the one hand, we need at least a $30\times$ coverage in order to provide GapFiller an adequate reads distribution, and, on the other hand, we noticed that coverages equal or higher than $100\times$ do not appreciably increase GapFiller performances.

### 3.4.2 Dataset preparation

In order to avoid the generation of wrong contigs, it is of utmost importance to use only good-quality reads over the entire extension phase. Several tools are available to perform error correction on Illumina data using the so-called "read spectrum" (consider QUAKE [80], Hammer [114], and ALLPATHS [43] just to mention the most recent ones). Other tools discard reads or try to improve their reliability using quality information (ERNE [169] and QSRA [18]).

Our approach, when we are given raw data, is to first trim (and possibly filter) the reads on the ground of quality information using `erne-filter` (refer to [169] for details), and to eliminate Ns from the reads sequences. An optional subsequent step consists in correcting the reads with a tool specifically designed for this purpose, like QUAKE [80].

An important way to assess the global quality of a dataset is to plot the *$k$-mers distribution* of the reads. This can be easily done using Jellyfish [107]. If the genome has been sequenced tens of times, then two peaks are expected: one in correspondence of the expected coverage and one in correspondence of coverage one. $k$-mers composing this second peak are likely to be sequencing errors. As a rule of thumb, a low number of $k$-mers occurring only once suggests that the dataset has a good global quality.

### 3.4.3 Design of experiments

We used simulated data in order to evaluate the ability of GapFiller to correctly reconstruct the gap between two paired reads and to assess the reliability of the output classification (`NO_MORE_EXTENSION`, `REPEAT_FOUND`, `LENGTH_EXCEED`, `MATE_FOUND`). In particular we used these datasets—easy to build and validate—to explore how coverage affects GapFiller extension phase.

GapFiller performances rely on the choice of three crucial parameters: the minimum overlap length $L$, the slack $\Delta$, and the length $b$ of the substring on which the fingerprint is computed. We decided to set $L = 50$ and $\Delta = 40$, as reads length is approximately 100 bp for every library used for the experiments. The value of $b$ identifies the length of a substring on which we (almost always) require an exact matching between read and contig (see Figure 3.4), due to the fact that the employed hash function has a low false-positives rate (see (3.2.2)). We set $b = 20$ because we observed that a greater value of $b$ (*i.e.*, close to $L$) dramatically prevents GapFiller to find even few-mismatch-affected overlaps.

The parameters $T_1$ and $T_2$, necessary to discern among high/low/non-represented positions in the consensus string (see Section 3.2.2), are set to $T_1 = 0.6$ and $T_2 = 0.9$. Recall that when a position in the consensus string has a representation rate lower than $T_1$, all the reads are trimmed on that position; instead, if the representation rate is lower than $T_2$, only the reads not matching the consensus string are dropped. The value of $m$, the minimum number of reads required in order to compute the consensus string, has always been set to 2. We chose not to let $m$ depend upon coverage, since the number of reads after Step 3 strongly depends on the parameters used (say, $T_1$ and $T_2$).

We set the maximum length of a contig to be much greater than the expected average insert size, *i.e.*, 1800 bp (see also Table 3.2). We allowed for the presence of mismatches when looking for the seed-mate in the contig being constructed with parameter $M$. In all the performed experiments we set $M = 10$ (*i.e.*, approximately 10% of the reads' length). This choice is justified by the fact that the data simulated with SimSeq have a quite high amount of low-quality bases even far from the rightmost positions within the reads. The value of $\delta$, representing the maximum number of mismatches allowed when computing overlaps, depends on the overlap length $l$ and was set to $Ml/|r|$, where $|r|$ is the average read length.

### 3.4.4 Output validation

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

Due to their nature, experiments on simulated data allow to precisely estimate correctness by aligning a contig in the exact place where it is supposed to occur in the reference genome. More precisely, we used the Smith-Waterman alignment algorithm [156], 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 forward 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 200 bp, 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 contigs classification and correctness:

**Table 3.3:** GapFiller performances on simulated datasets.

| | 30× | | 50× | | 70× | | 90× | |
|---|---|---|---|---|---|---|---|---|
| Organism | Output | Time | Output | Time | Output | Time | Output | Time |
| *Alcanivorax borkumensis* | 80× | $25'45''$ | 141× | $53'14''$ | 199× | $1h30'23''$ | 279× | $2h01'03''$ |
| *Alteromonas macleodii* | 83× | $30'55''$ | 146× | $1h12'12''$ | 203× | $2h05'36''$ | 262× | $3h12'36''$ |
| *Bacillus amyloliquefaciens* | 87× | $26'40''$ | 154× | $1h01'24''$ | 216× | $1h47'51''$ | 278× | $2h44'52''$ |
| *Bacillus cereus* | 86× | $35'54''$ | 151× | $1h20'50''$ | 213× | $2h21'28''$ | 274× | $3h36'37''$ |
| *Bordetella bronchiseptica* | 87× | $35'27''$ | 153× | $1h19'34''$ | 215× | $2h19'01''$ | 276× | $3h35'01''$ |

| | Aligned | Unaligned |
|---|---|---|
| Trusted | True Positive (TP) | False Positive (FP) |
| Non-trusted | False Negative (FN) | True Negative (TN) |

This gave us the possibility not only to estimate the percentage of correctly reconstructed contigs, but also to evaluate GapFiller ability to discern between trusted and non-trusted ones.

### 3.4.5 Computational requirements and output correctness assessment

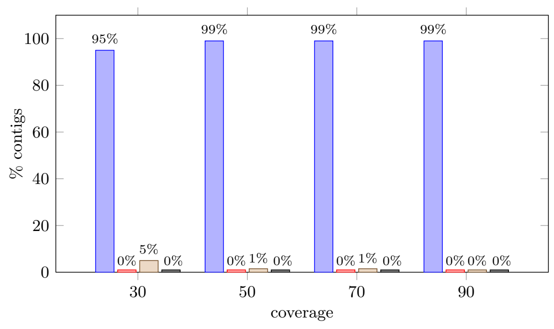
All the experiments were performed on a 8 CPU (2500 GHz) and 32 GB RAM machine. All of them required no more than 5.2 GB RAM memory. See Table 3.3 for the time requirements and for the output coverage produced for each experiment.

The experiments performed show how GapFiller performances improve as coverage increases (see Figure 3.5 and Table 3.3). From the histograms in Figure 3.5 we can clearly appreciate how the number of true positives increases with coverage, reaching an average value of 99% when coverage is above 50×. In a specular way, we can see that the number of false negatives decreases as coverage increases. Table 3.3 shows how a higher input coverage allows us to produce a higher output coverage composed by trusted contigs.

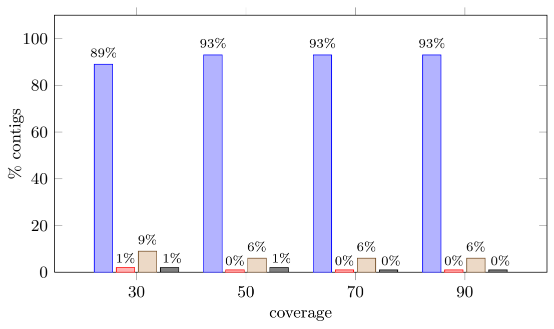
The simulated datasets allowed us to show how GapFiller is able not only to correctly reconstruct the gap between paired reads, but also to correctly flag the generated contigs as trusted (*i.e.* `MATE_FOUND`) and non-trusted (all other cases). Going into more detail, we observed that the majority of 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). An important result obtained from these datasets is that the percentage of uncovered bases is negligible, being strictly less than 0.1% even with low input coverages (*e.g.*, 30×).

## 3.5 Applications

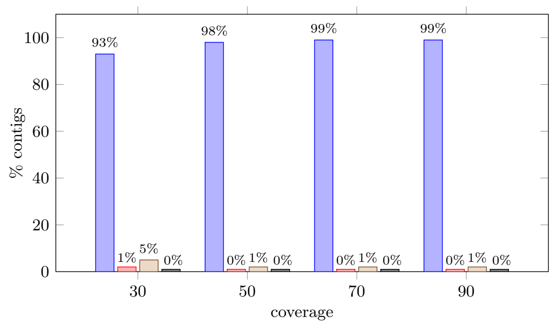
In light of the high accuracy reached by running GapFiller with different input coverages of simulated reads, we further analyzed the behavior of our tool in real scenarios. In particular, we employed sequencing data extracted from *Staphylococcus aureus*, *Rhodobacter*



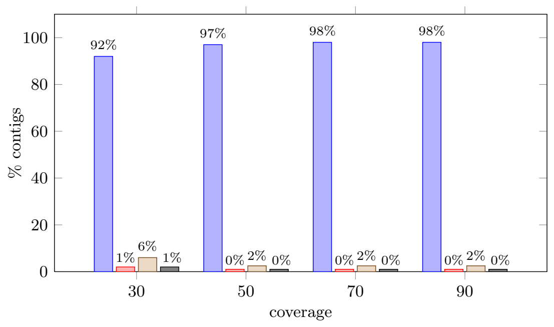
**(a)** *Alcanivorax borkumensis.*



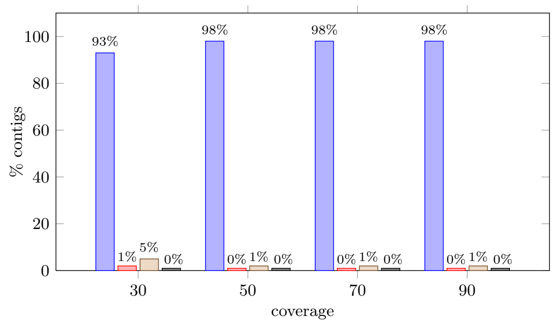
**(b)** *Alteromonas macleodii.*



**(c)** *Bacillus amyloliquefaciens.*



**(d)** *Bacillus cereus.*



**(e)** *Bordetella bronchiseptica.*

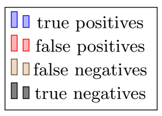


**Figure 3.5:** Results on 5 simulated datasets. The five histograms represent, for each dataset, the true positives, false positives, false negatives, and true negatives rates for different input coverages. In order to decide if a (positive or negative) contig is either true or false, it is aligned against the reference on the exact positions in which it is supposed to occur.

*sphaeroides*, and *Vitis vinifera* genomes, respectively. We employ the former two libraries for a *de novo* assembly pipeline in which GapFiller is used as a preprocessing step. The latter instead is used within a resequencing project, in particular we use our tool to assemble insertions in a resequenced variety with respect to the reference.

For the first application we know the ground truth, hence it is possible to precisely evaluate the results. We thoroughly discuss GapFiller output accuracy in the first part of the following section.

**Table 3.4:** Reference genomes and libraries for real datasets (ALLPATHS error-corrected).

| Organism | Genome len (bp) | Library | Avg read len (bp) | Insert size (bp) | Coverage |
|---|---|---|---|---|---|
| *Staphylococcus aureus* | 2, 903, 081 | Fragment | 101 | 180 | 29× |
| | | Short jump | 96 | 3500 | 32× |
| *Rhodobacter sphaeroides* | 4, 603, 060 | Fragment | 101 | 180 | 31× |
| | | Short jump | 101 | 3500 | 29× |

**Table 3.5:** GapFiller performances on real datasets.

| | Fragment | | Short jump + fragment | |
|---|---|---|---|---|
| Organism | Output | Time | Output coverage | Time |
| *Staphylococcus aureus* | 26× | 08′25″ | 517× | 3*h*34′01″ |
| *Rhodobacter sphaeroides* | 28× | 08′43″ | 230× | 5*h*27′21″ |

### 3.5.1 Whole genome assembly

In this section we will discuss the ability of GapFiller to produce accurate contigs from real data, and we analyze the results of a two-step assembly pipeline that sees GapFiller as a tool able to produce long in-silico sequences, that are subsequently used to feed an assembler for long reads.

The sequencing data for *S. aureus* and *R. sphaeroides* were dowloaded from GAGE website [58] (see Table 3.4). It is important to notice that the datasets provided by GAGE, together with the assembly results described in [143], represent the first available benchmarks that can be used to evaluate new instruments like GapFiller. These data represent state-of-the-art Illumina sequences, they are freely available, and they come with a reference sequence, a set of assemblies obtained with state-of-the-art assemblies, and with a set of evaluation scripts.

Fragment (paired-ends) and short jump (mate-pairs) libraries are available, and corrected data are provided as well. In our case, we choose to use ALLPATHS-corrected reads. For both datasets, we combined the two libraries in two ways: in a first attempt we ran GapFiller using only reads from the fragment library, while in a second experiment we used both libraries, but we selected seeds from the short jump dataset only, creating in this way contigs of average length 3, 500 bp. In particular, we set the maximum length of a contig to be much greater than the expected average insert size, *i.e.*, 550 bp and 4500 bp for GAGE fragment and short jump libraries, respectively (see also Table 3.4). The other options were set to the same values as for the experiments on simulated data (see Section 3.4.3).

GapFiller was run on a 8 CPU (2500 GHz) and 32 GB RAM machine, requiring no more than 5.8 GB RAM memory. See Table 3.5 for the time requirements and for the output coverage produced for each experiment.

When using a real dataset reads provenance is unknown, so in this case we tested output correctness by aligning the contigs against the reference genome using BLAST. We set the percentage of identity to be at least 95% and the hit length to be 100% of contig length, in order to accept an alignment. We are interested in extracting two pieces of information from alignments: the number of trusted contigs that correctly align against

**Table 3.6:** Validation of GapFiller output on GAGE datasets. The experiments performed with both short jump and fragment libraries are done by picking the seeds from the short jump library only. We state that a contig is aligned against the reference if the alignment is a single hit covering 100% of contig length and the percentage of identity is at least 95%. The statistics are computed on trusted contigs.

| | Fragment | | | | Short jump + fragment | | | |
|---|---|---|---|---|---|---|---|---|
| Organism | Avg ctg len (bp) | Aligned ctg | Aligned len | Genome cvg | Avg ctg len (bp) | Aligned ctg | Aligned len | Genome cvg |
| *Staphylococcus aureus* | 182 | 99.48% | 99.47% | 98.12% | 3648 | 98.74% | 98.76% | 95.00% |
| *Rhodobacter sphaeroides* | 188 | 99.91% | 99.92% | 98.65% | 3736 | 98.20% | 98.22% | 74.12% |

**Table 3.7:** Comparison statistics on *Staphylococcus aureus*. All percentages are referred to the true genome size.

| Assembler | # Ctg | NG50 | Chaff % | Dupl % | Comp % | Indels $\leq$ 5bp | Indels $>$ 5bp | Inv | Rel |
|---|---|---|---|---|---|---|---|---|---|
| ABySS | 301 | 29198 | 6.71 | 23.06 | 0.98 | 20 | 9 | 3 | 2 |
| ALLPATHS-LG | 59 | 96740 | 0.03 | 0.03 | 1.26 | 4 | 12 | 0 | 4 |
| Bambus2 | 108 | 50192 | 0.00 | 0.01 | 1.27 | 56 | 164 | 2 | 11 |
| MSR-CA | 93 | 59152 | 0.02 | 0.71 | 0.88 | 23 | 10 | 6 | 7 |
| **GapFiller+PHRAP** | **90** | **42398** | **0.00** | **0.28** | **1.07** | **12** | **4** | **0** | **3** |
| SGA | 1253 | 4005 | 21.34 | 0.01 | 1.26 | 2 | 2 | 1 | 3 |
| SOAPdenovo | 106 | 288184 | 0.35 | 1.42 | 1.39 | 25 | 31 | 1 | 16 |
| Velvet | 161 | 48440 | 0.46 | 0.14 | 1.31 | 6 | 14 | 5 | 9 |

the reference, as in the simulated case, and the coverage profile, as it is useful in order to estimate the percentage of genome reconstructed by GapFiller.

Table 3.6 shows GapFiller results on *Staphylococcus aureus* and *Rhodobacter sphaeroides* datasets. For both of them we run GapFiller twice: a first time using only reads from fragment library and a second time using reads from short jump library as seeds and reads from both libraries to close the gap. From Table 3.6 we can also see that, in both situations, GapFiller is able to reconstruct the insert with the expected size. Moreover, the amount of aligned trusted contigs is comparable to that obtained when simulated datasets are used as input (see Figure 3.5). The percentage of reconstructed genome is extremely high in *S. aureus* (for both experiments) and *R. sphaeroides* when fragment library is used alone. When reads from short jump library are used as seeds, instead, there is almost 26% of reference missing. This could have been caused either by a bias in the library (non-uniform mate-pairs distribution) or by the presence of difficult-to-assemble areas larger than the insert size.

In order evaluate GapFiller capabilities when used on real data, we extracted a random 10× coverage from the set of *S. aureus* output contigs (in particular from those obtained using short jump reads as seeds) and a random 15× coverage from *R. sphaeroides* output contigs (10× and 5× from those obtained using seeds from fragment and short jump libraries, respectively). Both coverages have been assembled with PHRAP [31], a well-known Overlap-Layout-Consensus assembler (see also Section 2.3.1), with default parameters. We compare our results with respect to those achieved with other assembly

**Table 3.8:** Comparison statistics on *Rhodobacter sphaeroides*. All percentages are referred to the true genome size.

| Assembler | # Ctg | NG50 | Chaff % | Dupl % | Comp % | Indels $\leq$ 5bp | Indels $>$ 5bp | Inv | Rel |
|---|---|---|---|---|---|---|---|---|---|
| ABySS | 1916 | 5872 | 1.67 | 10.07 | 0.49 | 278 | 34 | 2 | 17 |
| ALLPATHS-LG | 203 | 42455 | 0.01 | 0.38 | 0.33 | 150 | 37 | 0 | 6 |
| Bambus2 | 176 | 93198 | 0.00 | 0.00 | 0.25 | 149 | 363 | 0 | 5 |
| CABOG | 321 | 20211 | 0.00 | 0.12 | 0.71 | 145 | 24 | 1 | 9 |
| MSR-CA | 394 | 22128 | 0.02 | 1.05 | 0.53 | 179 | 32 | 1 | 8 |
| **GapFiller+PHRAP** | **1584** | **7809** | **0.12** | **0.49** | **0.76** | **158** | **14** | **2** | **7** |
| SGA | 3073 | 2284 | 3.49 | 0.05 | 0.98 | 114 | 5 | 0 | 5 |
| SOAPdenovo | 204 | 131681 | 0.44 | 1.07 | 0.54 | 155 | 406 | 0 | 8 |
| Velvet | 583 | 15665 | 0.55 | 0.29 | 0.96 | 148 | 27 | 0 | 8 |

tools and presented in GAGE [143]. It is worth noting that the assemblies presented in GAGE should be considered *the best* achievable assemblies with the employed tools. In order to obtain a comparison as fair as possible we employed the same scripts used by Salzberg and colleagues in [143]. It is important to say that the presence of a reference sequence for both the assembled genomes allows us to compute the real number of errors and mis-assemblies.

Tables 3.7 and 3.8 show the most important statistics obtained in the validation phase. For what concerns *S. aureus* assemblies, we can see that our assembly has a connectivity level (*i.e.*, number of contigs and N50) higher than that of many other widely used assemblers (*e.g.*, Velvet), moreover the number of small contigs (*i.e.*, chaffs), and the number of wrongly assembled repeats (*i.e.*, duplications and compressions) is always comparable and often better than the other assemblies. The most important columns, however, are the last four, showing the number of errors (the ideal assembler should have 0 everywhere). GapFiller+PHRAP not only is one of the assemblies with the fewest number of indels, but is also the one having less relocations (3) and inversions (0). These latest two types of errors are the most dangerous ones, due to the fact that they are the result of merging two completely different genome areas.

Results showed in Table 3.8 for *R. sphaeroides* are similar: this time GapFiller+PHRAP has a lower connectivity level (however greater than SGA and ABySS, two widely used assemblers). Also in this case our assembly is not seriously affected by indels (opposite to SOAPdenovo that has more than 550 indels). Concerning inversions and relocations, GapFiller+PHRAP performances are comparable to that of the other assemblers.

### 3.5.2 INDELS reconstruction

A difficult problem in resequencing projects is the identification, reconstruction, and validation of inserted regions within an unknown genome, with respect to a reference one. In most organisms, many structural variants are found in highly repetitive regions of the genome, making their identification difficult.

Recent studies demonstrate the feasibility of detecting structural variants using next-generation, paired-end sequencing reads [115]. In this section we will show the applicability of GapFiller in the complete or, at least, partial reconstruction of inserted sequences in a reference genome. We assume that a set of putative insertions location is known. No-

**Insertion detection.** An external program is used to identify putative insertions locations. The paired reads of the unknown organism are aligned against the reference. Reads oriented towards the insertion point are extracted.

**Insertion tips reconstruction.** The reads extracted at previous are extended with GapFiller towards the insertion point. In such a way a set of contigs is generated and those labelled as "trusted", *i.e.* spanning the gap within the paired reads, are used in the following step.

**Whole insertion assembly.** Pairs reached through GapFiller in previous step on both left and right tips are coupled to each other as they were mate pairs. The gap between the mates is filled using GapFiller again, producing a set of super-contigs.

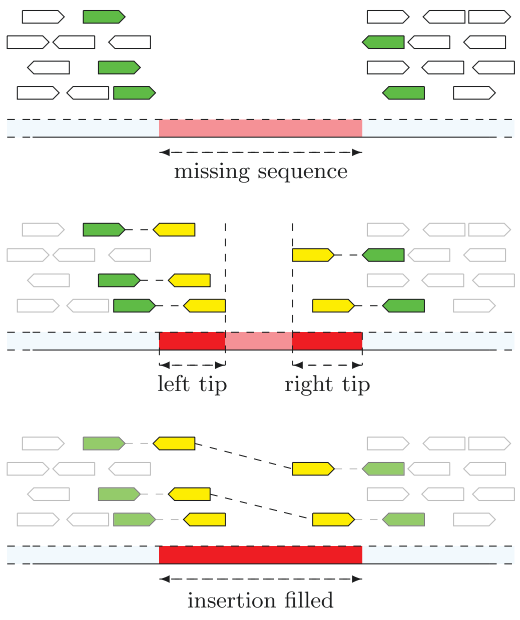


**Figure 3.6:** The three-step procedure for INDELs reconstruction.

tice that, however, reconstructing completely or at least both ends straddling the inserted sequence provides further evidence for the presence of an insertion event. Both reconstruction and validation are still weak in existing tools.

GapFiller is a tool developed to fill the gap between paired reads produced by NGS technologies, but, as a matter of fact, it can be applied to whatever pair of sequences known to lay at an estimated distance, as long as a set of uniformly distributed short reads are provided as input to fill the gap.

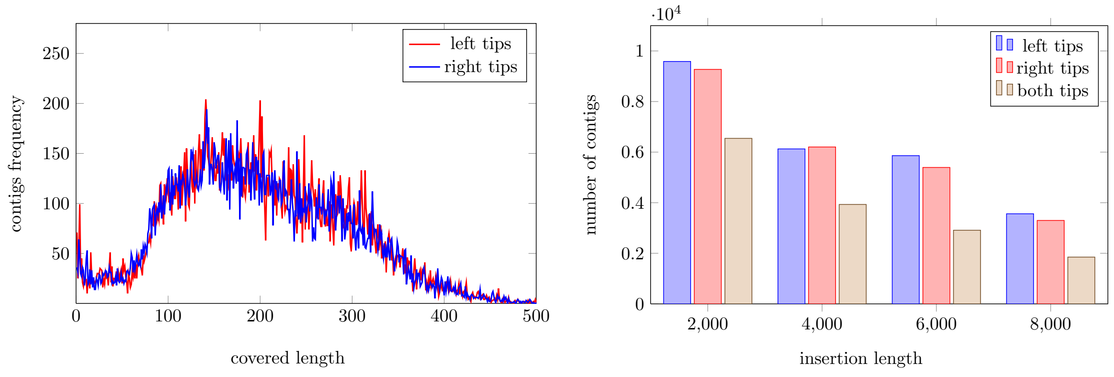
In [125] we proposed a pipeline to reconstruct and validate insertions in a resequenced individual. The procedure is divided in three main phases (see also Figure 3.6): the first one consists in detecting the insertion location and the reads aligning around them, the second one in building contigs straddling the borders of putative insertions, and the third one in filling the gap between them.

The task of determining putative insertions location is left to an external program (BreakDancer [24] in our case). Retrieving the reads placed nearby insertion points is then performed in-house using samtools [60]. More specifically, if insertions in organism $A$ with respect to organism $B$ are to be investigated, we first extract locations of putative insertions. Then the reads of $A$ are aligned against the reference $B$ and those mapping next to insertions are extracted, as well as their mates, with the proper orientation.

GapFiller is then run twice: a first time to fill the gap between the extracted paired reads in order to reconstruct the borders of each insertion, and a second time to reconstruct the sequence between the *trusted* contigs produced. In the latter phase we treat contigs on the left and on the right side of an insertion as if they were the two pairs of a paired read, respectively, and GapFiller output will be a *super-contig*. The event that the super-contig obtained starting from the left contig finally matches the right one, represents an evidence that we have reconstructed the desired sequence. Clearly, the level of confidence is a function of the number of super-contigs for each putative insertion (see also Table 3.9). Moreover, as an important byproduct, we have the assembly of the missing sequence.

**Table 3.9:** Statistics on assembled insertions: Sangiovese versus PN40024.

| | Left tips | Right tips | Both tips |
|---|---|---|---|
| Assembled tips | 2103 | 2200 | 1220 |
| Average tips length (bp) | 300 | 298 | 311 |
| Average number fo evidences | 11.5 | 10.5 | 11.4 |
| Assembled insertions | – | – | 111 |
| Minimum insertion length | – | – | 1002 |
| Maximum insertion length | – | – | 6714 |



**(a)** Contigs frequency as a function of the reconstructed tips length. Notice that the average insert size is ∼ 400 bp and reads length is ∼ 100 bp.

**(b)** Number of evidences (contigs) for insertions of different lengths. The number of contigs on both tips is the average value between those occurring on left and right tips.

**Figure 3.7:** Statistics on insertions reconstruction.

In order to check correctness we tested our pipeline on a real dataset, consisting of a ∼ 30× coverage of paired reads from the *Vitis vinifera* variety PN40024 (∼ 480 Mbp), for which the reference genome is known. The reads length is ∼ 100 bp and the average insert size is ∼ 400 bp. Using BreakDancer we extracted pairs of coordinates on the reference corresponding to deletions on the resequenced variety Sangiovese. This way we simulated insertions in PN40024, with the advantage of being able to check if the sequences assembled by GapFiller were correct, by simply aligning them against the reference.

As a preliminar validation step, for each sequence $S$ identifying a putative insertion we computed the maximal tails of $S$ covered by the trusted contigs produced in the first phase, *i.e.*, we consider only the contigs consisting of paired reads whose gap has been successfully filled. In particular, we identified ∼ 1200 simulated insertions for which we were able to correctly assemble at least 150bp on both tails (see also Figure 3.7). The average number of evidences (*i.e.*, the minimum number of contigs on both left and right insertion borders) and the average reconstructed insertion tips are reported in Table 3.9.

The second phase is more difficult as we attempt to reconstruct repetitive regions. For a few pairs of left and right contigs we were able to entirely reconstruct the inserted sequence, with respect to the reference genome PN40024.

Pairs of contigs are extracted from those for which both tails are assembled. We obtained that the gap between left and right tips is completely filled in 111 cases. The correctness of the assemblies has been proved by aligning the super-contigs against the reference, requiring the percentage of identity to be at least 98%.

## 3.6 Summary and conclusions

GapFiller is a local *de novo* assembly tool for paired reads libraries. It employs a hash table to store the reads and implements a refined seed-and-extend algorithm for inserts assembly. The employed algorithm, thanks to the refined heuristic implemented in the extension phase for consensus computation, allow to correctly reconstruct the sequence lying between the two mates of a paired read. GapFiller does not try and does not aim at assembling a whole genome but, instead, it aims at providing as output a set of insert-size contigs certified correct.

The above considerations are supported by experiments on simulated data. In fact, results show that the false positives rate is nearly negligible and that the inserts built produce a uniform genome coverage. These facts mean that GapFiller output correct sequences and that the heuristics employed allow to *globally* reconstructs also repeated sequences.

GapFiller can realize a preprocessing step, as the contigs produced can be used as input for an assembler for long reads; as an opposite strategy, it can also be used to partially assemble structural variations within an NGS resequencing project. Moreover, it can be employed to join the contigs produced by a *de novo* assembler (*i.e.*, perform gap filling after the scaffolding phase).

The promising outcomes of the studies exploited in this chapter open the possibility to use our method in diverse contexts requiring accurate biological sequence analysis.

# 4 Layout computation for local assembly

In Chapter 3 a local assembly tool was presented and some applications of this instrument were proposed. For non-trivial applications, however, the availability of the *layout* of contigs, that is, the complete information on the reads used for assembly, could be extremely important.

Consider, for example, the benefits of having additional information on how contigs have been built in a genome assembly project that uses as input the inserts produced by a preprocessing step (see Section 3.5.1). This would translate into an increase in precision, with respect to a "blind" procedure that uses pre-assembled sequences along the same lines of a shotgun assembly method.

A completely different field of application, which requires the availability of reads position within assembled contigs to allow further analyses, is that of RNA-Seq insert assembly. In this context, having an instrument for correct reads positioning is important for gene expression estimation purposes, which require prior alignment against the reference genome. The general problem of RNA-Seq reads mapping will be described in Section 5.2, whereas Chapter 6 will be focalized on a new approach to this task that uses insert information.

The above applications both enbody much importance in the field of biological sequence analysis and, thus, need robust computational methods for their tackling. In this chapter we focalize on efficient implementation and storage of contig layout in the context of GapFiller, but analogous ideas could be used for any local assembly tool.

## 4.1 Layout information and the problem of redundancy

An assembly tool like GapFiller is *local* in two aspects. First, it does not aim to assemble very long stretches of DNA (*i.e.*, chromosomes), but instead to build slightly longer sequences with respect to input reads. Second, GapFiller does not *choose* any location for the processed reads within a specified assembled sequence, resulting in the possibility of finding the same read, and, in general, the same set of reads, within many contigs. The latter consideration points out the high degree of redundancy of information concerned with layout representation.

The *layout* $L_r(C)$ of a contig $C$ is a set of 4-uples of the form $r = (id, start, len, strand)$, where $id \in \mathbb{Z}$ is the (univocal) read identifier, $start \in \mathbb{Z}^+$ is the position of the read within the contig, $len \in \mathbb{Z}^+$ is the read length, and $strand \in \{+, -\}$ is the read strand with respect to the contig. If a sorting of the 4-uples is set, field *start* can be equivalently replaced by the distance of the read from the one previously used in the contig, which will be referred to as *dist*.

The problem of storing layout information for *de novo* genome assembly has been treated yet in the context of algorithms based on the Overlap-Layout-Consensus paradigm (see Section 2.3.1). In this scenario, layout information is at the center of the assembly algorithm, in which *paths* on the overlap graph (or string graph), representing the assembled sequences, are built from the reads. Such paths can be directly translated into layout using information embedded in the graph structure. Because all-against-all overlaps among the reads are computed before the solution can be built, the amount of memory required for assembly tools based on the OLC paradigm is typically very huge (*i.e.*, up to quadratic in the dataset size).

In the context of local assembly, the technique used to join reads together can be of any type (in our case, we follow a greedy approach with GapFiller), and layout information can be provided at a later stage. This means that some fake overlaps are thrown away *before* the layout is computed, thanks to the assembly phase. However, this little reduction cannot compensate for the amount of data to be stored *globally*. In fact, each contig is assembled independently from the other ones and, thus, contains repeated information that is impossible to quantify. The global size of layout information depends on read distribution, exact insert size, and percentage of reconstructed inserts, quantities that are unknown until all contigs have been assembled.

The consequences of the above considerations are as follows: on the one hand, the huge amount of overlaps to be stored by an OLC-based assembly tool may be upper limited, but cannot be reduced much (up to transitive reduction); on the other hand, the memory required by a local assembly tool for the storage of contig layout information can be unexpectedly high, but, as its amount is basically due to redundancy in the representation, it can be substantially reduced.

In the following we describe a method for representing in a compact way the layout of each contig produced by a local assembly tool, tacking advantage of high redundancy of layout information, provided that it is represented in a convenient form.

## 4.2 A greedy method for condensed layout computation

With the aim of minimizing resources usage, being able to represent the layout in a condensed way concerning the number of its elements is a fundamental issue. If the layout of each contig is retrieved independently from the other ones, there is no way to reduce the size of layout information under a limit posed by the minimum number of bytes needed to store a single element (*i.e.*, the number of reads times the size of an element).

The idea is to use a *global list* containing each pair $(id, strand)$ only once. The layout $L_r(C)$ of a single contig $C$ will then be computed by accessing *sublists* of the global one. Clearly, this solution would be advantageous only if contig layouts share long chains.

Such an approach is not aimed at finding the best possible solution, that is, the minimal layout representation. Notice that the minimization of memory and disk requirement is of interest here, hence a method that computes the global layout list once *all* contigs layouts are available may be unfeasible. In fact, the purpose is to compute the global list on-the-fly while contigs are assembled.

In the following we describe a procedure that computes a loss-less condensed layout for each contig that is minimal in the hypothesis of using information on a single contig (*i.e.*, that currently being assembled) at-a-time.

### 4.2.1 General view of the algorithm

Our approach consists in letting contigs layout depend on other contigs', with the aim of identifying maximal common parts shared by several layouts. We make use of a structure *Layout_list* that contains all reads used to assemble a contig, and of a condensed layout structure $L_l(C)$ that contains pointers to elements in *Layout_list*, so that the original layout $L_r(C)$ can be easily accessed by scanning precise portions of the list.

In the following we will assume each layout $L_r(C)$ to be sorted in lexicographic order with respect to the triple $(start, strand, id)$ (we suppose $+$ precedes $-$ in the ordering). Because a read is univocally identified by its $id$, and because the same read would not be re-used in $C$ at the same position and with the same strand, the elements of $L_r(C)$ are all pairwise distinct with respect to the ordering defined above. The importance of this fact is essential to reach high condensation of the layout, a point that will be examined in detail throughout this chapter.

Both the sorting of $L_r(C)$ and the redundancy of the data (*i.e.* high coverages, proper of NGS experiments) are crucial to effectively shrink the size of $L_l(C)$.

The structure of *Layout_list* will depend on the order in which contigs are assembled, and is built by inserting each contig layout $L_r(C)$ properly, following an iterative procedure that simultaneously guarantees the requests below:

(i) irredundancy of representation;

(ii) conservation of contig information;

(iii) efficiency in original layout retrieval;

(iv) minimality of contigs representation.

Request (i) is met through the list representation described above.

Point (ii) refers to the fact that bifurcations in the assembly (*e.g.*, contigs sharing the prefix but not the suffix) are maintained; in other words, we do not make any choice but, in contrast, report exactly information provided by contigs as they were stored independently. The approach described here that takes advantage of a global list only affects the layout representation, not its structure.

Request of point (iii) is met with a simple modification in $L_l(C)$ that, providing direct access to *Layout_list*, allows to retrieve $L_r(C)$ with a number of operations that is linear in the number of reads of the layout.

Point (iv) is addressed by representing the layout of a contig in a minimal number of pieces, that is, those needed to handle bifurcation events or when a new chain of elements of $L_r(C)$ is added to *Layout_list*.

The procedure to build *Layout_list* on-line assures also a minimal consumption of RAM memory; indeed, the additional memory requirement at each list update step is due to the information stored for a single contig, which is nearly negligible compared to the data structure employed.

### 4.2.2 Overview of GapFiller data structure and some considerations on layout representation

Let us consider the data structure used by GapFiller to store the reads in the hash table (see Figure 3.3).

Array $readsMulti$ contains information on the reads. More precisely, $readsMulti[i]$ contains the sequence of the $i$-th read in the dataset and other information on the read (*i.e.*, read length, first or second read in the pair, read used as seed).

Array $HASHvalues$ contains pairs of the form $(i, s)$ where $i$ is the position of the read in the array $readsMulti$ and $s$ is the strand. Each element of $HASHvalues$ (*i.e.*, a "stranded" read sequence) is associated a fingerprint, and elements with the same fingerprint are grouped in contiguous windows within the array. The number of elements with fingerprint $\leq f$ is reported in $HASHcounter[f]$. Hence, pairs with fingerprint $f$ are found in $HASHvalues$ between positions $HASHcounter[f-1]$ and $HASHcounter[f]-1$ (array indexes are 0-based).

In order to store the layout in a compact way, elements of $HASHvalues$ can be used, because complete information on the reads can be easily retrieved from $readsMulti$. In fact, for each read $r$ used to assemble $C$, it suffices to store two pieces of information: an index $i$ in $HASHvalues$ and a distance $d$. Index $i$ allows to retrieve id, length, and strand of the read (with respect to $C$), whereas $d$ represents the distance of $r$ from the read previously used in $C$ ($d$ is set to 0 if $r$ is the first read). Hence, using pairs of the form $(i, d)$, instead of 4-uples of the form $(id, dist, len, strand)$, for each element of the layout, allows to store all needed information. The ordering of the elements in a single contig layout is determined by the extension phase of GapFiller algorithm and corresponds to the lexicographic ordering described in Section 4.1.

### 4.2.3 Data structures

The rules for the retrieval of contigs layout information from the dynamic list are depicted in Figure 4.1. We use two structures to store global information on the layout:

1. *Layout_list* is a list in which each element contains a pair $(i, dist)$, where $i$ is the position in $HASHvalues$ and $dist$ is the distance (in base pairs) from previous element in *Layout_list*;

2. *Pointer* is an array of size $2|\mathcal{R}|$, where the element at position $i$ is a pointer to the element of *Layout_list* containing $i$ in the first component.

Array *Pointer* is needed for direct access to the elements of *Layout_list* containing a given index. Elements in *Pointer* are in 1-to-1 correspondence with elements in $HASHvalues$.

The distance between two consecutive elements in *Layout_list* represents their distance (possibly infinite) in the layout of some contigs. In particular, given two indexes $i_1$ and $i_2$ in $HASHvalues$, there exist two consecutive elements containing $i_1$ and $i_2$ in *Layout_list* if the following conditions hold:

(a) both $i_1$ and $i_2$ are used in some contigs (*i.e.*, reads that are not used in any contig are not added to *Layout_list*);

(b) the distance $dist$ between $i_1$ and $i_2$ is finite if and only if there exists a contig $C$ such that $i_1$ is followed by $i_2$ at distance $dist$.

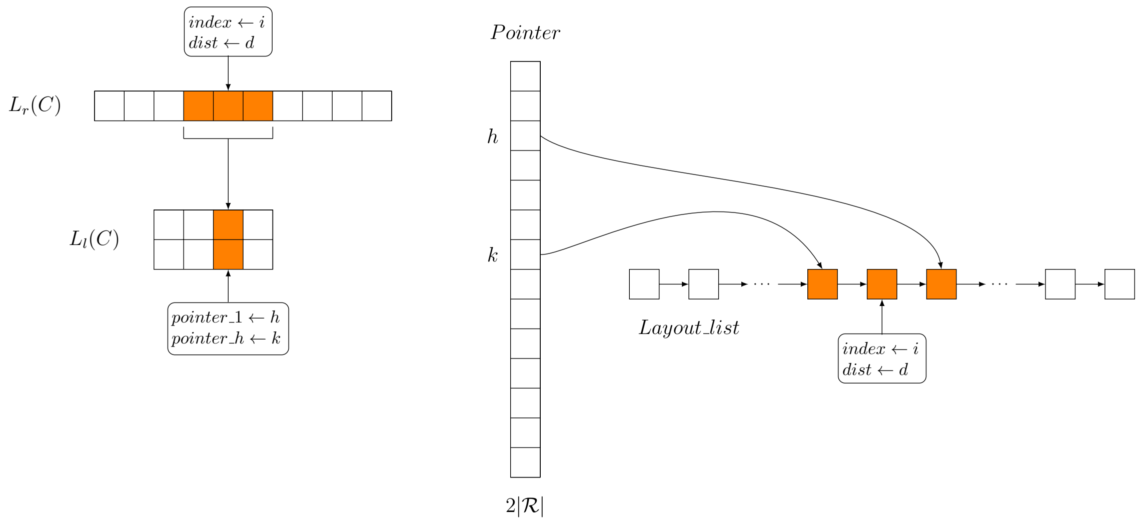


**Figure 4.1:** The dinamic data structure employed to efficiently store the layout of GapFiller contigs. The structure *Layout_list* is built sequentially every time a new contig is assembled (and its layout is retrieved). After the list is possibly updated with contig information, a new structure $L_l(C)$ is built, storing all information contained in $L_r(C)$. More precisely, an array of pairs of pointers to *Layout_list* is built, so that changes in the list do not affect pointers validity. Modifications are performed in such a way that sublists pointed by $L_l(C)$ in some previous steps are not destroyed.

The condensed layout of a contig $C$, *i.e.*, $L_l(C)$, stores local information on the layout of $C$ in terms of sublists in *Layout_list*. More precisely, each element of *Contig_sublists* is a triple containing two pointers to *Layout_list*, pointing to the starting and ending elements of a sublists, and an integer, representing the distance from previous sublist.

Condition (b) guarantees the minimality of the representation. In fact, requiring instead all contigs to satisfy (b) can cause an increase of the size of $L_l(C)$. Setting to $+\infty$ each distance that is not supported by *any* contig containing $i_1$ and $i_2$, or either one of them, requires to store information on the distance between $i_1$ and $i_2$ in a new element of $L_l(C)$, otherwise it would be lost. This is not necessary if the distance provided by the *first* contig containing both $i_1$ and $i_2$ is maintained within *Layout_list*. In other words, by minimizing the number of elements $(i, dist)$ such that $dist = +\infty$, the size of the representation of each contig layout is also minimized (modulo the order in which each contig is processed in the construction of *Layout_list*).

*Layout_list* is built iteratively by considering each contig at-a-time. The useful feature of using a list is that a condensed representation of the layout of a contig $C$ that makes use of *Layout_list* can be computed on-the-fly and can be retrieved at the end of the computation of all contigs. A branching occurring in a contig $C'$ with respect to a contig $C$ that has been processed previously may introduce changes in *Layout_list*, but the sublists pointed by the condensed layout of $C$ are not modified, hence $L_r(C)$ can still be retrieved from $L_l(C)$.

In the following we will describe an iterative algorithm that minimizes the size of the condensed layout of each contig, in the hypothesis that contigs are processed in the same order in which they are assembled. This approach does not require to store all contigs in main memory and we expect to see an increase in the size of the condensed layout as the number of assembled contigs grows.

## 4.3 An iterative algorithm for computing contig layout

In this section we describe the procedure that, starting with an empty global list, updates it iteratively by adding contig layout information, using one contig at-a-time. Before going into the details of the algorithm for the computation of the global list and the condensed contig representation, we introduce some notation.

### 4.3.1 Definitions and notations

The algorithm for the computation of condensed layout takes as input the complete information on the reads used to assemble $C$, provided with orientation and pairwise distance. As mentioned above, an index in $HASHvalues$ is sufficient to retrieve ID and strand of a read. Let us denote with $2|\mathcal{R}|$ the size of $HASHvalues$ (*i.e.*, twice the number of reads in the dataset). The layout of $C$ represented by reads, which is referred to as $L_r(C)$, will be denoted with the following

$$\langle (i_1, d_1), \ldots, (i_j, d_j), \ldots, (i_m, d_m) \rangle \quad i_j \in \{1, \ldots, 2|\mathcal{R}|\} \quad j = 1, \ldots, m, \tag{4.3.1}$$

where $i_j$ are positions in $HASHvalues$, $d_1 = 0$, and $d_j$ is the distance between the read indexed by $i_{j-1}$ and the read indexed by $i_j$, for every $j = 2, \ldots, m$.

Elements of *Layout_list* are pairs $(i, dist)$ as well. Elements of *Layout_list* are accessed through pointers, hence the structure *Pointer* is needed in the implementation. In particular, $Pointer[i]$ provides direct access the element of *Layout_list* that contains an index $i$. In this formalization, however, we do not make use of *Pointer*, which is only needed in the *Layout_list* construction phase. In fact, we will show that, once the condensed layout $L_l(C)$ is available, it is possible to retrieve $L_r(C)$ using information from $L_l(C)$ and *Layout_list* with a number of operations that is linear in the size of $L_r(C)$. We will make use of the following notation for *Layout_list*:

$$\langle (i_1, \delta_1), \ldots, (i_k, \delta_k), \ldots, (i_L, \delta_L) \rangle \quad i_k \in \{1, \ldots, |\mathcal{R}|\} \quad k = 1, \ldots, L, \tag{4.3.2}$$

where $d_k \in \mathbb{Z}^+ \cup \{+\infty\}$.

We define *sublist* to be a contiguous subsequence $S := \langle (i_{k_1}, \delta_{k_1}), \ldots, (i_{k_s}, \delta_{k_s}) \rangle$ of *Layout_list* such that $\delta_k \neq +\infty$ for all elements of $S$ except for the first one. $S$ is also *maximal* if it cannot be further extended towards either left or right directions.

The condensed layout of $C$, denoted by $L_l(C)$, is represented in terms of sublists as

$$\langle (p_1, p'_1, D_1), \ldots, (p_h, p'_h, D_h), \ldots, (p_r, p'_r, D_r) \rangle \quad h = 1, \ldots, r, \tag{4.3.3}$$

where $p_h, p'_h$ are pointers to elements of *Layout_list*.

The meaning of the above notation is as follows. The sublist delimited by $p_h$ and $p'_h$ constitutes a chain in $L_r(C)$, for each $h$. $L_r(C)$ can be retrieved by scanning the $r$ entries of $L_l(C)$, which globally contain exactly the $m$ elements of $L_r(C)$. The idea is that information on $i_j$ is all contained in *Layout_list*, whereas distances between elements belonging to different sublists of *Layout_list* are retrieved from $L_l(C)$. More precisely, we (a) guarantee that elements *internal* to sublists are equal to those of $L_r(C)$, and (b) store the distance between elements of $L_r(C)$ belonging to different sublists (*i.e.*, the last one of sublist $h$ and the first one of sublist $h+1$) within an additional field (*i.e.*, $D_{h+1}$).

In particular, a given element $(i_j, d_j) \in L_r(C)$ can be determined iteratively by previous one in $L_r(C)$ using the recurrence reported in (4.3.4), where pair $(i(p), \delta(p))$ denotes the element of *Layout_list* pointed by $p$:

$$(i_1, d_1) = (i(p_1), 0) \tag{4.3.4a}$$

$$(i_j, d_j) = \begin{cases} (i(p_k+1), \delta(p_k+1)) & \forall\, h\ p_k \neq p'_h \\ (i(p_{h+1}), D_{h+1}) & \exists\, h\colon p_k = p'_h \end{cases} \qquad i_{j-1} = i(p_k),\ j = 2, \ldots, m \tag{4.3.4b}$$

In (4.3.4b) we made use of pointer arithmetic, that is, $p_k + 1$ points to the element in *Layout_list* that immediately follows the element pointed by $p_k$. The above equations show that, given $L_l(C)$ and *Layout_list*, the original layout $L_r(C)$ is computed in $\Theta(m)$.

### 4.3.2 On-the-fly procedure for global list update and condensed layout computation

In this section the procedure to build *Layout_list* starting from a set of contigs is described. In the following we will assume that *Layout_list* is the partial global list computed until step $n$ and that $C$ is the contig assembled at step $n+1$, whose layout $L_r(C)$ is given by (4.3.1). The output of the algorithm will be (i) the *Layout_list* at step $n+1$, updated with layout information provided by $L_r(C)$, and (ii) the condensed representation of the layout, in terms of sublists of *Layout_list*, that is, $L_l(C)$.

The general idea of the algorithm is to either integrate missing information (*i.e.*, reads that have not been inserted in *Layout_list* yet) or modify *Layout_list* structure in order to maintain the number of generated maximal sublists as lower as possible. Condensed layout $L_l(C)$ can contain sublists that are not maximal, but the modifications in *Layout_list* due to the integration of $L_r(C)$ are such that the increase in the number of maximal sublists can be only due to insertion of new reads. This means that, once they are assigned, distances among elements of *Layout_list* cannot be modified, unless they are equal to $+\infty$. As a by-product, the size of the condensed layout already computed does not change when new contigs are introduced.

In the algorithm, every time a new layout $L_r(C)$ is considered, we identify two (possibly empty) chains $C_q$ and $C_s$ within $L_r(C)$, either satisfying one of the following two properties, respectively: $\mathcal{P}_1$, stating that no elements of the chain belong to *Layout_list*, and $\mathcal{P}_2$, stating that the chain corresponds to a sublist of *Layout_list*.

Notice that we can restrict ourselves to the case in which $C_q$ and $C_s$ are maximal with respect to either $\mathcal{P}_1$ or $\mathcal{P}_2$ properties, respectively, meaning that they cannot be further extended in $L_r(C)$ to longer chains without violating the property. Hence, there are three distinct cases:

**Case 1:** any element of $C_q$ do not belong to *Layout_list*, whereas those of $C_s$ do;

**Case 2:** all elements of $C_q$ belong to *Layout_list*, whereas those of $C_s$ do not;

**Case 3:** both elements of $C_q$ and $C_s$ belong to *Layout_list*, but the corresponding sublists are not consecutive to each other.

In order to identify $C_q$ and $C_s$, use two pairs of pointers, $(p_q, p'_q)$ and $(p_s, p'_s)$, representing start and end, respectively, of sublists in *Layout_list*. We suppose that elements of $L_r(C)$ still missing in *Layout_list* are previously stored in a temporary list with the same structure, so $p_q$ and $p_q$ are well-defined, and we assume $\delta(p_q) = +\infty$ or $\delta(p_s) = +\infty$, respectively. We will also denote with $j(q)$ and $j(s)$ the positions in $L_r(C)$ in which $i(p_q)$ and $i(p_s)$ are found, respectively (notice that $j(q)$ and $j(s)$ can be obtained in an iterative manner, similarly to (4.3.4), hence we can assume that they can be accessed in constant time, because they will be used only after a scan of $L_r(C)$ has been performed).

At the beginning of the algorithm, the *Layout_list* computed at step $n$ is available and the layout $L_r(C)$ of the $(n+1)$-th contig $C$ is given in input. In the following we describe the update of *Layout_list* and the construction of the condensed layout $L_l(C)$ at the generic iteration, within step $n+1$, identified by maximal chains $C_q$ and $C_s$ of $L_r(C)$.

### Case 1: forward insertion

With the aim of minimizing the number of sublists created, it would be desirable to insert $C_s$ immediately *before* $C_q$ in *Layout_list*. This is possible if and only if $p_s$ points to the first element of a maximal sublist, that is, $d(p_s) = +\infty$. In such a situation, $C_q$ is inserted before $C_s$ and $\delta(p_s)$ is set to $d_{j(s)}$. The triple $(p_q, p'_s, d_{j(q)})$ is inserted in $L_l(C)$.

If, instead, $d(p_s) < +\infty$, then a putative branching is found (*e.g.*, different sequences sharing the suffix). In such a case, $C_q$ is inserted after the last element of *Layout_list* and two triples are inserted in $L_l(C)$: $(p_q, p'_q, d_{j(q)})$ and $(p_s, p'_s, d_{j(s)})$.

### Case 2: backwards insertion

This case is analogous to previous one, with the difference that here we attempt to insert $C_s$ right after $C_q$. In order to do that, either $p'_q$ points to the last element of the list, or $p'_q + 1$ is such that $\delta(p'_q + 1) = +\infty$. In this case, the triple $(p_q, p'_s, d_{j(q)})$ is inserted in $L_l(C)$.

The case in which $\delta(p'_q + 1) \neq +\infty$ causes $C_s$ to be inserted at the end of *Layout_list*. Hence, the triples $(p_q, p'_q, d_{j(q)})$ and $(p_s, p'_s, d_{j(s)})$ are added to $L_l(C)$.

### Case 3: sublists relocation

In this case both $C_q$ and $C_s$ are present in *Layout_list* as sublists, but they are not consecutive to each other. The idea here is that $C$ may contain evidences for $C_q$ and $C_s$ to be joined and, as a consequence, for the number of maximal sublists of *Layout_lists* to be decreased. Indeed, $p_s$ points to the beginning of a maximal sublist and joining $C_q$ and $C_s$ will cause $\delta(p_s)$ to be set to the distance between elements in $L_r(C)$.

Following the rule telling that we can move sublists only if they do not break contiguous pieces of layout, we check the value of $\delta(p_q)$ and $\delta(p_s)$, and, in case neither $p'_q$ nor $p'_s$ point to the last element of *Layout_list*, we also check the value of $\delta(p'_q + 1)$ and $\delta(p'_s + 1)$.

More precisely, $C_s$ can be placed right after $C_q$ only if the following three conditions hold:

(a) either $p'_q$ points to the last element of *Layout_list* or $\delta(p'_q + 1) = +\infty$;

(b) $\delta(p_s) = +\infty$;

(c) either $p'_s$ points to the last element of *Layout_list* or $\delta(p'_s + 1) = +\infty$.

Similarly, $C_q$ can be placed right before $C_s$ only if (a), (b'), and (c) conditions hold, where

(b') $\delta(p_q) = +\infty$.

If either one of previous sets of conditions are met, then $\delta(p_s)$ is set to $d_{j(s)}$ and the triple $(p_q, p'_s, d_{j(q)})$ is added to $L_l(C)$. Otherwise, *Layout_list* remains unchanged and the two triples $(p_q, p'_q, d_{j(q)})$ and $(p_s, p_s, d_{j(s)})$ are both added to $L_l(C)$.

Contig $C$ is scanned until $p'_s$ points to the last element of *Layout_list*. At this point, $L_l(C)$ constitutes a succint representation of the layout of $C$. Indeed, iterating trough the elements of *Layout_list* delimited by pointers in $L_l(C)$ allows to retrieve full information contained in $L_r(C)$, as described in Section 4.3.1. Notice that every time a break is generated (cases 1 and 2), the distance of the element following the breakpoint is not changed, hence the layout of a contig containing the sublist covering the breakpoint remains valid. On the other hand, the layout of a contig that generates the break is available too, because information on the distance between the two elements around the breakpoint is stored in $L_l(C)$.

The overall procedure for the computation of *Layout_list* requires a linear scan of all contigs. If $M$ is the sum of the sizes of $L_r(C)$, with $C$ varying in the set of assembled contigs, then the procedure described above takes time $\Theta(M)$. Indeed, the operations performed on elements of *Layout_list* at each iteration (*i.e.*, either case 1, 2, or 3) are in constant number, thus they do not increase the computational complexity.

## 4.4 Resources requirements

In this section we analyse the resources usage of the algorithm described in previous section. Moreover, when propose a method for efficiently store the layout information for each contig in main mamory, in order to be used for further applications, in such a way that the hash table is not needed.

The code was written in C++ as a part of GapFiller module. In the following we will assume that the software is run on a 32-bit machine.

### 4.4.1 RAM memory usage for layout data structure

The data structure used for the implementation of the layout is composed of *Layout_list* and *Pointer*. The memory required to store contig layout is negligible, because at any stage of the algorithm, at most one contig is processed and stored.

Each entry of *Layout_list* contains two fields: a position in *HASHvalues* and an integer representing the distance from previous element. Value $+\infty$ is represented with a value larger than each "finite" distance. Elements of *Pointer* instead, are pointers to elements of *Layout_list*.

The layout produced by GapFiller is useful for further applications and should be stored in such a way that it can be easily retrieved without re-creating the data structure, hence, it should not depend on *HASHvalues* indexes, nor on pointers to *Layout_list*. For this reason, we make use of a new structure *Layout_vector* containing the same elements of *Layout_list* in the same order, but represented as 4-uples of the form $(id, dist, len, strand)$

instead of pairs $(id, dist)$. In this way, sublists pointed by elements of $L_l(C)$ can be directly accessed and the knowledge of the hash table is not needed. Conversion from *Layout_list* to *Layout_vector* and from pairs to 4-uples is done in linear time with the size of *Layout_list* and the sum of the sizes of $L_l(C)$, for each contig $C$, thanks to $HASHvalues$ and to an additional array in the data structure for *Layout_vector*.

```
typedef struct {
    uint32_t    index;
    uint16_t    dist;
} list_element;

typedef list <list_element>::iterator list_pointer;

class Layout_list {
    list <list_element>   list;
    vector <list_pointer> pointer;
};

typedef struct {
    readOriented  index;
    uint16_t      dist;
    uint16_t      length;
} read_layout_list;

class Layout_vector {
    vector <read_layout_list>  layout_vector;
    vector <uint32_t>          positions_in_list;
    Layout_list *              layout_list;
};
```

The data structures for the layout computation are implemented with `class Layout_list` and `class Layout_vector`. After *Layout_list* is computed, its content is transferred to *Layout_vector* in order to allow the representation of $L_l(C)$ as pairs of positions in the array. Elements of *Layout_vector* are also converted into a $t$-uple that does not depend on the hash table. The data structure *Layout_vector* is implemented with `class Layout_vector`.

Within `class Layout_list`, `list` and `pointer` contain up to $2|\mathcal{R}|$ elements, where $\mathcal{R}$ is the set of input reads (indeed, elements of `list` represent pairs $(id, strand)$). Structure `list_element` occupies 8 bytes in memory (due to padding after `dist`), whereas `list_pointer` is stored in 4 bytes. Hence, the overall RAM memory required by `class Layout_list` is approximately $(8 + 4) \cdot 2|\mathcal{R}| = 24|\mathcal{R}|$ bytes.

Concerning `class Layout_vector`, `positions_in_list` has size $2|\mathcal{R}|$ and its $i$-th element is the position in *Layout_list* in which the $i$-th element of $HASHvalues$ is found. Apart from the space required by pointers, the overall memory required by `Layout_vector` is $(8+4)\cdot 2|\mathcal{R}|$ bytes, where elements of `layout_vector` contribute for 8 bytes each, whereas a single element of `positions_in_list` is stored in 4 bytes. Hence, the RAM memory required by *Layout_list* and *Layout_vector* is approximately the same. Because elements of *Layout_list* are iteratively erased while *Layout_vector* is built, the overall memory required by the procedure is still $\sim 24|\mathcal{R}|$ bytes.

### 4.4.2 Compact layout storage with parametric type size

The overall information on the layout of each contig is available from *Layout_vector* and $L_l(C)$. We denote with $m$ the number of elements of *Layout_vector*, with $m \leq 2|\mathcal{R}|$, each elements being of the form $(id, dist, len, strand)$. Information to be stored for $L_l(C)$ consists of (i) contig ID, (ii) number of elements, (iii) triples $(start, end, strand)$ representing sublists. $L_l(C)$ contains $r$ elements, with $r \ll |\mathcal{R}|$.

In order to efficiently store *Layout_vector* and $L_l(C)$ structures, we use two ideas: (1) print unformatted output in order to minimize bits padding, and (2) use only the strictly necessary number of bits to store each piece of information. All entities are processed before output is printed, and maximal values for data sizes are known. In the following we will denote with $|q|$ the maximal number of bits needed to store any value of $q$.

The minimum amount of bits needed to store *Layout_vector* is $m(|id|+|dist|+|len|+1)$. Similarly, $L_l(C)$ needs at least $r(|start|+|end|+|dist|)$ bits to be stored. A minimal waste of resources is guaranteed using unformatted data structures, that is, employing the bits actually needed for each field and using the ones that are possibly left to store further information. Based on this idea, we define an *ad hoc* succint output format.

In general, a data structure $S$ can be stored in an unformatted way into an array $v$ of $b$ bytes. Each element of $v$ is a block that cannot be broken into smaller pieces, that is, a waste of up to $8b-1$ bits can occur in the form of padding at the end. Thus, the total amount of bytes needed to store $S$ is given by $b\lceil |S|/8b \rceil$.

For simplicity, we choose $b$ as the minimal number of bytes needed to store the largest element among $id$, $dist$, $len$, $r$ (notice that contig ID is always smaller than the maximal value for $id$). In realistic scenarios, we never exceed 32 bits, hence we set $b = 4$.

Consider an NGS experiment that outputs a set $\mathcal{R}$ of length 100. In this case, we have $|id| = \lceil \log_2 |\mathcal{R}| \rceil$ and $|dist| < |length| = 7$, and the size of *Layout_vector* is at most $|\mathcal{R}|$. Hence, the disk space required by *Layout_vector* is approximately $4\lceil |\mathcal{R}|(\lceil \log_2 |\mathcal{R}| + 15)/32 \rceil$ bytes.

## 4.5 Summary and conclusions

In this chapter the problem of efficient storage and retrieval of assembly layout information is analyzed. In particular, we focused on local assembly, where layout information is not crucial for overlap computation, nevertheless it can be important for further applications in which local assembly is used as a preprocessing. Our approach would be worth applying in the hypothesis of high redundancy of the data: in such a context, it is possible to identify quite long conserved chains in constant space. High coverages of NGS data seem the ideal environment for the success of our method, that we aim to use with both genomic and transcriptomic data. RNA-Seq reads, in particular, present the difficulties of having extremely divergent coverages, due to high variability in gene expression levels. We accurately estimated the resources requirements and, even though it is not possible to predict in advance the space occupied by our condensed layout representation, we expect to dramatically decrease disk space requirements.

GapFiller is provided with a light data structure to store all the reads, a fast overlap search method employed with a well-suited hash function, and, finally, an efficient algorithm to store layout information in a condensed way. Our conclusion is that GapFiller, being able to output correct sequences, constitutes a valuable add-on in a number of genomic and transcriptomic studies; indeed, its efficiency and low memory requirements, substantially lower with respect to traditional *de novo* assembly algorithms, make it practically usable as a preprocessing step.

# II

# Applications to RNA-Seq data

# 5 RNA-Seq data analysis

The understanding of the transcriptome is of utmost importance for a full comprehension of cells metabolism. RNA-Seq technologies allow to tackle all issues related to the comprehension of the transcriptome: *de novo* or reference-based assembly, novel isoforms annotation, and transcripts quantification. Despite the intrinsic biases related to the multiple steps required by libraries preparation for sequencing, RNA-Seq technologies carry many advantages for transcriptomics studies. For this reason they have been established as the standard method for massive-scale analyses.

The main problems related to the comprehension of gene expression are: finding genes loci, identifying alternative transcripts, and estimating transcripts abundances. The availability of both a reference genome and, possibly, a partial annotation provide important knowledge that, together with RNA-Seq data, allow to tackle the above mentioned problems with a degree of precision (*i.e.*, single-base resolution) that was impossible at the time microarray technologies were the main source of information for these types of study. RNA-Seq allows various types of analysis even when the annotation is completely missing, as EST sequencing, but at a fraction of cost and with higher throughput. In the first section of this chapter we discuss how the intrinsic limitations of RNA-Seq technologies, especially on reads length, are usually partially overcome providing suitable coverage and paired reads libraries.

The main purpose of this chapter is to give an overview of methods and tools for reads alignment and splicing junctions detection, transcriptome assembly, in presence of either a reference genome and/or an existing annotation, and transcripts abundances estimation.

Evaluation of assembly and alignment with RNA-Seq data are complicated tasks and their standardization is still lacking in literature. Consider that genome assembly, for which both coverage information and total assembly length are usually known or can be quite accurately estimated, pose several challenges for evaluation and are still at the center of significant research efforts.

At the end of this chapter we address validation issues in the context of transcriptomic analyses, with particular attention at some approaches proposed in literature attempting, if not to design statistical methods for transcriptome reconstruction evaluation, at least to compare state-of-the-art tools, either in the context of RNA-Seq reads alignment (and splicing sites detection as well) or concerning the task of assemblying the transcriptome.

## 5.1 Applications of RNA-Seq: genome annotation and transcripts quantification

When RNA-Seq experiments are performed, two main applications are usually envisaged: annotation and quantification. The genome annotation problem is concerned with understanding of genes structure, finding all types of expressed transcripts in a sample, detecting alternative splicing events, and calling post-transcriptional modifications (*e.g.* , structural variants, RNA editing, allele-specific expression, *etc.*).

Quantification is commonly measured in RPKM units (see also Section 1.2.3) and represents the amount of molecules produced for each type of transcripts.

Annotating new transcripts and refining an existing annotation are tasks that, before the advent of the so-called NGS technologies, could be solved with EST sequencing. The expensiveness of traditional sequencing technologies limited the application of cDNA sequencing to transcriptome assembly, whose outcome, due to low throughput proper of EST sequencing, is not reliable especially for extremely low expressed transcripts. What made the matter even worse is that some low expressed transcripts may be produced by alternative splicing of highly expressed genes, making their identification almost impossible.

Concerning expression levels estimation, the most widely used technique were those based on microarrays. Such technology is quite cheap but several defects limit their application. If, on the one hand, microarrays allow to find expression levels of a set of target genes, the result strongly depends on an existing annotation and the output, in general, lacks in precision. Indeed, microarray-based technologies suffer from the so-called non-specific *cross-hybridization* among genes within the same sample, and also do not allow to precisely detect transcripts abundances.

RNA-Seq reads alignment allows the identification of exons, the annotation of splicing sites, and the computation of expression levels. The latter problem is usually addressed by counting the reads aligned onto gene sequences, normalized by the exons total length [165]. A major challenge consists in the computation of expression levels of alternative transcripts, as the reads must be assigned to the right isoform they come from, a task that is particularly difficult for genes whose alternative transcripts only slightly differ from one another. For the above reason, capturing almost all the reads coming from a certain region, possibly with gaps, is of crucial importance.

### 5.1.1 Design of RNA-Seq experiments

The ability of both correctly detect new isoforms and compute accurate expression levels not only depends on specific care in the algorithms and tools designed to tackle these tasks, but also on the steps taken at the library preparation stage. Issues concerning both the sample or tissue from which RNA is extracted and the type of analyses one wants to carry out require targeted RNA-Seq library preparation.

Among the problems related to the RNA sample, we mention the high level of ribosomal RNA content, the level of degradation, and the RNA amount, the latter being a serious issue for important types of studies (*e.g.*, in cancer). Library preparation must also be performed to target the specific problems to be studied, for instance, paired reads are usually required if accurate reads mapping is considered a central step (*e.g.*, for novel splicing sites detection, which require single-base resolution) and are unavoidable for gene fusion events discovery [105]. Moreover, normalized cDNA libraries may help accurate transcripts reconstruction, whereas may not be suitable for quantification purposes. For the same reason, duplicate reads removal is recommended for the former task, whereas can introduce biases in the latter one (duplicate reads can be either an effect of PCR biases in presence of low GC-content or the product of highly expressed genes).

Also considering RNA-Seq data yield is fundamental. Even though it is not possible to accurately estimate transcriptome coverage, being it a possible and, in fact, standard, issue, concerning studies on RNA it is only possible to foresee the transcriptome complexity by looking at genome length [161]. For instance, trancriptome assembly do not require deep coverage, at least if the main form of each expressed genes is to be analyzed. Instead, for the comprehensive discovery of alternative splicing events, including extremely low expressed transcripts, and for the detection of rare variants, Principal Investigators should foresee an amount of reads that roughly doubles the dataset size ideal for the former problem.

Entering into the details of the issues of preparing sequencing libraries for RNA-Seq experiments is beyond the scope of this thesis. However, knowledge of the practical design of such experiments enables us to both target the solution to the biological problem with a know-how of the data we are going to use, and design realistic simulated library that should allow us to precisely evaluate our results, with an eye towards the actual applicability of our methods to real RNA-Seq experiments.

## 5.2 RNA-Seq reads alignment and splicing sites detection

The most widely used approach addressing the analysis of the transcriptome when a reference genome is available and the annotation is either partial or absent, is to directly align the cDNA reads against the genome. Tools addressing this task need to map the reads allowing large gaps. Indeed, splicing junctions can be identified by gaps in the alignment delimited by donor-acceptor dinucleotide pairs among exons (see also Figure 1.2). Moreover, the end of a transcript can be identified by the presence of a poly-A tail (*i.e.*, a monomer of As or Ts) within a read.

NGS technologies initially provided very short reads (*e.g.*, 25-36 bp for Illumina Solexa), hence gapped alignment was nearly impracticable due to ambiguity (*i.e.*, read sub-sequences coming from consecutive exons may be extremely short). Recent sequencing technologies improvements allowed production of longer reads (*i.e.*, up to 150 bp for Illumina HiSeq and up to 300 bp for Illumina MiSeq) and made it possible to perform safer spliced alignment. Moreover, if paired reads are provided, additional information on reads location is available and can aid mapping.

Several tools have been proposed in the last few years addressing the problem of efficiently mapping RNA-Seq reads against a genome. This task is not trivial because of the subtle trade-off between alignment sensitivity and junction prediction specificity. Allowing for errors in the alignment is mandatory in order to cope with errors within RNA-Seq reads sequences, however, the combination of short query length (*i.e.*, a fragment of a read in case of split alignment) and approximate mapping could result in the extreme difficulty of unambiguously locate reads on the reference. For these reasons, new techniques for RNA-Seq reads mapping have been proposed to specifically and efficiently address gapped alignment. The available tools can be roughly divided into two classes: *exon-*

*first* and *seed-and-extend.* The former alignment paradigm often take advantage from an external tool for ungapped reads alignment, whose speed and precision is crucial for further alignment steps. Some fast tools for genomic reads alignment are Bowtie [89], BWA [91], and ERNE [169]).

In Section 5.2.1 and 5.2.2 we put forward a brief overview of software tools for RNA-Seq reads mapping belonging to the two classes mentioned above, which do not require an annotation as input (see also [1, 40, 41]). The vast majority of tools for splice mapping report also unspliced reads, a feature that is useful for subsequent expression levels quantification, other than splicing sites prediction.

### 5.2.1 Exon-first alignment methods

The exon-first spliced alignment paradigm was the first one proposed to specifically handle RNA-Seq reads, yet in 2008, with QPALMA [30]. The general idea of the exon-first algorithm is as follows: (i) reads are aligned against the reference without allowing gaps, then (ii) contiguous alignments are merged to form clusters on the reference genome, identifying putative exons, and finally (iii) reads that turn out to be unmapped at step (i) are employed to connect clusters and to predict splicing sites. Among the tools based on the exon-first paradigm we cite also Tophat [164], SpliceMap [7], MapSplice [172], HMMsplicer [32], SOAPsplice [75], PASSion [181], and TrueSight [98]. Some of them employ a hybrid approach, that is, they use an exon-first approach for ungapped reads mapping and align split reads in a seed-and-extend fashion; it is the case of QPALMA, SpliceMap, MapSplice, HMMsplicer, and TrueSight.

QPALMA implements an alignment technique based on the tool Vmatch [74]. The first step of unspliced alignment is performed, but before flagging a read contiguously mappable against the genome as "unspliced", futher checks are performed. The idea is that a read may be incorrectly flagged if $r$ spans a splice junction that falls next to one of the read's tails. The second step involves spliced alignment of reads that are still unmapped, or they are found to have been incorrectly mapped contiguously. Vmatch is employed again to align one half of the read first, and depending on the result each read is classified as either *low-quality* or *spliced.* In the latter case, alignment is done in a Smith-Waterman-like fashion. A refinement of the traditional algorithm is proposed, namely, the authors set optimal parameters for the dynamic programming procedure by solving an optimization problem that maximize parameters value in case of wrong alignments. The last step consists of splicing junctions refinement. This is done through a Support Vector Machine (SVM) [119] trained on a database of known splicing junction.

Tophat is a widely used aligner for RNA-Seq reads and is the main module based on Bowtie. The procedure consists in first aligning the reads without allowing any gap. Locations of the reads in the genome are then used to create clusters that identify putative exons locations. The assembly step is performed with a reference-guided assembler (*i.e.*, MAQ [93]). Clusters are further extended beyond their ends; at this point, pairs of consecutive clusters can be either joined or not. In the latter case, $k$-mers adjacent to the ends of two consecutive clusters are employed to find splicing junction locations. This phase is performed employing reads that result to be unmapped by Bowtie and that match those $k$-mers.

SpliceMap differs from the standard exon-first method, because it starts by mapping *half-reads* against the reference genome instead of the entire read sequences. The authors use an external tool for unspliced alignment (*e.g.*, ELAND or SeqMap). The idea that, if a read spans exactly one splice junction, then at least one of its halves must map contiguously. By setting to 25 the minimum length of a fragment that can be unambiguously mapped against the genome, the procedure starts by dividing each read into disjoint 25 bp-long fragments (*half-reads*). Half-reads are aligned independently, "exonic hits" are selected depending on their location in the genome, and possibly (locally) multiply aligned reads are discarded. Exonic hits are used to feed a seed-and-extend algorithm that expands each half-read location base-by-base, until a splicing nucleotide is found. The remaining part of the read that result to be unmapped is then placed at a certain distance from the seed, using a hash function to efficiently search for matching along the genome. SpliceMap allows also the employment of paired reads to enhance the specificity of splicing junctions prediction.

MapSplice employs a similar method with respect to Tophat and SpliceMap, because alignments are assembled before performing splicing sites prediction. The difference consists in the technique employed for alignment. Contiguous and spliced alignments are performed simultaneously. Indeed, MapSplice performs a "tag alignment phase" by first splitting each read in a number of tags and then mapping them against the reference. The alignment type (*i.e.*, contiguous or split) is determined by the location of tags in the genome. Differently from SpliceMap, that sometimes considers multiple alignments to be unsafe, MapSplice allows multiple alignments for each tag and discerns among them using splicing information. More precisely, in the "splice inference" phase, the best alignments are found by maximizing both alignment quality and splicing site confidence. Thus, splicing sites prediction is done without assuming any hypothesis on splicing dinucleotides, nor setting any threshold on introns size.

HMMsplicer is addressed at finding only spliced alignments. Indeed, it uses Bowtie to map reads against the reference genome, hence reads successfully aligned are not considered for the subsequent steps. Unmapped reads are cut in two pieces (called *read-halves* by the authors) that are aligned independently. The splicing position is inferred using a Hidden Markov Model (HMM) [38] trained on a subset of read-halves aligned to the genome. Each read-half alignment is extended with the remaining part of the read; the same procedure is done independently on the other read-half referred to the same read. At this point, read-halves alignment are possibly merged to find the final read mapping. From spliced alignment a set of junctions is found and each element is assigned a score based on read alignment.

SOAPsplice employs the Burrows-Wheeler transform [19] to improve on space consuption while storing the reference sequence and efficiently search for reads placement. Two different algorithms are employed, for spliced and unspliced alignment, respectively. Firstly, SOAPsplice tries to align RNA-Seq reads contiguously. The reported hit is the best one obtained in terms of number of mismatches or number of indels. Reads that that cannot be mapped in a single hit are divided in two parts (or more than two, if reads are longer than 50 bp) and the 5' tail is aligned first. The algorithm tries to find the longest read prefix that can be aligned without (large) gaps. Then, the remaining part of a read is searched, giving suitable criteria for the choice of the best alignment when multiple hits are reported, in terms of number of mismatches/indels, distance with the mapped fragment, and splicing dinucleotides (priority is given to the canonical pair GT-AG).

PASSion takes as input only paired reads libraries and is structured in five steps. The first one consists in aligning reads with SMALT [73]. The second step is the creation of the so-called *islands*, which are maximal contiguous areas on the genome that are covered by some reads, through the samtools pileup command. Islands are further extended on both sides to incorporate intron neighbouring sequences. The third step consists in reads remapping. More precisely, paired reads such that only one mate has been mapped at first step are aligned on islands through a pattern-grow algorithm. The procedure consists in finding two alignments for each unmapped mate, one for the left tail and one for the right tail, respectively, that are extended until the entire read sequence is covered. If this happens, then a gapped alignment is found. In the fourth step splicing junctions are filtered, that is, gapped alignments found at previous step are retained only if read depth is beyond a threshold. In the fifth and last step each splicing junctions is refined so that its boundaries match a dinucleotide pair (*i.e.*, GT-AG, GC-AG, or AT-AC).

TrueSight uses a different approach that jointly employs read mapping quality and sequence coding potential to retain only trusted splice junctions. A useful feature of this tool is that spliced mapping is not limited to a single spanned junction. On the contrary, the algorithm allows to find multiple junctions spanned by a single read. This feature is also present in TopHat, MapSplice, and PASSion. TrueSight uses Bowtie for alignment. First, entire reads are aligned without gaps. They are not re-considered for gapped alignment, however, they are included in the regression model employed for splicing sites prediction. Spliced aligment is performed for reads that cannot be mapped contiguously, with a seed-and-extend technique. Reads are cut into pieces that are aligned against the genome. If multiple alignments are found for those read pieces, then every possible combination is considered. For every spliced read, an anchor is selected and the alignment is extended starting from the anchor and following the path that allows to reach the highest alignment score.

### 5.2.2 Seed-and-extend alignment methods

Seed-and-extend alignment tools, differently from those based on the exon-first approach, employ reads alignment in both unspliced and spliced forms at the same time. Some aligners based on this idea are MapNext [8], SuperSplat [17], GSNAP [177], SplitSeek [4], ABMapper [103], STAR [34], CRAC [135], Subread [99], and OLego [176]. Some of them use a more refined technique, called *multi-seed*, that employs more than one seed for each read. Tools based on this idea are described at the end of this section.

The commonly used technique is to identify substrings of a read, called seeds, and to map them independently against the genome. The alignment is then extended to find splicing junctions. Multiple alignments are handled in different ways, and the position a read is associated to is defined either by selecting the best seed or by considering features that can help in finding the best alignment depending also on the location of other reads. Depending on the algorithm used, seeds are used to index either the set of reads or the genome, which are stored in a hash table or in a suffix array.

MapNext is based on the observation that, if a read is aligned with no more than $k$ mismatches, then by dividing it into $k+1$ seeds, at least one of them must be aligned without errors. Each seed and is stored in a hash table, each seed entry being provided with the list of reads it comes from and the position within them. The algorithm works as follows: the text (*i.e.*, the reference genome) is scanned base-by-base on seed-length sliding windows. Each time a window in the reference is considered, the corresponding sequence is searched in the hash table. Then, each read containing the current seed is checked in order to see if the alignment can be extended beyond the seed sequence. If a read turns out to be successfully mapped in the reference, it is employed to map other reads in order to create a cluster. The procedure described above is employed first in the ungapped case and in a second moment to map reads allowing gaps in the alignment. Splicing sites are detected by looking both at the annotation, if provided, and at the canonical splicing dinucleotides (*i.e.*, GT-AG).

SuperSplat indexes the reference genome with variable-length $k$-mers. More precisely, for each position in the genome, each $k$-mer of length $c \leq k \leq i$ is stored. Hence, there are $i-c+1$ $k$-mers for each position. Reads alignment is performed in the following way. Each read is searched in the reference by looking at variable-length $k$-mers both in the left tail and in the right tail of the read. More precisely, if a read has length $m$, the "left search" consists in looking at all $k$-mers belonging to the $(m-c)$-length prefix, whereas the "right search" is performed by looking at variable-length $k$-mers belonging to the $(m-c)$-length suffix of the read. All $k$-mers found are checked for intron-consistency: if the distance between the left block of $k$-mers and the right one is intron-size-consistent, then the gap is flagged as a putative read split.

GSNAP is a tool especially designed to allow for the presence of errors in alignments. In particular, it can handle multiple mismatches, long indels, and gaps due to splicing junctions. The general idea is to look at approximate mapping as the problem of aligning a pattern against a set of sequences, each of them differing from the text in terms of errors that should be tolerated. This allows to treat more easily both sequence variations and spliced alignments. Furthermore, splicing junctions prediction is performed with a probabilistic model based on an existing annotation. The mapping is done once short $k$-mers (namely, 12 bp long) are computed to index the reference genome.

SplitSeek is aimed at predicting splicing junctions only. The algorithm works by first identifying, for each input read, two non-overlapping "anchors". Anchors are aligned separately against the genome and their mapping is then extended. Each split aligned read is stored and represents an evidence for a splice junction. Reads for which some anchors cannot be aligned may be due to the fact that a splice junctions falls within the unmapped anchor. In this case, putative junctions found at previous step are scanned to see if the anchor overlaps any of them. Finally, junctions are grouped together and splicing points are refined. Splicing junctions are distinguished by indels and reported in different outputs.

STAR is a mapping algorithm that, differently from most methods for splice mapping, aligns split reads direclty to the reference genome. This means that a preliminary ungapped alignment phase, based on either whole reads sequences, as for exon-first aligners, or seed extracted from the reads, as for seed-and-extend mapping methods, is not needed. The alignment procedure is based on the so-called Maximal Mappable Prefix (MMP). The algorithm finds, for each read and for each position in the read, the maximal substring that exactly matches a substring in the reference. Once an MMP is found, the algorithm tries to extend it to find an alignment for the whole read. If this is not possible, then the MMP is used as "anchor" and the alignment is determined by allowing mismatches and indels. Anchors are selected in order to generate a minimum number of loci that support the alignment. In this phase, paired read information is employed to select pair-consistent alignments.

### Multiseed methods

ABMapper uses a suffix array to store all $k$-mers of the reference sequences. For each read $r$, two seeds are selected, the first one ($A$) at distance $k$ from the beginning of $r$, the second one ($B$) at distance $k$ from the end of $r$. $A$ and $B$ are aligned against the genome and then the alignment is extended. In the extension phase, two scenarios can happen: (i) extensions encounter each other, or (ii) extensions starting from $A$ and $B$ eventually stop as they cannot be merged to form a contiguous alignment. In the latter case, only "good" alignments are considered, that is, those covering the entire read sequence. At the end, spliced alignments are refined around the stop points by looking for the presence of canonical (GT-AG) and semi-canonical (GC-AG, AT-AC) splicing junctions.

CRAC assumes that the input (*i.e.*, the reference genome and the reads) satisfies two properties. The first one says that, given a genome $G$, it is possible to determine a value for $k$ such that on average the probability of that a sequence of length $k$ is found in $G$ once is high. The second one says that, because sequencing errors involve a single read whereas structural variants involve the genomic sequence, reads overlapping in the genome should differ in terms of sequencing errors, and, in contrast, they should differ in the same way with respect to the genome but they should be equal to each other in correspondence of structural variations. The algorithm infers mapping by basically looking at $k$-mers profile. Location profile is the frequency of a $k$-mer in the genome, whereas the support profile is the frequency of a $k$-mer in the reads. These two profiles help in determining the "type of difference" between genome and reads, and hence to defining the correct read location.

Subread propose the *seed-and-vote* paradigm to efficiently report contiguous and gapped alignments. The idea is to first divide each read into overlapping and evenly distributed seeds, called *subreads.* subreads are aligned separately against the genome and, instead of considering only the best seed, each subread is considered and its contribution to read alignment is expressed in term of "votes". Before performing alignment, the reference genome is indexed by short $k$-mers, which are stored in a hash table. For each read, only specific subreads are considered (*i.e.*, high repetitive ones are discarded) and they are aligned without mismatches. Then, each aligned subreads votes for some positions in the genome. A set of subreads voting for the same positions is called consensus set. The read is aligned against the position having the larger consensus set. The determination of the best consensus set is dependent on threshold values. In case of spliced reads, multiple consensus sets can be selected. In particular, a maximum of two positions in the genome can be selected for each read. Lastly, reads are aligned entirely by extending subreads and junctions are filtered to eliminate those supported by few reads.

OLego, analogously to MapSplice, uses a multiple seed-and-extend approach. The alignment algorithm is similar to that implemented in BWA [92]. The approach consists in first aligning reads contiguously. For those resulting to be unaligned, substrings called seeds are selected. Seeds do not overlap each other and they are evenly distributed throughout the read. Then, seeds are aligned and particular care is devoted to seeds that are unmapped but their neighbouring seeds in the read are mapped instead. Such a situation is a signal of a putative small exon. Splicing junctions are predicted based on a regression model.

## 5.3 Transcriptome assembly and gene expression estimation

Reconstructing the transcriptome means retrieving the complete collection of RNA molecules produced by a cell, or by a population of cells, in a specific condition. The transcriptome is constituted by highly fragmented molecules and varies in time and space, differently from the genome, which is a single molecule, or a set of few long molecules, that remain more or less the same during cell life (up to mutations). For these reasons, the *de novo* transcriptome assembly problem, that is, the task of reconstructing the complete transcriptome of a cell without a reference genome, is a completely different problem with respect to *de novo* genome assembly.

The nature of RNA pose several challenges to transcriptome assembly, concerning both sequence structure (*e.g.*, alternative transcripts) and dataset characteristics (*e.g.*, amount and distribution of RNA-Seq reads). Classical problems related to the assembly of the transcriptome are the detection of alternative isoforms and expression level quantification. The consequence is that, while working with transcriptomic data, no coverage information is available and it has to be inferred from the data.

A method addressing transcriptome assembly should be able to identify the different isoforms of the same gene, for organisms in which alternative splicing is allowed. In practice, instead of selecting the most sensitive path when a bifurcation in the assembly occurs, the correct output should contain both of them, that is, a transcriptome assembly tool should be able to assemble alternative transcripts separately and entirely.

Another problem hindering transcriptome assembly is the presence of repeated sequences in the form of paralogs, *i.e.*, genes with similar sequences, but, having different genomic locations, with different sets of transcripts. This scenario is similar to the detection of repeats in genome assembly, with the further difficulty of not having any coverage information to disambiguate repeated regions.

Different expression levels and alternative splicing patterns make assembly very complex, as the amount of reads on a specific region can be due to either expression level or PCR artifacts and overlapping genes can be erroneously clustered together whereas a correct transcript reconstruction would require them to be assembled separately. When working with NGS data, one should keep in mind that RNA-Seq process is characterized by many sources of bias (see Section 1.2.3).

In the last few years, transcriptomic data analyses attracted much attention. The available tools for transcriptome assembly with RNA-Seq reads seem to leave significant room for improvements because, in practice, they are not able to solve the problem in a satisfactory way [148]. Probably due to the well-known difficulty of correctly and completely assemble a reliable reference genome from NGS reads, or simply because sometimes the reconstruction of the whole genome is not of primary interest, several tools for *de novo* transcriptome assembly started to appear since less than three years ago. Clearly, if neither a reference genome nor a reference transcriptome are available, the only way to study RNA is to assemble the transcriptome *de novo*. In this case no information on coverage can be used, but normalized RNA-Seq libraries exists in order to ease assembly task. Assembling RNA-Seq reads into complete transcripts, however, may aid in addressing the problem of recognizing overlapping genes. For instance, the situation in which a gene is completely contained into the intron of another gene cannot be recognized by aligning reads against the genome directly.

*De novo* assemblers have been often employed for the reconstruction of the transcriptome when a reference genome was not available, such as for *Saccharomyces pombe* [45,182] and whitefly [45], or to improve the existing annotation, as in the case of *Drosophila melanogaster* [182], *Camelia sinensis* [182], and *Aedes aegypti* [159]. Reference-guided methods have been recently proposed as well, finding application especially on the analysis of the transcriptome and of gene expression for organisms whose genome is already well-studied (*e.g.*, human [97] and mouse [48, 97, 165]) and, possibly, a draft annotation is also available.

Tools for transcriptome assembly, especially those requiring a reference genome, are often provided with a method for computing the expression levels of the assembled isoforms. A different task consists in the so-called *differential expression*. Input for such problem are two (or more) RNA samples for which finding the subset of genes (or gene isoforms) which expression level is *significantly* different between the two samples is of interest. The above statement means that a statistical method is needed in order to assess that the difference in expression is actually representing a biological change, taking into accout various sources of errors, related both to the tool or pipeline employed for transcripts quantification (*e.g.*, incorrect RNA-Seq reads mapping, erroneous assignment of reads to their correct isoforms) and to the biases introduced by the cDNA library preparation and sequencing (*e.g.*, uneven transcripts coverage, overrepresentation of GC-rich regions during PCR amplification, sequencing errors, reverse transcription biases). Among methods for differential expression analysis we cite edgeR [141], DESeq [5], and Cuffdiff [163].

In Section 5.3.1 and 5.3.2 we describe the algorithms commonly adopted and we provide an overview of some of the available assembly tools for RNA-Seq data.

### 5.3.1 Reference-guided assembly algorithms

It is important to say that, if transcriptome assembly is done either to obtain a complete catalog of genes or, instead, to retrieve the complete transcriptome to understand which genes are expressed and which ones are not, there is still some work to be done after assembly. First, transcripts should be mapped against a reference in order to detect the genes they come from. Second, a tool for transcriptome assembly should output both the sequences and the expression level, for each gene (or isoform). These two tasks are addressed using accurate mapping methods, so that reads are hopefully assigned to the correct isoform, and allowing gaps in the alignment to enable (alternative) splicing sites identification (see [14, 164]).

A way to avoid the post-processing phase described above is to perform assembly and reads mapping at the same time, using a reference genome, when available. In this scenario, reads extracted from the transcriptome can be aligned against the reference, allowing large gaps in the alignment; then, the mapping can be used to retrieve the assembly. Examples of tools based on this strategy are Cufflinks [165], Scripture [49], IsoLasso [97], and Traph [162]. We must cite ERANGE [118] as well, for the importance of the work by Mortazavi and colleagues in the field of RNA-Seq data analysis; however the approach they use, requiring substantial information from an existing annotation, differs from the one employed by the above mentioned tools. A reference-based annotation-independent tool for transcriptome reconstruction and expression estimation usually (i) maps the reads against the genome with an alignment software (usually Tophat [164]), (ii) retrieves exons location employing information coming from gapped alignment, and then (iii) selects a set of transcripts for each predicted locus, possibly provided along with the expression level.

Cufflinks [165] was proposed by Trapnell and colleagues. Crucial for this method is the concept of *fragment*: a fragment is a location of a read or an insert (identified by the two mates of a paired read) within the genome. The first step consists in mapping the reads against the genome using TopHat, which allows fragments identification. The second step consists in detecting sets of compatible fragments, each of them ideally covering a single transcript. Then, redundant fragments are eliminated and a graph is built with those that are still available. Finally, a minimum set of transcripts is computed by finding a minimum path cover in the graph. A statistical method, based on maximum likelihood estimation model, was developed, aimed at inferring transcripts abundancies.

Scripture [49] maps the reads against the genome with TopHat and uses information from both contiguous and gapped alignments to build a *connectivity graph*. The nodes of a connectivity graph are the positions (*i.e.*, bases) in the reference genome, whereas the edges are the connections between contiguous bases either within the genome sequence or within a spliced read. In this way, both genomic and transcriptomic information is represented. The subsequent step consists in weighting all connections, which is done by computing reads alignment likelihood within windows sliding troughout the genome. The third step consists in merging the exons found at previous step into transcripts, which are subsequently refined at the final stage by incorporating paired reads information.

IsoLasso [97] follows the approach used in [49] to find the *candidate* isoforms, and then uses a statistical framework to infer which isoforms are expressed and which are not, and quantifies their expression level too. The difference of the approach by Li and colleagues with respect to previously described methods consists in the usage of a constrained least square problem. The model is based on an $n \times m$ matrix for each gene, where $n$ is the number of candidate isoforms and $m$ is the number of exons, and on a vector of $n$ variables $x$, each component representing the expression level of the respective isoforms. The optimization problem is defined in such a way that the solution $x$ should meet the following objectives: (i) the error between the estimated expression levels and the observed coverage is minimal and (ii) the number of expressed isoforms per gene is also minimal. A problem with this method in particular, which has important shortcomings on other reference-guided assembly methods as well, is represented in the bias introduced by uncertain reads mapping (especially for short sequences) and by uneven transcripts coverage. A method strongly based on exons coverage may suffer of such events, but least square models can be customized by either adding constraints (*e.g.*, evidence for connections between exons provided by gapped reads alignment) and/or properly weighting exons according to the "degree of confidence" of coverage estimation (*e.g.*, depending either on exon length or on its location within the gene).

Traph [162] implements an algorithm for finding the min-cost flow on a graph. It works without any knowledge on the transcriptome and the procedure is divided in two steps: exons detection and transcripts annotation. The advantage of using a network flow method lies on the fact that the computation of expression levels is embedded in the effort for finding expressed transcripts, which are viewed as paths in the graph traversed by some flow. The determination of the exon graph is done using TopHat, which allows to find the splicing sites thanks to gapped reads alignment. TopHat is employed also to compute the average exons coverage. Once exons are found, and their coverage has been computed, a labelled directed graph $G = (V, E, w, v)$ is built for each gene. Each node in $V$ corresponds to an exon and is assigned a weight representing the average exons coverages. Each edge in $E$ corresponds to the adjacency relation between two exons and is assigned a weight equal to the number of reads spanning the junction. A graph built in this way is acyclic and the computation of a min-cost flow on $G$, where the cost is represented by the error of expression estimation with respect to coverage data, can be computed in polynomial time. The authors prove that a partition in paths exploiting the flow is also computable in polynomial time. The paths reported, provided by the units of flow associated to them, represent the transcripts and their expression level returned in output.

It is fair to say that, even though a reference-guided transcriptome assembly algorithm can benefit from genomic information, at the same time it may hinder the detection of either gene fusion events (*i.e.*, exons belonging from different genes that are put together during transcription) or some alternative splicing pattens such as intron retention (*i.e.*, introns that are not removed during the splicing process and thus are still present in mature RNA, see also Figure 1.4). Again, a *de novo* approach is the only feasible stategy to address the transcriptome assembly of organisms for which the reference genome and/or the annotation are either partial or absent.

### 5.3.2 *De novo* assembly algorithms

Heber and colleagues [50] tried to formalize the reference-independent transcriptome assembly problem. They proposed a mathematical model similar to the de Brujin problem (see Section ), with the difference that the solution is a graph, rather than a path. The authors work in the hypothesis of EST input data (*i.e.*, transcriptomic reads sequenced with traditional technologies), but their reasoning applies to RNA-Seq as well. They noticed that algorithms for *de novo* genome assembly are not suitable for EST datasets: the reason is that such algorithms perform reads clustering, with the aim of maximizing output contiguity. Such an approach is not suitable for transcriptome assembly, because (i) the transcriptome is not a single sequence and (ii) overlapping transcripts should not be clustered together (in fact, they may constitute an evidence for alternative splicing patterns). Following the approach of Heber and colleagues, an algorithm for *de novo* transcriptome assembly takes as input a de Brujin graph built from EST sequences and extracts the *splicing graph* described in Definition 5.3.1.

**Definition 5.3.1 (Splicing graph)** *Let $T = \{t_1, \dots, t_n\}$ be the set of all transcripts for a gene $g$ and let $V_i$ be the set of genomic positions associated to $t_i$, for each $i = 1, \dots, n$. The splicing graph of $T$ is the directed graph $G = (V, E)$ such that $V = \bigcup_{i=1}^{n} V_i$ and $(v, w) \in E$ if and only if $v$ and $w$ are contiguous genomics positions for some $t_i$.*

The idea is that, through the splicing graph, one could retrieve all the splicing patterns of the set of transcripts. After the graph has been built, the problem of identifying paths associated to the actually expressed isoforms should be addressed. Hence, the Transcript Assembly Problem can be defined as follows.

**Definition 5.3.2 (Transcript assembly problem)** *Let $\Sigma = \{A, C, G, U\}$, $R \subseteq \Sigma^*$ be a finite set of strings extracted from an unknown set $T \subseteq \Sigma^*$, and $G$ be the splicing graph associated to $T$. Find $T$.*

Given the splicing graph $G$ for a gene $g$, the transcript assembly problem is solved once the paths associated to expressed isoforms are detected. The task of obtaining $G$ starting from the set of reads $R$, also called Graph Reconstruction Problem, can be addressed using a de Brujin graph built with the $k$-mers occurring in the reads. The solution of this problem consists in finding a set of paths covering all the edges of the de Brujin graph, similarly to the Eulerian path approach. In order to produce a set of transcripts consistent with the data, information from genomic sequence, mRNA, ESTs, and proteins, may be employed [178]. The solution of the problem will be a minimal set of transcripts that is consistent with the observations.

Approaches for *de novo* transcriptome assembly are either inherited from former techniques for genome assembly or built from scratch to tackle the intrinsic different nature of this problem. Similarly to *de novo* genome assembly tools, algorithms for 454 reads are mostly based on the OLC paradigm, whereas RNA-Seq reads produced by Illumina or SOLiD platforms are commonly assembled with software tools based on de Brujin graphs.

Most of OLC-based assemblers commonly used for transcriptomic data [86] are some of those that have already been designed for genomic data, such as, MIRA, its new version miraEST [25], CAP3, and SOAPdenovo. TGICL [132], instead, has been developed *ad hoc* for transcriptome assembly. More precisely, TGICL is a *local* assembler that clusters reads using a graph in which each node is a sequence and edges represent alignments; its output is then used to feed an external tool (*i.e.*, CAP3) in order to obtain the final assembly.

The de Brujin graph approach for transcriptome assembly differs from the classical algorithm designed for genome assembly. In the latter case, the "ideal" $k$-mer length is selected depending on the dataset used, expecially on coverage, and it is usually possible to set a value for $k$ that lead to optimal results. In the context of transcriptome assembly, instead, different values of $k$ allow to reconstruct transcripts at different ranges of expression levels. More precisely, large values of $k$ allow to *specifically* assemble highly expressed transcripts, whereas small values of $k$ are more suitable to *sensitively* capture lowly expressed transcripts. For these reasons, an optimal strategy, actually adopted by most assembly tools, is to vary the value of $k$ within the pipeline [182]. Some de Brujin graph-based assemblers have been developed on the basis of existing tools for *de novo* genome assembly, for instance, Oases [150] (*i.e.*, the adaptation of Velvet to handle RNA-Seq reads) and Trans-ABySS [12] (*i.e.*, the version of ABySS for transcriptomic data). The implementation of Oases and Trans-ABySS substantially changed with respect to the genome assembly algorithms they grow out of; in the following, we spend some more words to describe these approaches.

Oases [150] performs different assemblies for different values of the hash word length $k$, and the outputs are finally merged. For each fixed value of $k$, the first two phases of Velvet are used: the hash and the graph construction. Then, a draft assembly is built by visiting the graph without solving ambiguities. At this step a set of short contigs is produced. Heuristics performed by Velvet to solve ambiguities in the assembly cannot be used in this context, because uniform read coverage is a required assumption. In Oases the graph is clustered and trimmed in the following way: if two paths start and end in the same nodes, and the sequences associated to them are similar to each other, then the two paths are clustered together to build a single path, taking the highest coverage path as a consensus; very low covered edges are deleted, as well as those having much lower coverage with respect to the other edges exiting from the same node. Then, the contigs obtained after the merging step are connected using both reads and paired reads information. In the latter case, insert size variation is taken into account. Every connection is associated a weight, that is higher for *direct* (read-based) connections and lower for *indirect* (paired read-based) ones. Connections are discarded if the corresponding weight is too low (namely, lower than the weight assigned to a single read connection). Finally, putative splicing events are identified, looking at the topology of the graph. Some topologies are easy to be solved exactly, whereas for complex ones an heuristic algorithm is used. The procedure described above is repeated for all $k$-mer values and the transcripts built are stored. All the transcripts obtained, after redundancy elimination, are used to build a graph with parameter $k_{merge}$. The assembly built on this final graph is the set of transcripts outputted by Oases.

There are also several pipelines that use Velvet as a subroutine, possibly launched several times with different $k$-mer values, and employ some heuristics to obtain a more accurate assembly, with respect to the result obtainable with a single run, with fixed $k$. Among these approaches we cite Rnnotator [111] and Multiple-$k$ [159].

Trans-ABySS [12] is the transcriptome assembly version of ABySS. This tool allows diversification in the assembly algorithm in order to capture different expression levels. This is done by requiring variable amounts of reads supporting paths in the de Brujin graph.

A tool that was designed *ex novo* for the transcriptome assembly problem is Trinity [45]. The approach followed by Trinity envisages the construction of a de Brujin graph using a fixed value of $k$. Being strongly based on $k$-mer frequencies, Trinity is well-suited for the reconstruction of highly expressed transcripts. The tool is composed by three modules: Inchworm, Chrysalis, and Butterfly. Inchworm creates a $k$-mer dictionary from the reads. First, it excludes low-frequency and low-complexity (*i.e.*, tandem) $k$-mers; second, it selects seed $k$-mers iteratively, starting from the most frequent ones; third, every $k$-mer seed is extended on both $5'-3'$ and $3'-5'$ directions. All the $k$-mers used are then deleted from the dictionary and the procedure goes on until the dictionary is empty. Then, Inchworm clusters $k$-mers as long as there are neither ambiguities, nor available overlapping $k$-mers. This is equivalent to finding linear paths on de Brujin graphs built with $k$-mers. The output of Inchworm is a set of *linear* contigs. Chrysalis, starting from the linear contigs generated by Inchworm, uses $k$-mers information to group together overlapping contigs. For every maximal group of overlapping contigs, a de Brujin graph is built. Then, Chrysalis uses reads information to validate contigs, *i.e.*, a read is assigned to the contig sharing the highest number of $k$-mers. Finally, Butterfly compresses the graph by storing each maximal linear path within a single node, and deletes short paths (*i.e.*, spurs). The graph obtained is a *sequence graph*. Every edge is assigned a weight corresponding to the number of reads supporting it, *i.e.*, the number of reads sharing the highest number of $k$-mers with the contig corresponding to the edge. A dynamic programming-based procedure is then used to identify the paths that are best supported by the reads, requiring contiguity and using paired reads information. These paths are the assembled transcripts returned in output.

## 5.4 Evaluation of methods for RNA-Seq data analysis

When a transcriptome assembly project starts, one of two major objectives is usually addressed: (i) quantify the expression level of transcripts and (ii) obtain a complete catalog of all the genes of an organism. Problem (i) is hindered by the presence of paralogous genes because, even being able to discern among very similar transcripts instead of clustering them together, assigning the correct expression level to each of them (see Equation 1.2.2) is a hard task. Problem (ii) is nowadays unsolved, in the sense that there are very few organisms for which the set of all genes is completely known. Moreover, all the different functions of a cell cannot be explained by gene complexity. Alternative splicing is the key in this scenario, allowing for the production of different transcripts from the same gene (see also Section 1.1.3). Hence, gene prediction problem is extremely difficult even when a high-quality reference genome is available.

Several issues have to be taken into account when judging tools for RNA-Seq data analysis, but literature still lacks of standardized methods for evaluation when the ground truth is unknown. Some attempts have been made towards transcriptome assembly evaluation [16,148], but they cannot be considered as standardized approaches to this problem. This point will be discussed in Section 5.4.1

Extensive evaluation of both mapping and assembly tools on simulated data has recently been addressed by RGASP competition [69], and some simulation pipelines have been recently proposed as well. We review some features of RNA-Seq libraries simulation in Section 5.4.2 and describe the approaches towards tools comparison in Section 5.4.3.

### 5.4.1 Draft approaches to transcriptome assembly evaluation

Among the problems related to transcripts assembly and genes identification, a fundamental task to be addressed is results validation. Different approaches should be used instead of standard contiguity measures (*e.g.*, N50, mean contig length, and total contig length), for the following reasons: (i) the transcriptome is highly fragmented, (ii) alternative transcripts, or transcripts originated from paralogs, must be kept separated, and (iii) transcriptome length is unknown and impossible to estimate.

For both genome and transcriptome assembly, an important aspect is sequence correctness. If a reference genome is available, then a partial indication of assembly correctness can be achieved by aligning the transcripts against the DNA sequence, possibly allowing large gaps in order to account for introns. The availability of a complete annotation, *i.e.*, the location of each gene within the genome provided with the set of feasible isoforms, allows for the computation of more accurate statistics, although the question of whether a transcript correctly mapped against the reference transcriptome is actually expressed in the dataset or not remains open. If neither a reference genome nor a reference transcriptome are available, assembly evaluation is much more difficult. A possible way to assess correctness consists in aligning single or paired reads against the assembly. Uniform read coverage throughout the transcript can be seen as a positive signal towards sequence accuracy. However, even though paired reads mapping can help evaluating genome assembly, insert-size information may not be particularly informative when trying to unveil the structure of alternatively spliced genes.

Few attempts, although not representing a standard for evaluation, have been proposed by Bräutigam *et al.* [16] and by Schliesky *et al.* [148]. The former work is devoted to *assemblers* evaluation: the authors simulate reads libraries with 0, 3, and 5 sequencing errors per reads, respectively, and run several assemblers using the simulated reads as input. However, they computed poor statistics, namely, few contiguity measures (*e.g.*, N50), the percentage of uncovered transcriptome, and the number of hybrids (that is, incorrectly reported transcripts). Simulated data were generated from *Arabidopsis thaliana*, for which the transcriptome is well-studied and annotated. The aim of the work by Bräutigam and colleagues was not to design a standard evaluation method, but instead to provide an indication of which assembler was best suited for the assembly of a transcriptome of interest based on the knowledge of a closely related species.

The work proposed by Schliesky and colleagues, instead, is aimed at *assemblies* evaluation and represents a fair attempt to both standardize the validation pipeline and provide a public set of *reference-independent* evaluation scripts. In this work a database of annotated transcriptomes is needed in order to compute the following statistics: (i) number of unigenes (*i.e.*, contigs) that match the reference, (ii) number of hybrids (misassemblies and read-through), and (iii) consistency of assembled unigenes with the reads. Also, reads coverage of assembled transcripts helps to estimate correctness (*e.g.*, uneven read coverage and regions supported by few reads, may suggest that misassemblies occurred). The merit of the work by Schliesky and colleagues lies on the fact that they both provide a clear and public evaluation pipeline and employ real data, resulting in a realistic method for transcriptome assembly projects evaluation.

### 5.4.2 Simulation pipelines for RNA-Seq experiments

In order to evaluate the ability of software tools to correctly predict splicing junctions in real scenarios, including reverse transcription and alternative splicing events, sequencing errors, polymorphism, and indels, some RNA-Seq simulation pipelines have been developed. Among available tools we cite flux simulator [53] and BEERS [46].

Flux simulator is mainly devoted to the customization of the RNA-Seq library generation, including several parameters that allow simulation of template switching or antisense transcription. It also allows to customize expression levels variation among alternative isoforms. For these reasons, it can be used to test assembly tools in presence of complex splicing patterns.

BEERS was developed to specifically address the evaluation of the performances of spliced reads alignment tools in presence of problematic regifons. Concerning this last point, to be as much realistic as possible, BEERS generates a synthetic transcriptome by extracting random genes from public annotation repositories and generates alternative isoforms by randomly excluding exons. The useful feature of BEERS in terms of alignment evaluation consists in the careful labelling of “wrong” predictions in terms of indels location (*e.g.*, monomers of different lengths) and in the effort of levelling differences among tools’ outputs.

The RNA-Seq unified mapper (RUM), developed by the same authors of BEERS, is a pipeline for RNA-Seq reads alignment based on Bowtie and BLAT. The element of novelty of RUM, with respect to other alignment methods, lies on the fact that multiple alignment positions are disambiguated using information from both the transcriptome and paired reads. BEERS detects splicing junctions by checking for 11 different dinucleotide pairs, which are called *characterized splice signals* by the authors.

### 5.4.3 Comparison of tools for RNA-Seq

Due to the large amount of tools available for RNA-Seq reads assembly and mapping, the need of a standardized evaluation of aligners performances on real data gave birth to the RGASP [69] competition, started about one year ago. The target of the competition was to analyze the capabilities of transcriptome assemblers and spliced aligners with NGS reads (*i.e.*, Illumina) sequenced from mammalian transcriptomes (*i.e.*, human and mouse). The importance of this project lie on the fact that the data employed, both simulated and real, are publicly available, as well as the evaluation scripts used to compute evaluation statistics.

Evaluation performed by Steijger and colleagues [158] allowed to rather fairly evaluate the performances of tools for transcripts reconstruction and quantification with RNA-Seq, even though the competition was quite unbalanced towards reference-based assembly tools.

Engstrom and colleagues [40] evaluated the results in terms of alignment yield, reads mapping accuracy, and splicing sites detection precision. The main conclusions of the study are concerned with the fact that (i) the performances of the same tool do not sensibly change between simulated and real data, nor with the availability or not of an annotation, (ii) the false positives rate of splicing sites prediction is quite high for all tools, and (iii) high mapping accuracy is reachable, but often at the price of low sensitivity (mainly depending on parameters strictness and/or fixed values for the maximum number of mismatches and indels length). The above statement suggests that there is still some room for improvement for spliced mapping tools.

The main problem with exon-first spliced aligners is that they often map reads without gaps first, and in a second step they perform gapped alignment with reads so far unmapped. This can lead to an unbalance towards ungapped alignment in spite of gapped one, resulting in a possible under-representation of splicing junctions. Concerning seed-and-extend methods, instead, seed length must be constrained to be greater than a minimum value in order to guarantee alignment "safety". This limitation could cause a bias in splicing sites coverage estimation, or even prevent their detection, especially if they are spanned by low expressed transcripts.

Very few tools take into account the issues mentioned above, by either using an existing annotation (*e.g.*, QPALMA), looking at the $k$-mer profile (*e.g.*, CRAC), or collecting information on the other reads' mapping (*e.g.*, Subread). However, transcriptomic information is not always available and, moreover, it can prevent novel splicing sites detection. $k$-mers or reads mapping profile can only provide global information, that is, can guide a mapping tool to an output such that exons are uniformly covered. Paired read information is used by existing tools only to discern among multiple alignments (usually for spliced reads), but, in contrast, if inserts are available they could provide more precise information on mapping. In Section 6.1 we will deepen this reasoning with a massive usage of paired reads, by accurately assemblying inserts before performing RNA-Seq reads alignment. Of course, this approach requires a careful local assembly algorithm, able to output correct in-silico sequences.

# 6 RNA-Seq reads alignment and splicing sites detection

The comprehension of the complex machinery that allows for the production of mature RNA from a set of expressed genes requires accurate isoforms annotation and expression levels estimation. The first steps for any downstream analysis lie in both precisely detecting splicing sites and accurately quantifying expressed transcripts. Hence, the potential of RNA-Seq technologies is deployed at the alignment stage, and reads mapping constitutes the basis on which almost all analyses are carried out. For this reason, in the last few years, several tools especially addressing high-throughput RNA-Seq reads alignment have been proposed [100].

In this chapter we introduce a new method that specifically addresses accurate junctions prediction and reads mapping sensitivity. The idea is to overcome the limitations of exon-first and seed-and-extend alignment method, as described in Section 5.4. To the best of our knowledge, there does not exist a spliced aligner that explicitly addresses the problem of optimally selecting among ungapped and gapped alignments for the same read. Exon-first methods are intrinsically incapable to prefer a gapped alignment in spite of a contiguous one, as most tools usually align reads in two stages, mapping firstly without gaps and then reconsidering only unaligned reads for spliced alignment. Seed-and-extend methods are, in fact, most suitable for this problem and may hope to be more sensitive with splicing junctions prediction.

Our approach is based on the idea that a tool or a pipeline for transcriptomic reads alignment should meet the following objectives: (i) safely map short reads fragments around splicing sites, (ii) improve as much as possible the amount of uniquely mapped reads, and (iii) depend as little as possible on annotation for new splicing events detection. We attempt to meet these three objectives by firstly assembling inserts using an RNA-Seq paired read library and subsequently "guiding" reads mapping against the genome, using the collected information on reconstructed inserts.

## 6.1 An insert-guided alignment technique

Having a method able to *guide* alignment using correct and longer in-silico built sequences is an important issue for addressing correct location of RNA-Seq paired reads on the refererence genome. To this aim, a whole transcriptome assembly is not needed and, in fact, this is not even the best solution. The methods usually employed for genome as well as for transcriptome assembly adopt heuristics that put together sequences in order to

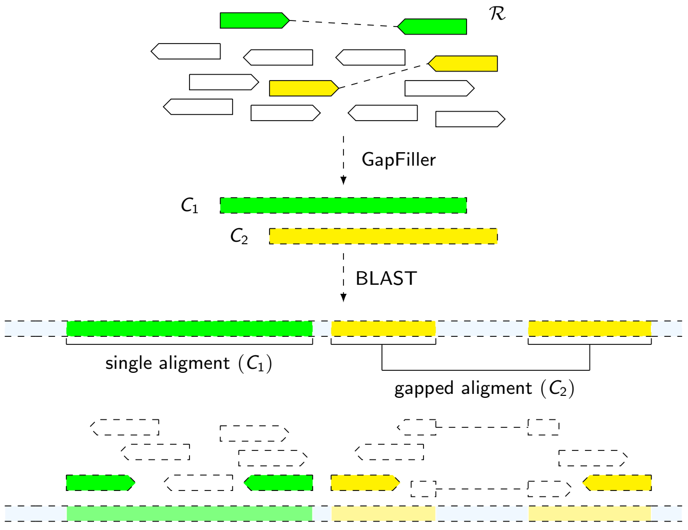


**Figure 6.1:** The three-steps pipeline for RNA-Seq reads alignment. All inserts from an RNA-Seq paired reads library are assembled with GapFiller. Using a specific GapFiller option, the layout is stored for each insert (*i.e.*, ID, position, length, and strand of each read used for assembly). Inserts are then aligned with BLAST against the reference genome and the best hits are selected. Gapped alignments are also refined around the gap by looking for the presence of splicing dinucleotides. Reads mapping is retrieved from inserts alignment and GapFiller layout. In case of multiple alignment, paired read information is used to select the best read location.

obtain higher connectivity, often causing a drop in sequence correctness. We, therefore, propose a method that builds *slightly* longer—with respect to NGS reads—sequences, exhibiting a high level of accuracy. Our aim is to improve downstream RNA-Seq reads mapping, using information coming from the alignment of assembled in-silico sequences encompassing them.

The idea of firstly assemblying NGS paired reads into longer sequences as a preprocessing step has been shown to be advantageous in various contexts, especially for *de novo* assembly purposes [183]. Using a local reads assembly method within an alignment pipeline is a new idea that may be particularly useful for RNA-Seq input data because of the large number of gaps introduces, which hinder mapping accuracy.

Our pipeline goes through with the following three-steps: (1) the *assembly* of paired reads into insert-contigs, (2) the *mapping* of the inserts-contigs against the reference genome, and finally (3) the *retrieval* of RNA-Seq reads position in the reference by insert-contigs' layout and alignment (see also Figure 6.1).

Step (1) is done with GapFiller (see also Chapter 3). GapFiller takes as input a paired reads library and outputs a set of insert-size contigs. GapFiller output correctness has been extensively tested and aligning such longer and reliable sequences, instead of RNA-Seq reads themselves, significantly improves the overall quality of the pipeline. Step (2) is performed using BLAST and, if multiple hits are found for a single contig, they might be selected and refined. Hits selection is done with a greedy algorithm aimed at finding a set of hits covering the highest percentage of the contigs sequence. Then, if short overlaps occur among selected hits, donor-acceptor dinucleotide pairs are searched within the reference sequence and the alignment is trimmed accordingly. Step (3) requires information both from (1) and (2). During step (1), the position of the reads within the assembled contigs (*i.e.*, the *layout*) is stored. Hence, for each read, the location within the contigs is combined to the coordinates of the contigs' alignment against the genome. This allows to derive the mapping locations of each read from contigs' alignment and layout information.

Our alignment pipeline first assembles inserts, then aligns them against the reference, and finally goes back to the data to retrieve reads alignments. Hence, our method requires both (i) a good quality reference genome and (ii) an RNA-Seq paired reads library. The reasons why we chose this three-steps computation is the following. On the one hand, direct RNA-Seq reads alignment against the reference genome may be inaccurate due to small query length. On the other hand, RNA-Seq data analyses often require to quantify exons coverages, which cannot be computed directly from sequences produced in-silico. The assembly technique used by our central module, GapFiller, allows to easily store the layout of each contig, resulting in the possibility of either (i) keeping only the reads belonging to correctly assembled inserts, and, once contigs are aligned against the reference, (ii) eliminating redundant coverage by retaining only one copy of a read at each alignment position.

### 6.1.1 Inserts assembly and layout

GapFiller [123] is a tool designed to close the gap between the two mates of a paired read. See Section 3.2 for details on the algorithm. GapFiller is conceived to work with Illumina data and should behave better on reads whose sequence quality is higher on positions close to $5'$ read tail (see also Section 3.3.2). However, it can be used to assemble different reads of different types and lengths. Every contig outputted by GapFiller is classified into *trusted*, if the mate of the seed read has been found, *non-trusted*, otherwise. Only a trusted contig is a sequence that we consider correct and hence can be used for further analyses. In the following, such a sequence will be called *insert-contig*. GapFiller insert-contigs have been shown to be highly accurate at various coverage depths and insert lengths, both with simulated and real NGS genomic data [123].

The criteria used to extend the consensus sequence are based on relative abundances computed on-line and are not sensitive to even significant changes in coverage depth. Moreover, GapFiller allows to simply retrieve the layout of the reads actually employed during the assembly. These two characteristics turn out to be, theoretically and practically, very useful in dealing with an assembly-guided alignment method for RNA-Seq data. It is fair to say that, however, GapFiller performances are supposed to suffer from extremely low reads coverages (*i.e.*, very low expression levels). In such cases direct RNA-Seq reads mapping is preferable to any assembly-guided alignment technique and can always be used in addition to our pipeline on the subset of reads that result to be unmapped after the pipeline.

During assembly, the full information on the reads used to assemble every insert-contig is computed on-the-fly at each extension step. Let $\mathcal{R} \subseteq \mathbb{Z}$ be the set of the input reads (univocally identified by their IDs). For each contig $C$, we consider its layout $L_r(C)$ (see Section 4.1). We implement $L_r(C)$ as an array $L$ containing the following information for each component $L[k]$:

$L[k].id$ is the read ID;

$L[k].start$ is the starting position of the read in $C$;

$L[k].length$ is the read length;

$L[k].strand$ is the read orientation within $C$.

Field $L[k].length$ is maintained so that all information needed for alignment is stored in the layout data structure. Indeed, the read sequence is not needed, as explained in detail in Section 6.1.3. Contig layout allows to retrieve reads position from insert-contigs alignment against the genome.

### 6.1.2 Insert-contigs alignment against the reference

The presence of introns interspersing exonic sequences cause large gaps to occur while aligning insert-contigs against the reference genome. To address this issue, we use a gapped aligner for "long" sequences (*i.e.*, longer than the input reads). In our case we use BLAST [3], but any other gapped aligner would serve the purpose. BLAST provides us with a set of *hits* for each insert-contig. Given an insert-contig $C$, we proceed as follows. The first step consists in selecting a minimal set of *consistent* (*i.e.*, feasibly coming from consecutive exons) hits covering the highest possible fraction of $C$. A further step is performed whenever $C$ cannot be aligned almost entirely with a single hit (*i.e.*, the set of consistent hits contains more than one element). In this case, the hits boundaries are refined in correspondence of the positions in which splicing dinucleotides are possibly found. In the following we provide detailed description of insert-contigs alignment selection and refinement around splicing sites.

#### Hits selection

BLAST outputs multiple hits for a single contig, due either to repeats within the genome, or to splicing events. We will refer to *insert coordinates* as the positions of a hit with respect to the insert-contig, and to *reference coordinates* as the positions of a hit in the reference. Given an insert-contig $C$, the insert coordinates of the hits associated to $C$ make it possible to distinguish repeats from splicing events. Indeed, if two hits overlap for a significant region in $C$, then a sensible explanation is that the hits fall into two distinct occurrences of the same repeated sequence; otherwise, the two hits are the result of the alignment of two different sequences that are not contiguous in the reference. The latter case suggests that the break in the insert-contig is probably due to a splicing event. Given an insert-contig $C$, we define $H$ as its *alignment array.* A single component in $H$ consists of the information on a single hit. Assuming $C$ has $n$ hits, for each $1 \leq i \leq n$, $H[i]$ maintains the following fields:

$H[i].\mathit{ref}$ is the reference sequence name;

$H[i].\mathit{q\_start}$ is the hit start in the insert-contig;

$H[i].\mathit{q\_end}$ is the hit end in the insert-contig;

$H[i].\mathit{t\_start}$ is the hit start in the reference;

$H[i].\mathit{t\_end}$ is the hit end in the reference.

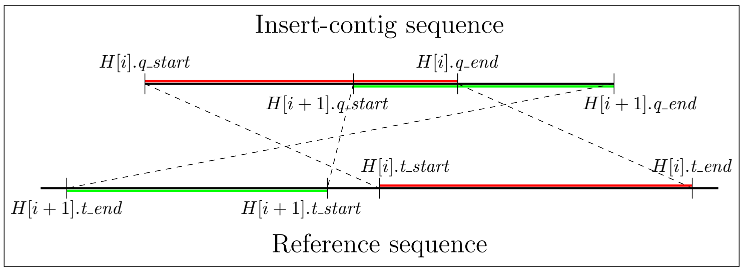
From the above information, we stored an additional boolean field $H[i].\mathit{strand}$ representing the strand of the alignment of $H[i]$ with respect to the reference. $H[i].\mathit{strand}$ is set to TRUE if and only if $H[i].\mathit{t\_start} < H[i].\mathit{t\_end}$ and to FALSE if and only if $H[i].\mathit{t\_start} > H[i].\mathit{t\_end}$ (case $H[i].\mathit{t\_start} = H[i].\mathit{t\_end}$ do not occur in practice).

To select an optimal consistent subset of $\{H[1], \ldots, H[n]\}$, we define a *consistent set of hits* to be an ordered set $\langle H[i_1], \ldots, H[i_m] \rangle$ such that

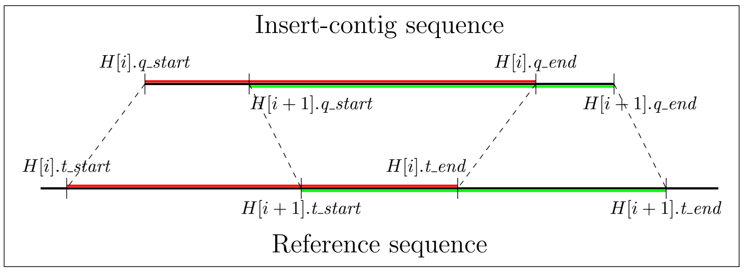
(a) $H[i_h].\mathit{ref} = H[i_k].\mathit{ref}$;

(b) $H[i_h].\mathit{strand} = H[i_k].\mathit{strand}$;

(c) $H[i_h].\mathit{q\_start} \leq H[i_{h+1}].\mathit{q\_start}$;



**(a)** Incompatible hits



**(b)** Compatible hits

**Figure 6.2:** Compatible and incompatible hits for a single insert-contig alignment.

(d) $H[i_h].t\_end + max\_intron\_size \geq H[i_{h+1}].t\_start$;

(d') $H[i_{h+1}].t\_start + max\_intron\_size \geq H[i_h].t\_end$,

where (a) and (b) hold for each $h, k = 1, \ldots, m$, whereas (c), (d), (d') hold for each $h = 1, \ldots, m-1$. Conditions (a) and (b) state that two hits are compatible only if they belong to the same reference sequence and are on the same strand. Condition (c) defines the order on the set of hits and condition (d) states that two hits cannot be farther from each other than the maximum intron size. Condition (d') is analogous to (d) in case of reverse alignments.

In practice, we chose to use an "approximate sorting" algorithm, that is, we allow for a little violation $\delta$ in the ordering:

(c') $H[i_h].q\_start \leq H[i_{h+1}].q\_start + \delta$.

Condition (c') turns out to be useful at the alignment selection step. The above conditions guarantee consistency; we say that $\langle H[i_1], \ldots, H[i_m]\rangle$ is also *optimal* if $m$ is minimal among all the consistent sets of hits that cover every position covered by the whole set $\{H[1], \ldots, H[n]\}$. In practice, we want only one hit for each region in the insert-contig, up to short overlaps. We define a parameter $\Delta$ representing the maximum overlap among consecutive hits of a minimal consistent set, so we get the following additional condition:

(e) $H[i_h].q\_end \leq H[i_{h+1}].q\_start + \Delta$,

for $h = 1, \ldots, m-1$. If previous condition does not hold, then we say that the two hits overlap. In such a case, only one must be kept, *i.e.*, the highest quality one.

Hits approximate sorting according to *q_start* is obtained using a stable algorithm (see [29]). The reason we use a stable algorithm is that BLAST output hits for non-increasing quality values, and we expect high-quality BLAST hits to be those better covering the insert-contig they refer to. More precisely, we want hits with *q_start* values differing from each other for at most $\delta$ to be reported in the same order as given by BLAST. This is done in order to keep the sequence in the reference that best fits with a subsequence of the insert-contig, whenever multiple hits with similar *q_start* values are found. Clearly, this choice influences the selection of following hits.

The procedure for selecting an optimal consistent set of hits works iteratively as follows. Every time a new hit $H[i]$ is considered, it is compared with the previous one and, if conditions (a), (b), and (e) do not hold, then the hit is neglected. Otherwise, each following hit $H[j]$ is considered: if $H[j]$ precedes $H[i]$ in the approximate ordering (with respect to condition (c')), then it is inserted immediately before $H[i]$ and the algorithm restarts from $H[j]$. At the end, an approximate ordering of non-overlapping hits (with respect to (e)) is returned.

### Alignment refinement around splicing sites

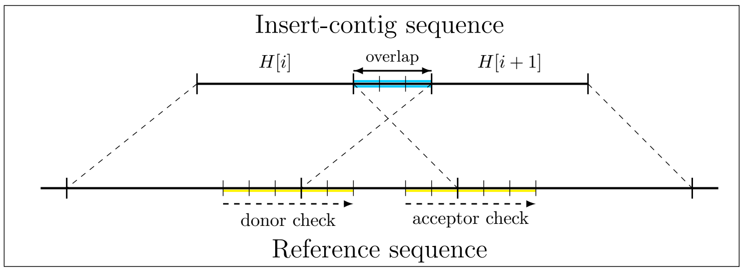
When the optimal consistent set $\langle H[i_1], \ldots, H[i_m]\rangle$ found at previous step contains more than one element, overlaps among consecutive hits should be eliminated, so that "artificial" redundancy is not introduced in the alignment. Cases to be considered at this points are those in which $H[i_j].q\_end \geq H[i_{j+1}].q\_start$, for some $j = 1, \ldots, m-1$. An overlap occurs every time an insert-contig spans a splicing junction in which either an intron prefix is identical to the right exon prefix of the same length and/or an intron suffix is identical the left exon suffix of the same length. Overlaps of few dinucleotides can happen quite often, but, having a reference genome available, the correct splicing site position can be found by checking the presence of donor/acceptor dinucleotides pairs, in particular we considere the dinucleotide pairs GT-AG, GC-AG, and AT-AC. If a *unique* couple of dinucleotides is found in the reference, then we can set the position of the intron unambiguously; otherwise, we opt for trimming the whole overlapping sequence.

More in detail, given $\langle H[i_1], \ldots, H[i_m]\rangle$, let $j$ be such that $H[i_j].q\_end > H[i_{j+1}].q\_start$, hence the length of the overlap between $H[i_j]$ and $H[i_{j+1}]$ is $l = H[i_j].q\_end - H[i_{j+1}].q\_start + 1$. Let $G$ be the reference sequence associated to $H[i_j].ref$. Donor and acceptor are searched within two regions of length $l$, respectively:

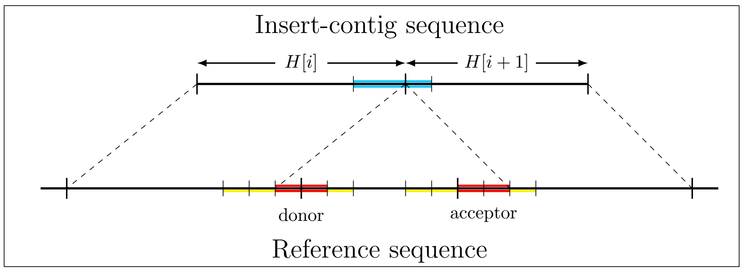
$$G[H[i_j].t\_end - l, \ldots, H[i].t\_end + 1];$$
$$G[H[i_{j+1}].t\_start - 2, \ldots, H[i_{j+1}].t\_start + l - 1].$$

This case refers to forward alignment; reverse complement case is analogous except for $i_j$ in place of $i_{j+1}$ and *t_end* in place of *t_start*, and *vice versa*. The two above regions in the reference are scanned simultaneously from left to right one position at a time, and the current dinucleotide pair is compared to the splicing dinucleotides (see also Figure 6.3). If short indels are present, reference subsequences may have different lengths and dinucleotides positions are searched accordingly.

Finally, if exactly one splicing dinucleotide pair is found, the alignment is refined in correspondence of the dinucleotides position, and the new alignment will be such that $H[i_j].q\_end + 1 = H[i_{j+1}].q\_start$. Otherwise, the whole overlapping sequence is trimmed and this will cause a gap within the insert-contig. In any case, the new hits do not overlap within the insert-contig, hence redundancy is eliminated.



**(a)** Overlapping hits in the insert-contig



**(b)** Donor-acceptor check and hits refinement

**Figure 6.3:** Redundancy elimination through splicing dinucleotides check.

### 6.1.3 RNA-Seq reads alignment from insert-contigs layout and alignment

The layout and the refined insert-contig alignments computed as above are then used to determine reads alignments. Before entering into some detail of the method, let us briefly discuss how we deal with the fact that that we may have multiple alignments for a given read, as well as multiple hits for a given alignment. Each hit can be identified by an interval whose boundaries represent the alignment coordinates. Moreover, since alignment redundancy has been eliminated, intervals are also pairwise disjoint. An alignment can be seen as a set of intervals lexicographically ordered with respect to the first coordinate. Alignments are not necessarily pairwise disjoint, but must be distinct (this will be thoroughly discussed in Subsection 6.1.3). From this point of view, mapping a read $r \in \mathcal{R}$ against the genome could result into a collection $\mathcal{A}$ of ordered sets of disjoint intervals. If $\mathcal{A} = \emptyset$, then we say that $r$ is unmapped, otherwise we say that $r$ is mapped. In the latter case, if $|\mathcal{A}| = 1$ then we say that $r$ is uniquely mapped, otherwise we say that $r$ is multiply mapped. Given $r \in \mathcal{R}$, we implemented $\mathcal{A}$ as an array $A$. We refer to *global coordinates* as the positions in the reference sequence, and to *local coordinates* as the positions in the read (reference and insert coordinates, respectively, of 6.1.2). Each component $A[j]$ contains the following fields:

$A[j].\mathit{ref}$ is the reference sequence;

$A[j].\mathit{coord}$ are the global coordinates;

$A[j].\mathit{loc\_coord}$ are the local coordinates;

$A[j].\mathit{read\_strand}$ is the strand of the read in the insert-contig;

$A[j].\mathit{contig\_strand}$ is the strand of the contig in the reference.

There is an univocal correspondence between elements of $A[j].coord$ and elements of $A[j].loc\_coord$. If $C$ is the insert-contig containing $r$ from which the alignment $A[j]$ is derived, then $A[j].read\_strand = L[k].strand$, where $L[k].id = r$, and $A[j].contig\_strand = H[i].strand$, where $H$ is the alignment array of $C$. We say that the alignment of $A[j]$ with respect to the reference is *forward* if and only if $A[j].read\_strand = A[j].contig\_strand$, *reverse* otherwise.

### Reads alignment extraction

Each time a contig $C$ is selected, the alignment of every read contained in the layout $L$ of $C$ is processed. Given a read $r$ used to assemble $C$, let $L[k]$ be the information on $r$ (*i.e.*, $L[k].id = r$) and let $A$ be the alignment array of $r$. Also, let $H$ be the alignment array for $C$. Information provided by $H$ and $L$ is sufficient to retrieve the alignment of $r$. If $A$ contains $j-1$ alignments, with $j \geq 1$, then the current alignment should be used to fill $A[j]$. In particular we obtain:

$$\begin{aligned} &A[j].ref = H[1].ref;\\ &A[j].read\_strand = L[k].strand;\\ &A[j].contig\_strand = H[1].strand. \end{aligned}$$

Notice that fields $.ref$ and $.strand$ are the same for each component of $H$. Concerning $A[j].loc\_coord$, it suffices to select the hits of $H$ overlapping $r$ in the layout. Such hits are those satisfying one of the following conditions:

$$\begin{aligned} &L[k].start \leq H[i].q\_start \leq L[k].start + L[k].length - 1;\\ &L[k].start \leq H[i].q\_end \leq L[k].start + L[k].length - 1. \end{aligned}$$

Pairs $\langle H[i].q\_start, H[i].q\_end\rangle$ are directly added to $A[j].loc\_coord$, except for the first and last intervals, which are intersected with $[L[k].start, L[k].start + L[k].length - 1]$. If the alignment $H$ is forward, then the components of $A[j].coord$ are computed by operating a simple change of coordinates. In the case the alignment is reverse, the situation is a bit more complicated because insert and reference coordinates arrays must be scanned in opposite directions. At the end, $A[j]$ will contain two ordered sets of local and global coordinates, respectively, referred to the reference (in fact, information on alignment strand is kept within $A[j].read\_strand$ and $A[j].contig\_strand$). If $j = 1$ then the procedure goes on by considering the successive read $L[k+1].id$ in the layout. Otherwise, the alignment $A[j]$ must be compared with the existing ones, and possibly deleted, before continuing with $L[k+1].id$. This point is discussed below.

### Multiple alignments and redundancy elimination

In the following, criteria to decide whether or not $A[j]$ is a new alignment for $r$ are presented. There are two cases in which the decision is easy to take: (i) $A[j].coord$ is disjoint from any other $A[j'].coord$, or (ii) $A[j].coord = A[j'].coord$, for some $j' < j$. In case (i), the alignment is kept, whereas in case (ii), the alignment is discarded, as it is identical to a previously reported one. Only alignments that are distinct from every stored alignment should be added to $A$, otherwise they are neglected or, at most, used for refining an existing alignment. Given a read $r$, let $A[j']$ and $A[j'']$ be two alignments for $r$. We say that $A[j']$ and $A[j'']$ are *redundant* if and only if:

(a) $A[j'].ref = A[j''].ref$;

(b) $A[j']$ and $A[j'']$ have the same strand;

(c) $A[j'].coord \cap A[j''].coord \neq \emptyset$ and contains the same number of intervals in $A[j'].coord$ and $A[j''].coord$, respectively;

(d) internal coordinates of $A[j'].coord \cap A[j''].coord$ are the same;

(e) the relative position of boundaries is the same.

We say that $A[j']$ and $A[j'']$ are *distinct* otherwise. Condition (c) states that $A[j']$ and $A[j'']$ must share a non-empty sequence and that a gap occurs in $A[j']$ if and only if it occurs also in $A[j'']$, that is, there must not be gaps in the exonic sequence. Condition (d) states that the alignment coordinates must be the same, at least for those lying on intron boundaries. Condition (e) puts in relation local coordinates with global coordinates. Two alignments may come from the same region even though their boundaries are different; in contrast, two alignments may be different even if they come from the same region. The important point here is that the *difference* between global and local coordinates must be the same, not the absolute positions. If $A[j]$ is found to be distinct from any other alignment for $r$, then it is maintained. If $A[j]$ is found to be redundant with respect to a previously added alignment $A[j^*]$, then $A[j^*]$ should be updated using information coming from $A[j]$. Update is performed whenever one of the following two scenarios occurs: (i) $A[j]$ extends $A[j^*]$, or (ii) $A[j^*]$ highlights a splicing event that $A[j]$ did not find. In all other cases, $A[j^*]$ remains unchanged. Case (i) occurs when the first (the last) overlapping sequence of $A[j]$ starts before (ends after) the starting coordinate (the ending coordinate) of $A[j^*]$. Case (ii) arises when $A[j]$ presents a gap in the reference sequence that is not present in $A[j^*]$. In other words, $A[j].coord$ contains more elements than $A[j^*].coord$. This means that, for $A[j]$, BLAST finds a better alignment by splitting the insert-contigs into an additional hit; this possiblity did not occur when processing $A[j^*]$ because the length of the insert-contig subsequence beyond the gap was too short. In case (i), $A[j^*]$ is extended by keeping the new first (last) interval of $A[j]$. In case (ii), $A[j^*]$ is modified by adding a new hit to the alignment provided by $A[j]$. At this point, regardless whether $A[j]$ has been used to refine $A[j^*]$, it is eliminated from the alignment array $A$.

### Paired reads-guided alignment selection

After all insert-contigs have been processed, the final alignment array of each read in the library is available. We decided to choose a single alignment for each multiply mapped read, keeping, however, the information on whether the alignment was originally unique or not. The criteria we use take advantage of paired reads information in order to recognize *pair-compatible* alignments. If multiple pair-compatible alignments are found, we choose one of them at random. Given a paired read $(r, r')$, let $A$ and $A'$ be their associated alignment arrays. Suppose that both $A$ and $A'$ are non-empty and that at least one among $A$ and $A'$ contains more than one element. We say that $A[j]$ and $A'[j']$ are *pair-compatible* if and only if the following hold:

(a) $A[j].ref = A'[j'].ref$;

(b) $A[j]$ and $A'[j']$ have opposite strand;

(c) if $A[j]$ is forward, then $A'[j']$ starts after $A[j]$ but not farther than $max\_intron\_size + insert\_size$;

(c') if $A[j]$ is reverse, then $A[j]$ starts after $A'[j']$ but not farther than $max\_intron\_size + insert\_size$;

If no pair-compatible alignments are found, then a random alignment is selected. Otherwise, non-compatible alignments are discarded and the random choice is done on pair-compatible ones only.

### 6.1.4 Implementation

The algorithm has been implemented in C++ and requires as input the layout computed by GapFiller. We employed the same data structures described in Section 4.4 to retrieve the layout from the condensed representation.

The total disk space required to store layout information may be very huge and, in general, unpredictable. This is particularly true when dealing with RNA-Seq data, because being able to accurately estimate the amount of information to be stored, means being able to quantify expression levels in advance. For this reason, using a method to store the layout of each contig in a condensed way, as explained in Chapter 4, turns out to be crucial in tacking insert-guided alignment of RNA-Seq data.

Moreover, we used information on the dataset in order to store layout information in a succint way, to minimize the disk space required (see also Section 4.4.2).

## 6.2 Results

In this section we evaluate the results otained with our pipeline. The goal is to accurately measure the accuracy of our method in both detecting splicing sites and mapping reads with gaps. Our assembly-first approach, together with the careful exons boundaries detection, is specifically addressed to ambiguity reduction. This characteristic of the method should allow to reach high specificity in junctions prediction, and high precision in alignment.

In order to completely characterize the output of our pipeline, we test on simulated data. We employ an external tool for RNA-Seq libraries simulation, deeply analyze the type of detected junctions, collect statistics on reads alignment, and compare the performances with other three state-of-the-art tools for gapped alignment and splicing sites detection. Moreover, we compare the performances on different organisms in order to evaluate the ability of tools to cope with different gene structure and alternative splicing patterns.

### 6.2.1 Datasets

We performed experiments on simulated RNA-Seq paired reads. The reference genomes we tested on were *Arabidopsis thaliana* and *Homo sapiens* chromosome 1. For simulating RNA-Seq libraries we employed Flux simulator [57]. We simulated expression, library construction, and sequencing. The location of each read in the genome is provided, allowing us to precisely check the performances of our pipeline. For *A. thaliana*, Flux simulator was run with different values of parameter `EXPRESSION_K` ranging from $-0.3$ to $-0.8$, which determines the variability in differential expression levels [47]. Each simulation produced $\sim$ 5 M reads.For human chr1, RNA-Seq paired reads libraries were simulated varying coverage instead of expression level. We simulated libraries ranging from 1 M to 5 M reads. Reads length has been set to 100 bp in all simulations.

### 6.2.2 Design of experiments

An input preprocessing was performed. We eliminate Ns from the reads sequences and discarded reads shorter than 30 bp. Reads resulting to be unpaired after preprocessing are discarded. The first step of the pipeline is to run GapFiller on the paired reads selected at the preprocessing stage. For all datasets, we set the minimum overlap length (parameter `overlap`) to be 30 bp and we required 1 overlapping read at each extension step (parameter `extThr`). We also stored the reads information for each insert-contig produced (option `layout`). The output consists of a FASTA file of insert-contigs and a text file containing the layout. The second step consists of insert-contigs alignment against the reference genome. BLAST was run requiring a percentage of identity of 95%. After hits selection and alignment refinement, we kept only insert-contigs whose sequence was covered by hits for at least 98% of their length. During the third step the reads alignment is retrieved. The output consists of alignment files (in BED and BAM format, respectively) and of a text file containing information on the discovered junctions.

### 6.2.3 Analysis of correctness

We evaluated our results in terms of (i) junctions prediction accuracy and (ii) reads alignment precision. We did not perform extensive tests, but the results we obtained were highly encouraging. In particular, we did thorough analysis of our pipeline performances on simulated data. The results from the first step of the pipeline, *i.e.*, the assembly of insert-contigs with GapFiller, are reported in Table 6.1. Discarded insert-contigs are those whose aligned sequence (with respect to the genome) is below 98%. We observed that this was mainly due to the presence of a splicing dinucleotide at the beginning (at the end) of an insert-contig. However, this does not sensibly affect the number of mapped reads (see Tables 6.4 and 6.5).

Concerning junctions evaluation, we wanted to test the ability of our pipeline both to identify splicing events and to correctly report junctions. At the second step of our pipeline, the presence of dinucleotide pairs GT-AG, GC-AG, and AT-AC is checked. We flag *validated* every junction for which the dinucleotide pair is (uniquely) found, *non-validated* otherwise. A junction annotated in the reference will be referred to as *canonical* if it is delimited by either GT-AG, GC-AG, or AT-AC, *non-canonical* otherwise. A schematic view of the above classification is reported in the following table:

| | GT-AG/GC-AG/AT-AC | other dinucleotides |
|---|---|---|
| annotated junction | canonical | non-canonical |
| predicted junction | validated | non-validated |

A validated junction provides information on the exact donor and acceptor dinucleotides, whereas a non-validated junction is an evidence that a splicing event occurred, with no precise information on its location. A *true positive* (TP) is consequently defined as a junction found by our method that is actually annotated. We distinguish among four

**Table 6.1:** Simulated datasets. Reads selected after preprocessing and insert-contigs assembled by Gap-Filler.

| | library size | assembled insert-contigs | discarded insert-contigs |
|---|---|---|---|
| *A. thaliana* | 4, 994, 194 | 2, 418, 328 | 258,155 |
| | 4, 993, 886 | 2, 425, 671 | 255,370 |
| | 4, 994, 868 | 2, 435, 295 | 254,765 |
| | 4, 995, 236 | 2, 440, 141 | 251,375 |
| | 4, 999, 262 | 2, 428, 268 | 248,080 |
| | 4, 996, 600 | 2, 439, 987 | 273,541 |
| | 4, 993, 894 | 2, 427, 707 | 269,980 |
| *H. sapiens* chr 1 | 970, 262 | 435, 668 | 85,067 |
| | 1, 930, 878 | 889, 416 | 174,709 |
| | 2, 934, 638 | 1, 361, 972 | 272,156 |
| | 3, 831, 788 | 1, 770, 762 | 353,741 |
| | 4, 895, 284 | 2, 311, 954 | 465,542 |

types of TP, depending both on the annotation type (canonical/non-canonical) and on the prediction type (validated/non-validated):

| | Canonical | Non-canonical |
|---|---|---|
| Validated | TP1 | TP2 |
| Non-validated | TP3 | TP4 |

Cases TP1 and TP3 refer to junctions whose position can be exaclty determined by our method, as they are identified by canonical dinucleotides. Cases TP2 and TP4 instead are junctions that cannot be determined precisely. In particular, cases TP1 and TP4 are those expected, *i.e.*, a canonical junction correclty flagged validated, and a non-canonical junction that is not validated. In case TP2, our pipeline was able to refine the annotation, by finding a canonical pair of dinucleotides near a junction reported as non-canonical. Case TP3 instead, is a canonical junction that is not validated by our method; if our approach is accurate, then the number of TP3 should be low. Of course, we want FP to be low as, in the case of simulated data, they represent uncorrectly predicted junctions.

The results on *Arabidopsis thaliana* and human chr1 are reported in Tables 6.2 and 6.3, respectively. Accuracy is extremely good, as FP rate is below 0.4% for all experiments on *A. thaliana*, and below 0.6% for those on chr1, respectively. This shows high accuracy in the alignment technique, which is a consequence of mapping insert-contigs onto the genome instead of RNA-Seq reads directly. Insert-contigs alignment reduces the number of gap opens outside introns, hence allows for robust splicing events identification. We noticed also that the amount of TP3 is quite low, especially for chr1, meaning that when our pipeline detects a canonical splicing event, it is able to precisely identify its location most of the times (*i.e.*, more than 98% for *A. thaliana* and 92%-95% for chr1, respectively). A further consideration is in order on case TP2, that is, junctions annotated as non-canonical but validated. We found that our pipeline was able to refine the annotation of few junctions, for *A. thaliana*, and of up to 5% of the non-canonical predicted junctions, for human chr1. Moreover, we noticed that the performances of our pipeline only slightly change with expression variability (*A. thaliana*) or dataset size (human chr1).

As far as reads alignment is concerned, we computed accurate statistics on the ability of our method, especially when gaps occur. To this aim, we compared our BED file with

**Table 6.2:** Simulated experiments on *A. thaliana*. Junctions prediction statistics.

| TP | | | | | FP |
|---|---|---|---|---|---|
| TP1 | TP2 | TP3 | TP4 | Total | |
| 21604 | 411 | 317 | 20156 | 42488 | 110 |
| 20439 | 387 | 302 | 18667 | 39795 | 141 |
| 17764 | 358 | 272 | 16534 | 34928 | 110 |
| 17624 | 316 | 263 | 16157 | 34360 | 76 |
| 16131 | 326 | 237 | 15097 | 31791 | 109 |
| 15994 | 288 | 219 | 15124 | 31625 | 104 |
| 14326 | 271 | 211 | 13239 | 28047 | 89 |

**Table 6.3:** Simulated experiments on *H. sapiens* chr 1. Junction prediction statistics.

| TP | | | | | FP |
|---|---|---|---|---|---|
| TP1 | TP2 | TP3 | TP4 | Total | |
| 152 | 385 | 8 | 7420 | 7965 | 43 |
| 157 | 473 | 10 | 8776 | 9416 | 48 |
| 169 | 530 | 16 | 9526 | 10241 | 53 |
| 192 | 527 | 10 | 10030 | 10759 | 64 |
| 189 | 514 | 17 | 9574 | 10294 | 59 |

**Table 6.4:** Simulated experiments on *Arabidopsis thaliana*. Percentage of correct reads alignment reported by our pipeline.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| aligned (%) | 97.82 | 97.99 | 97.88 | 98.03 | 97.68 | 98.01 | 97.34 |
| perfect (%) | 86.45 | 86.64 | 86.37 | 86.79 | 87.82 | 86.61 | 87.54 |
| reference (%) | 99.14 | 99.30 | 99.45 | 99.23 | 99.50 | 99.02 | 99.44 |
| start (%) | 97.84 | 97.93 | 98.02 | 98.03 | 98.55 | 97.51 | 97.97 |
| end (%) | 96.90 | 96.87 | 97.06 | 96.80 | 97.47 | 96.76 | 97.01 |
| strand (%) | 99.38 | 99.55 | 99.38 | 99.49 | 99.67 | 99.49 | 99.60 |
| hits number (%) | 99.43 | 99.44 | 99.42 | 99.41 | 99.42 | 99.31 | 99.36 |
| hits length (%) | 87.92 | 88.08 | 87.70 | 88.20 | 88.70 | 88.04 | 88.83 |
| hits start (%) | 89.70 | 89.92 | 89.56 | 90.05 | 90.39 | 89.94 | 90.37 |

**Table 6.5:** Simulated experiments on *H. sapiens* chr1. Percentage of correct reads alignment reported by our pipeline.

| | | | | | |
|---|---|---|---|---|---|
| aligned (%) | 90.29 | 92.89 | 94.07 | 94.17 | 94.98 |
| perfect (%) | 77.52 | 77.95 | 75.50 | 76.38 | 77.98 |
| start (%) | 92.47 | 93.27 | 91.34 | 91.56 | 93.14 |
| end (%) | 90.06 | 90.58 | 88.77 | 88.83 | 90.32 |
| strand (%) | 96.78 | 96.86 | 96.13 | 96.45 | 97.12 |
| hits number (%) | 98.90 | 98.91 | 98.90 | 98.83 | 99.00 |
| hits length (%) | 82.61 | 82.28 | 81.28 | 82.06 | 82.67 |
| hits start (%) | 88.70 | 88.45 | 88.00 | 88.36 | 88.81 |

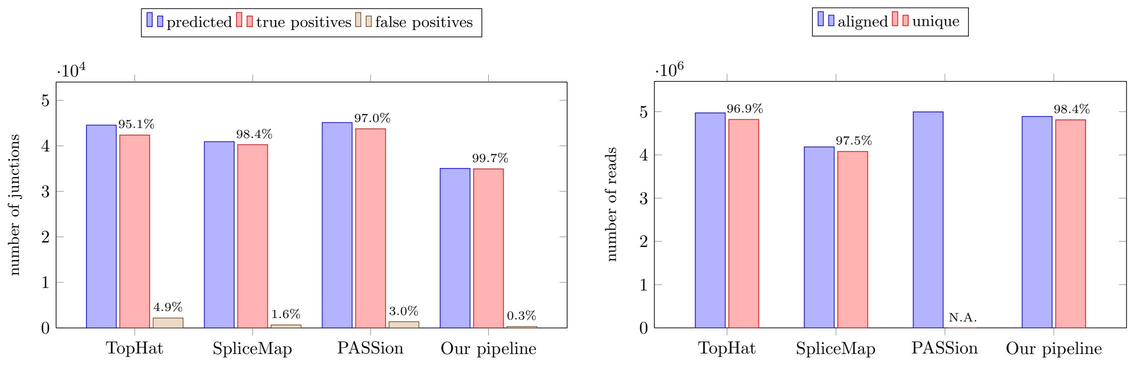
that provided by the simulation pipeline. The results of the comparison are reported in Tables 6.4 and 6.5. Concerning *A. thaliana* datasets, we observed that more than 97% of the reads is aligned against the reference. Beyond rough statistics, *i.e.*, the percentage of reads correclty matching reference chromosome and strand, we observed that our method performs very well in splitting the reads in the right number of hits (hits number field). This means that we have been able to detect reads spanning a splicing sites most of the times, and this translates into more than of 99% correctly split reads. The worst values are for the number of perfect alignments and for the exact length and location of the hits of a single read alignment. This is due to the presence of non-canonical junctions (see also Table 6.2), which caused a little misplacement of splicing dinucleotides. Reads mapping evaluation metrics for chr 1 are reported in Table 6.5. In this case, notice that the percentage of aligned reads increases with the library size (see also Table 6.1). This means that the more insert-contigs are available, the more chances to align a read are found. And such alignment maintain the same degree of accuracy; indeed, the statistics on alignment correctness, computed with respect to the number of aligned reads, do not depend on the library size. The determination of reads position in the reference is not as good as for *A. thaliana*, because the splicing patterns are more complicated and tend to hinder correct alignment. Indeed, the number of discarded insert-contigs (*i.e.*, those aligned for less than 98% of their length) is quite high, due to the pervasive presence of short tails that cannot be safely mapped by BLAST. Similarly to the results on *A. thaliana*, we found out that the split alignments were correctly detected, whereas the exact alignment coordinates suffer from the high percentage of non-canonical splicing sites (see also Table 6.3).

### 6.2.4 Comparison with state-of-the-art tools

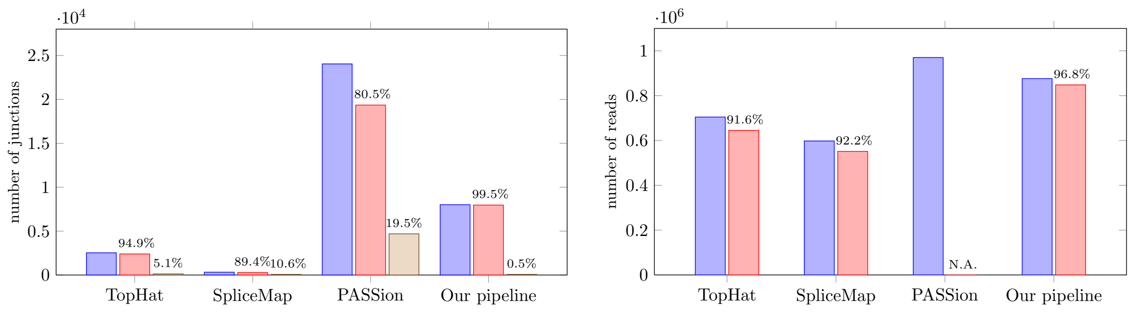
Comparison were done with respect to three tools for RNA-Seq reads mapping and splicing junctions detection, *i.e.*, TopHat [164], SpliceMap [7], and PASSion [181]. Evaluation of the performances of the four tools are reported in Figure 6.4. Concerning junctions prediction (left column), we observed that PASSion reports many events, and that our pipeline has the lowest FP rate. PASSion actually reports more junctions than those predicted, and we observed that some of them are redundant, resulting both in an overestimation of the splicing pattern complexity, and in a high FP rate (*e.g.*, nearly 20% for chr1 dataset). As far as reads alignment is concerned (right column), we noticed that our method is extremely sensitive and precise in mapping simulated reads. Also the other tools behave well on *A. thaliana* simulated dataset; we cannot say anything about PASSion, for which we were not able to compute the actual number of uniquely mapped reads. Another important point is that our pipeline behaves quite well on both *A. thaliana* and *H. sapiens* chr1 datasets, thanks to GapFiller, which do not make assumption of the input data type. In contrast, it is quite evident that TopHat and SpliceMap performances are strongly related with the specific organism.

## 6.3 Conclusions and future work

In this chapter, we discussed a new method for RNA-Seq reads alignment and splicing sites detection. The idea of guiding alignment through the layout of assembled inserts allowed to reach high precision.



**(a)** *A. thaliana* simulated data.



**(b)** *H. sapiens* chr1 simulated data.

**Figure 6.4:** Comparison of tools performances on junctions detection (left column) and reads alignment (right column). Reads uniquely mapped by PASSion are not reported.

Results on simulated data show that our method turns out to be powerful in detecting splicing events, and to correclty validate canonical junctions. This allows to align the RNA-Seq reads in the correct number of hits. Such a feature is of utmost importance for junctions quantification, and, more broadly, for gene expression estimation.

We conclude that our method can cope with different (and complex) splicing pattern types. Future work will include the employment of a more sophisticated technique to choose the best alignment for each multiply mapped read, beyond the pair-compatibility check, and exploiting organism-related heuristics. More in general, the proposed technique can be extended to broader applications (*i.e.*, larger datasets) by employing a method for eliminating redundancy from the layout.

# 7 Estimating expression levels of alternative transcripts

When both a reference genome and an annotation are available, there are still challenging issues that can be addressed starting from raw RNA-Seq reads. Among the most important ones we mention the problem of estimating expression levels of gene isoforms and identifying which isoforms are differentially expressed within different RNA samples. The latter strongly depends on the ability of accurately quantifying transcripts abundances for each dataset, other than on the statistical model employed, and computing correct expression levels envisages as previous step a careful mapping procedure that assigns each exon the correct number of reads.

An accurate estimation of exons coverage depends not only on the algorithm or tool employed for RNA-Seq reads mapping, but also on intrinsic biases of RNA-Seq libraries. On the one hand, the presence of mismatches and short indels can be partially accomplished by specifically designed algorithms for NGS reads mapping, on the other hand, uneven transcripts coverage is difficult to threat during the alignment phase.

Variability of transcripts coverage by RNA-Seq reads is seen, at the genomic level, as an increase or drop in coverage along exons. This fact can be erroneously interpreted as an evidence for either alternative isoforms to be expressed simultaneously, or the detection of a new isoform that was not yet annotated. Thus, *ad hoc* methods should be employed to take into account biases in transcripts coverage, possibly identifying exons that should not be included in the model for the computation of expression levels.

In the following we briefly recall the insert assembly-based alignment technique proposed in previous chapter, and discuss a new method that, by identifying a subset of high-quality representative exons for each gene, allows to retrieve the expression level of each isoforms by simply solving a system of linear equations.

## 7.1 A parsimony approach to transcripts quantification

It can be reasonable to identify, when possible, regions of transcripts in which reads are correctly mapped or accurately assembled. In some cases, when the goal of estimating expression levels is addressed, it is not necessary to assemble all exons completely, or consider all reads alignments. Intuitively, the exon-isoform structure of a gene should allow to identify one or more subsets of exons such that the knowledge of cumulative exons coverage is sufficient, at least in theory, to retrieve the expression level of each isoform. This would translate into the possibility of estimating the expression level of all

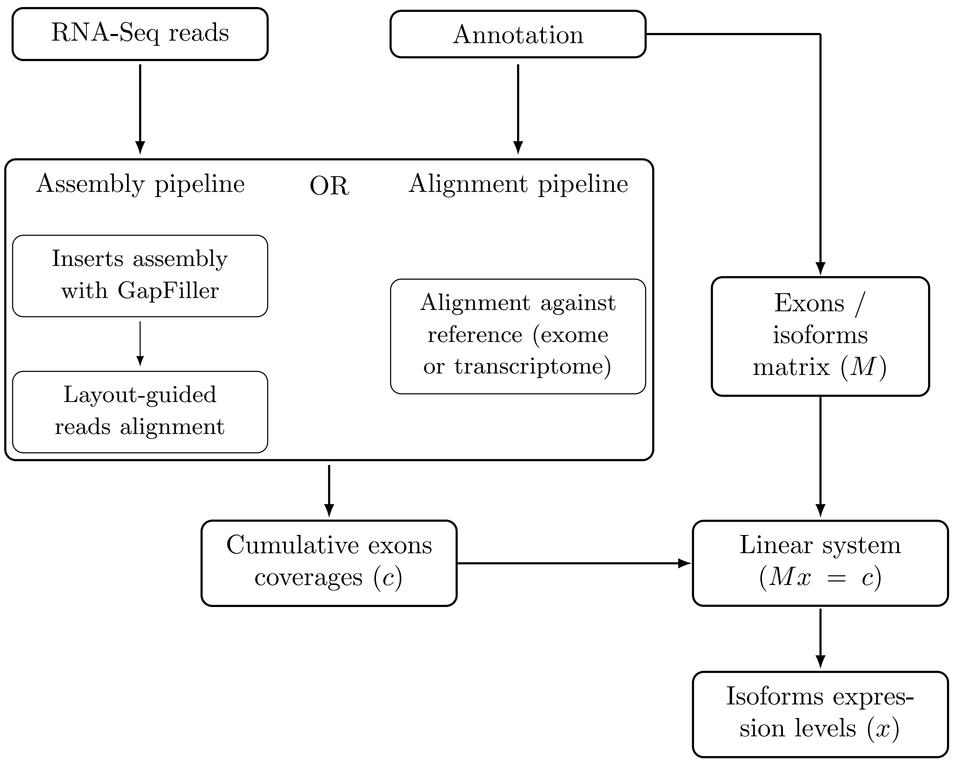

RNA-Seq reads
Annotation
Assembly pipeline
OR
Alignment pipeline
Inserts assembly with GapFiller
Layout-guided reads alignment
Alignment against reference (exome or transcriptome)
Exons / isoforms matrix ($M$)
Cumulative exons coverages ($c$)
Linear system ($Mx = c$)
Isoforms expression levels ($x$)

**Figure 7.1:** A possible pipeline that takes as input (i) RNA-Seq reads libraries and (ii) an annotation of esons and isoforms, and returns as output the etimated expression levels of isoforms.

isoforms of a gene by looking at a key number of exons.

A first observation in this respect is that, given a gene $G$ and an exon $E$ belonging to only one isoform $I$ of $G$, then the expression level of $I$ should be equal to the coverage of $E$ with RNA-Seq reads. Exon $E$ is *characteristic* of $I$ and, in principle, it suffices to get the expression level of $I$. In the following we extend this reasoning in two directions. First, we do not require characteristic exons in order to compute expression level, instead we assume that the exon-isoform structure can be represented as a full-rank binary matrix. Second, we introduce a measure of the quality of an exons in order to introduce only good ones in our linear model.

The idea is to divide the analysis into three steps, which are depicted in Figure 7.1 [122]. Step 1 consists in computing cumulative expression levels of exons, using either an alignment algorithm specifically designed for RNA-Seq data, or an assembly tool from which reads layout can be retrieved. In our case we employ the local assembly-guided pipeline introduced in Chapter 6.

Step 2 consists in computing a system of linear equations (*e.g.*, a linear system defined by a linear matrix $\mathbf{M}$) for each gene, which states the relationship between expression levels of isoforms and cumulative expression levels of exons. Notice that step 1 and step 2 can be run independently.

Finally, step 3 consists in solving the linear systems provided at step 2 using the data computed at step 1; the output should be the expression levels of the isoform of each gene.

### 7.1.1 General definitions

The computational problem of estimating expression levels regards the collection of alternative isoforms of a gene as a family of sets that overlap each other in correspondence of common genomic sequences. Each isoforms sharing some sequence $E$ will contribute to genomic coverage with its transcript abundance, or expression level. We will refer to $E$ as exon, despite $E$ may be a fraction or a concatenation of exons. The fundamental requirement is that $E$ should not be broken into distinct isoforms. We opt for the definition of exon as a maximal sequence with previous property, that also do not span any splicing junction.

A gene $G$ can be identified by an ordered set of exons, *i.e.*, $G = \langle E_1, \ldots, E_m \rangle$, and its isoforms $\mathcal{I} = \{I_1, \ldots, I_n\}$, by the set of ordered subsets $I_j \subseteq G$, for each $j = 1, \ldots, n$. Given $G$ with $m$ exons and $n$ isoforms, let us define its *associated* $m \times n$ *boolean matrix* $\mathbf{M} = (M_{ij})$, as follows:

$$M_{ij} := \begin{cases} 1 & \text{if } E_i \in I_j; \\ 0 & \text{otherwise.} \end{cases} \tag{7.1.1}$$

In this formalism the estimated exons coverages can be represented as a vector $\mathbf{c} \in (\mathbb{R}^+)^m$ whereas the (unknown) expression levels of isoforms as a vector of variables $\mathbf{x} \in \mathbb{R}^n$.

At this point, a possible—and, actually, the most widely used—approach to determine $\mathbf{x}$ consists in using the following system:

$$\mathbf{M}\mathbf{x} = \mathbf{c}. \tag{7.1.2}$$

Note that, if exons exact coverages were known, then the following identity

$$c_i = \sum_{\substack{j=1 \\ E_i \in I_j}}^{n} x_j \tag{7.1.3}$$

would hold for all $i = 1, \ldots, n$.

In this context, system (7.1.2) gives us the possibility to study a *space* of possible solutions. Moreover, a solution exists if $\mathbf{c}$ is a linear combination of the columns of $\mathbf{M}$, with unicity guaranteed whenever $rank(\mathbf{M}) = n$.

In real cases, however, the entries of $\mathbf{c}$ are known only in estimate and may introduce inconsistencies. For this reason, the solution to (7.1.2) is often searched using a least-square approach and providing a suitable *objective function* to be minimized together with a set of constraints to be satisfied. The objective function can be seen as a *measure* of the error between $\mathbf{M}\hat{\mathbf{x}}$ and $\mathbf{c}$, where $\hat{\mathbf{x}}$ is an approximate solution, while the constraints represent additional knowledge we may have on the system's solutions.

### 7.1.2 Approximate solutions to the linear system associated to a gene

In the literature several studies have been carried out in order to define a suitable model formulation [90, 97, 165]. High accuracy in the model description, together with an (often complex) statistical machinery for parameters estimation (possibly requiring hypotheses on reads distribution), is of undeniable importance to reach the best possible precision of the solutions, see [163]. However, even integrating such techniques, a least-square formulation is intrinsically not able to cope with the *structure* of the problem, as well as with the

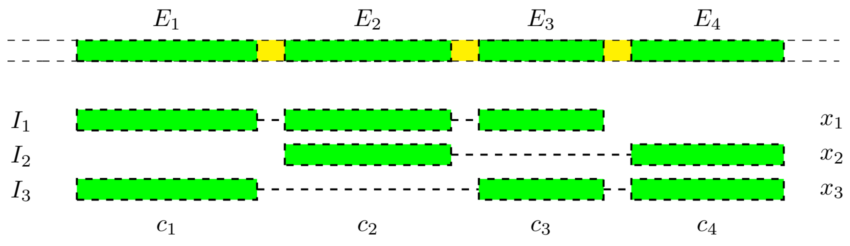


**Figure 7.2:** A gene $G$ with four exons and three isoforms. In order to retrieve the expression levels of $I_1$, $I_2$, $I_3$ only 3 exons are necessary. Sets $\{E1, E_2, E_4\}$ and $\{E_2, E_3, E_4\}$ are individually sufficient to retrieve $x_1$, $x_2$, $x_3$ from exons coverages.

structure of its sub-problems. Moreover, bad quality coverage estimates may affect the accuracy of the final solution. On this last aspect we observe that, clearly, *any* method for expression levels estimation would suffer from reads biases and coverage irregularities. Nevertheless, being able to compute both *local* and *global* information on the gene, using single exons features and their relationships with other exons in the gene, respectively, may aid accurate solution computation. In our approach, instead of adding constraints or penalty terms within the linear model, we introduce a quality parameter measuring both local and global exon's coverage accuracy.

In general, in order to be able to establish the expression level of the isoforms of a gene, not all exons coverages are necessary. The structure of a gene can allow to identify sets of exons such that knowledge on their *cumulative expression level* is sufficient, at least in theory, to deduce the expression level of each isoform.

The interpretation of our general approach to find a solution to (7.1.2) can be given as follows. If the exons and the isoforms of a gene $G$ are known, then it is possible to check whether the structure of $G$ allows to retrieve the expression levels of each isoform from the knowledge of exons coverage by means of RNA-Seq reads. Moreover, when the previous hypothesis holds, one can also identify *minimal* sets of exons that, in principle, are individually sufficient to retrieve the expression levels of all isoforms. At this point, a criterion to select the *best* subset of exons can be introduced. A crucial point is that of finding a quality function that best describes the level of reliability of each exon as a function of, *e.g.*, length, coverage uniformity, and coverage consistency with respect to other exons. This point is discussed in Section 7.3.4.

A mandatory step to be done before expression levels computation, consists in estimating the entries of coverage vector $\mathbf{c}$ precisely. To this aim, we propose a method that circumvents the problem of short RNA-Seq reads mapping against the reference genome by firstly assemblying them into longer sequences, by actually defining an *indirect* alignment technique. The details of RNA-Seq reads alignment and coverage computation phase are given in the next section.

## 7.2 Computation of exons coverage

Our aim here is to accurately estimate cumulative expression levels of exons to ease the computation of expression levels of isoforms. At this step we want to provide a method that is hopefully more robust than simply aligning RNA-Seq reads agaist the reference genome directly. Our approach tends to provide *correct* rather than *complete* output, because in our model is crucial to get precise exons coverage estimation. Once we have cumulative expression levels, we can use them in our model and then compute expression levels of isoforms.

### 7.2.1 Assembly-guided RNA-Seq reads alignment

Our mapping technique relies on an indirect method that firstly assembles RNA-Seq paired read inserts and then retrieves reads alignments from contigs layout (see also Section 6.1). We work in the hypothesis that a reference genome is available, but at this stage an input annotation is not required.

We employ GapFiller [123] to provide a local assembly of insert sequences. The advantages of GapFiller are: (i) it is designed to assemble a genomic sequence identified by its ends (*i.e.*, reconstruct inserts from paired reads), (ii) it allows to retrieve the layout, and (iii) it is able to output correct sequences (*i.e.*, high specificity). If the seed-and-extend technique employed for the assembly eventually reaches the mate of the seed read, the obtained sequence is considered correct and is called *insert-contig*.

In the context of transcripts abundances estimation, information provided by inserts is a valuable one for precise alignment, but exons coverage cannot be computed directly from ireconstructed sequences, despite being accurate. As a consequence, going back to the reads is mandatory. The assembly technique used by GapFiller allows to simply store the layout of each insert-contig $C$, that is, the position and orientation of each read employed for the assembly. Contigs layout allow to either keep only reads belonging to correctly assembled inserts, and eliminate redundant coverage by keeping only one copy of a read.

Insert-contigs are first aligned against the genome using BLAST [55] (but a different tool for gapped alignment can be employed as well). Then, high quality alignments are extracted from BLAST output and reads locations on the genome are computed combining layout and alignment information. For each insert-contig $C$, BLAST hits are sorted by non-increasing score and non-overlapping hits are selected in succession until $C$ is almost completely covered. If multiple hits are selected, then they are refined in correspondence of putative splicing junctions by checking for the presence of donor-acceptor dinucleotide pairs.

At this point, reads location is retrieved and multiple alignments are disambiguated in different ways, and, depending on their relative position, they are either merged or discarded if they turn out to be pair-compatible. If, after previous checks, multiple alignments still exist, a single one is reported at random.

### 7.2.2 Coverage computation from reads alignment

Each read is assigned a precise region in the genome in order not to introduce coverage artifacts, as stated in previous section. Coverage is computed by plotting reads abundances on the reference. More precisely, a WIG file is generated that reports, for each position $x$ in the genome, the number of reads covering $x$.

At this point, genome annotation is needed in order to compute cumulative exons coverages. For each exon $E$, reads abundancies are computed at nucleotide resolution. Coverage is then estimated in two steps: (i) the average coverage on fixed-length subsequences of $E$ is computed, and (ii) the final coverage estimate for $E$ is computed by keeping the average again, neglecting minimum and maximum peaks. This method should allow to reduce the standard deviation of average exons coverage.

## 7.3 Maximum quality linear sub-system

Let us consider a gene $G$ with four exons and three isoforms, as depicted in Figure 7.2. Based on notation given in Section 7.1.1, we have

$$x_1 + x_3 = c_1 \tag{7.3.1a}$$
$$x_1 + x_2 = c_2 \tag{7.3.1b}$$
$$x_1 + x_3 = c_3 \tag{7.3.1c}$$
$$x_2 + x_3 = c_4. \tag{7.3.1d}$$

Not all equations are needed to solve the system, for instance, equations (7.3.1a), (7.3.1b), and (7.3.1d) are sufficient to obtain a solution. However, the constraint $c_1 = c_3$ imposed by Equation (7.3.1c) may not be satisfied by the solution $x$ of the sub-system defined by (7.3.1a), (7.3.1b), and (7.3.1d). In contrast, the uniqueness of a solution depends on the existence of at least 3 *independent* equations. These two aspects are related to (i) the coverage uniformity along transcripts, and (ii) the gene structure.

If the exons and the isoforms of a gene $G$ are known, then it is possible to check whether the isoforms/exons structure of $G$ allows to retrieve the expression levels of each isoform from the knowledge of exons coverage by means of RNA-Seq reads. Moreover, in case previous hypothesis holds, one can also identify every *minimal* set of exons that, in principle, are individually sufficient to retrieve the expression levels of isoforms. It must be stressed that a correct quantification of expression levels of isoforms strongly depends on the data. In the ideal case, three equations are enough but, in practice, it is more reasonable to check the consistency of RNA-Seq reads coverage for each gene, for example, in terms of constraints violation (*e.g.*, when $c_1 \neq c_3$).

Based on previous example, the existence of a solution depends on the accuracy of expression levels estimation. Hence, it should be reasonable to identify good quality exons on which transcripts abundances should be computed.

It is fair to say that the applicability of our method depends on genes structure. In other words, we must check whether it is possible to find a unique solution to the linear system (step 2). In practical scenarios, we observed that the percentage of genes that are intractable with our method is negligible. An example is *Arabidopsis thaliana*, for which 32665 out of 32678 genes can be analyzed with our method.[1]

### 7.3.1 Definitions and notations

In the following, we assume exonic sequences to be disjoint from one another. This is not a restriction, as genes with overlapping exons can be reduced to this definition by simply splitting exons in correspondence of overlaps. In general, $E_i$ may be fractions or concatenations of "true" exons. We will identify with $E_i$ a maximal contiguous sequence such that, for all $j = 1, \ldots, n$, either $E_i$ is a subsequence of $I_j$ or the overlap of $E_i$ and $I_j$ sequences is the empty string.

[1]Source: TAIR [65]

We will denote with $\mathbf{M}_i$ the $i$-th row of $\mathbf{M}$, that is, a row vector with $n$ components. Given a set of indexes $H \subseteq \{1, \dots, m\}$, we will indicate with $[\mathbf{M}_h]_{h \in H}$ the $|H| \times n$ submatrix of $\mathbf{M}$ whose rows are the ordered rows of $\mathbf{M}$ with indexes in $H$.

### 7.3.2 Problem definition

Given a gene $G$, in our model we want to retrieve the expression levels $x_j$ of $I_j$, for each $j = 1, \dots, n$, knowing the annotation, identified by $\mathbf{M}$, and the estimated coverages of exons, identified by vector $\mathbf{c}$. The method we propose targets two problems: (i) treats the cases in which the linear system $\mathbf{Mx} = \mathbf{c}$ has no solutions, and (ii) copes with coverage biases, that is, equalities of the form (7.1.3) that are not satisfied, for some $i$.

The case in which $\mathbf{Mx} = \mathbf{c}$ has infinite solutions, corresponds to the situation in which there is not enough information to indentify the expression levels for each isoform. If this is due to incomplete annotation, isoforms prediction methods should be used to provide a new set of alternative isoforms. Otherwise, there is no way to extrapolate the expression levels, unless there are evidences for some components of $\mathbf{x}$ to be null. In the following, therefore, we will suppose that $rank(\mathbf{M}) = n$ holds.

In our model it is realistic to take into account the fact that different $n \times n$ full-rank submatrices of $\mathbf{M}$ may lead to different solutions. The idea to deal with this kind of situations is to define an objective function $F$, on the set of exons, suitable to jointly represent (i) the relevance of each exon within the gene, and (ii) the balance among coverages estimates within sets of exons. Given a set of indexes $H \subseteq \{1, \dots, m\}$, we identify $\{E_h\}_{h \in H}$ as an *optimal set of exons* if

- it maximises $F$,
- it completely characterizes the set of isoforms $\mathcal{I}$, and
- has minimal cardinality.

The idea is to define $\mathbf{x}$ as the solution of the following square sub-system

$$\mathbf{M}_h \mathbf{x} = c_h \qquad h \in H, \tag{7.3.2}$$

whose existence and uniqueness is guaranteed by the hypotheses on $\mathbf{M}$ and by the minimality of $H$. As a consequence, once $H$ is selected, $\mathbf{x}$ is completely determined and we can define the problem of finding the expression levels as follows.

**Definition 7.3.1** *Let $G$ be a gene and let $\mathbf{Mx} = \mathbf{c}$ be the linear system associated to it, with $\mathbf{M}$ defined as in (7.1.1) and $\mathbf{c} \in (\mathbb{R}^+)^m$. Let $F\colon \{1, \dots, m\} \to \mathbb{R}$. Suppose that $rank(\mathbf{M}) = n$.*

*We define $\mathcal{H}$ as the collection of sets of linearly independent rows of $M$, namely:*

$$\mathcal{H} := \{H \subseteq \{1, \dots, m\} \mid |H| = n \wedge rank([\mathbf{M}_h]_{h \in H}) = n\}. \tag{7.3.3}$$

*An* optimal subset of exons *for $G$ is $\{E_h\}_{h \in H}$, where $H$ is a solution of*

$$\begin{aligned} &\text{maximize} \quad F(H) \\ &H \in \mathcal{H}\,. \end{aligned} \tag{7.3.4}$$

When a set $H$ satisfying the above definition is found, the solution $\mathbf{x}$ of the linear system $[\mathbf{M}_h]_{h\in H}\mathbf{x} = [c_h]_{h\in H}$ will be an assignment of the expression level to each isoform.

Notice that, by using an additive function $F(\cdot)$, a solution of Problem 7.3.4 can be obtained by adding elements to $H$ iteratively, starting from the empty set and checking at each step whether $rank([\mathbf{M}_h]_{h\in H\cup\{i\}}) > rank([\mathbf{M}_h]_{h\in H})$, where $i$ is the element to be added.

**Proposition 7.3.2** *The problem $\mathcal{P}$ stated in Def. 7.3.1 is polynomial in $m$.*

*Proof.* As already observed, problem $\mathcal{P}$ can be solved with a greedy algorithm, that is, by firstly sorting the elements of $\{1,\dots,m\}$ by non-decreasing values of $F(i)$ and then selecting a new index $i$ to be added to the current set $H$ iteratively, such that $rank([\mathbf{M}_h]_{h\in H\cup\{i\}}) > rank([\mathbf{M}_h]_{h\in H})$. The check to be done at each step can be performed with Gauss-Jordan elimination, whose complexity is $O(|H|^3) \in O(n^3)$. The ordering of $\{1,\dots,m\}$ is $O(m\log m)$ and the number of operations globally required by the iterative procedure is $O(mn^3)$. Hence the complexity of solving $\mathcal{P}$ is $O(m(\log m+n^3)) \in O(m^4)$ as $n \le m$ from hypotheses. □

A possible definition for $F$ is a linear combination of two functions $w$ and $p$:

$$F(i) := aw(i) - bp(i) \tag{7.3.5}$$

where $a, b \in \mathbb{R}^+$. For each $i = 1,\dots,m$, $w(i)$ represents the weight of exon $E_i$, while $p(i)$ represents the penalty associated to $E_i$ with respect to other exons, in terms of coverage incompatibility.

For each exon $E_i$, we compute the standard deviation $\sigma_i$ from the average coverage $c_i$. Given $E_i$, a suitable definition for $w(i)$ should depend either on the exon length $L(E_i)$, on the coefficient of variation $\frac{\sigma_i}{c_i}$, and, possibly, on the standard deviation $\sigma_i$. More precisely, $w(i)$ should be directly proportional to $L(E_i)$ and inversely proportional to $\frac{\sigma_i}{c_i}$.

The penalty value $p(i)$ should take into account the differences in (mean) coverage among exons that are related to each other, weighted accordingly. Let us denote with $\mathcal{I}(E)$ the set of isoforms $E$ belongs to. If $E_i$ and $E_j$ are such that $\mathcal{I}(E_i) \subseteq \mathcal{I}(E_j)$, then one should expect that $c_i \le c_j$. Otherwise, the difference $c_i - c_j$ represents an inconsistence and should appear in the penalty term. Either cases $\mathcal{I}(E_i) \subset \mathcal{I}(E_j)$, $\mathcal{I}(E_i) \supset \mathcal{I}(E_j)$, and $\mathcal{I}(E_i) = \mathcal{I}(E_j)$ can be all considered, or some of them solely may be taken into account in $p(i)$. An exon $E$ will have high penalty if heavy exons are inconsistent with $E$. If average inconsistencies measured for $E_i$ and $E_j$ are equal to each other, and $E_i$ is related with more exons compared to $E_i$, then $p(i)$ should be slightly lesser than $p(j)$.

### 7.3.3 A heuristic algorithm based on exact solution of max-quality full-rank square sub-system

As stated in Def. 7.3.1, one can think of providing a solution of the problem of determining the expression levels of alternative transcripts with a parsimony approach. However, several considerations should be done in order to provide solutions that fit well with the biological context. Just to mention a few ones: (i) only not-too-short exons should be used for the computation of cumulative expression levels, (ii) exons with expression level equal to zero must be treated in special way, and (iii) high coverages on 5' side may be due to cDNA production artifacts.

In particular task (iii) should be treated very carefully: on the one hand, point (ii) suggests not to discard low expression levels in spite of higher quality exons whose coverage is zero; on the other hand, several studies pointed out the need of treating expression also in light of the location the reads are coming from. A tool for transcripts abundances estimation should meet both these task, for instance Guttman and colleagues [49] address this task by suitably designing lab experiments.

In our method, we firstly discard short exons from each gene's linear system. As most of *Arabidopsis thaliana* genes are slightly different from one another, discarding exons of length $< 30$ bp results into the impossibility of disambiguating $\sim 3\%$ of all genes. This is due to the fact that the exons discarded make the associated isoforms/exons matrix' rank to be lower than the maximum. Such a situation, however, would cause problam to any method whose goal is to compute expression levels for each alternative isoform separately.

In order to maximize sensitivity, we decided to adopt a heuristic that should avoid low covered transcripts to be seen as not expressed. The idea is that, if a set of exons belong to the same set of isoforms, that is, the set $\mathcal{I}(E)$ for some $E$, then at most one exon $E'$ such that $\mathcal{I}(E') = \mathcal{I}(E)$ will be selected for the square sub-system. Moreover we observed that, if such exon has both high quality and zero coverage, then the cumulative expression levels of all the isoforms of $\mathcal{I}(E)$ will be zero. Hence, we decided to discard all exons $E$ that have coverage 0, if and only if there exist $E'$ such that $\mathcal{I}(E') = \mathcal{I}(E)$ of coverage $> 0$. This choice would clearly increase the amount of false positives, but we also observed that, concerning the problem of expression levels prediction, missing expressed isoforms is a more serious problem than that of declaring as expressed isoforms that are not.

A problem arising from the combinatorial technique used to solve the linear system is the possibility to generate negative solutions. These are the effect of incompatibilities among exons coverages and may be corrected by slightly modifying the coverages within the neighborhood defined by (a multiple or a fraction of) the standard deviation. An incompatibility between two exons should result in an increase in the penalty value of the exons involved. The idea is to change the coverage and then re-compute penalties. An improvement will be measured as a decrease in the penalty value of the new system with respect to the previous one. Then, a solution computed on such a system should be better than the previous one (in terms of amount of negative components).

The method described above can be applied in two ways, that is, either on all solution, or only on those exhibiting negative components. In the former case, the benefit will consist in potentially improving *every* solution, but, as a drawback, the possibility of taking into account worse exons could lead to worse solutions. It is a matter of experimental results to decide which approach would work well: a uniform one, applied massively to all solutions, but with the risk of involving bad quality information that one should have wanted to disregarsd, or a particularized one, used on special cases only, but with the inherent impossibility of improving the large part on solutions quality.

### 7.3.4 Definition of objective functions

Several objective function of the form (7.3.5) can be defined. Concering the weight function $w(\cdot)$, we choose to emphasize exons length and coverage uniformity. As already mentioned, we exclude short exons (*i.e.*, shorter than a threshold value $L_{thr}$ for exons length) from the computation of the solution. Concerning coverage, we estimate the standard deviation, and we denote it with $\sigma_i$, from the average exons coverage obtained from RNA-Seq reads location (see Section 7.2.2).

In practice, we want the weight of exon $E_i$ to be direclty proportional with respect to its length $L(E_i)$ and inversely proportional with respect to the coverage standard deviation $\sigma_i$. Three definitions of $w(\cdot)$ satisfying the above requirements are as follows:

$$\begin{aligned} w^{(1)}(i) &= \frac{L(E_i)}{L_{\max}}\Big(1-\frac{\sigma_i}{c_i}\Big); \\ w^{(2)}(i) &= L(E_i)\Big(1-\frac{\sigma_i}{c_i}\Big)\frac{1}{\log(\sigma_i+2)}; \\ w^{(3)}(i) &= \log\Big(\frac{L(E_i)}{L_{thr}}\Big)\Big(1-\frac{\sigma_i}{c_i}\Big)\frac{1}{\log(\sigma_i+2)}, \end{aligned} \tag{7.3.6}$$

where $L_{\max}$ is the maximal exon length in the gene. If negative quantities occur, we set them to zero so that all weights are non-negative.

The penalty function $p(\cdot)$ is defined, for each exons, considering coverage consistency with other exons in the same gene. The idea is to put in relation $E_i$ with "comparable" exons. More precisely, we can say something about coverage relation among $E_i$ and $E_j$ only if one of the following holds: $\mathcal{I}(E_i) = \mathcal{I}(E_j)$, $\mathcal{I}(E_i) \subset \mathcal{I}(E_j)$, or $\mathcal{I}(E_i) \supset \mathcal{I}(E_j)$.

For each $i$, we define as $H_i$ the set of indexes in $\{i, \dots, m\}$ such that exon $E_h$ is "related to" exon $E_i$, for $h \in H_i$. We also denote with $w(H_i)$ the sum of the weights of $E_h$, for all $h \in H_i$. A possible way to define the penalty $p(i)$ for exon $E_i$ is to put in relation $c_i$ with $c_h$, for each $h \in H_i$. Set $H_i$ can be defined in different ways, depending on the choice to consider, for two exons $E_i$ and $E_j$, either equality, inclusion, or both relations among $\mathcal{I}(E_i)$ and $\mathcal{I}(E_j)$. Depending on the choice of taking into account either the cardinality of $|H_i|$ or the coverage standard deviation, we define two penalty functions:

$$\begin{aligned} p^{(1)}(i) &= \frac{1}{w(H_i)}\sum_{h\in H_i} w_h(|c_i - c_h| - \sigma_i); \\ p^{(2)}(i) &= \frac{2^{-|H_i|}}{w(H_i)}\sum_{h\in H_i} w_h|c_i - c_h|. \end{aligned} \tag{7.3.7}$$

Quantities $p^{(2)}(i)$ were defined in 3 different ways, by changing the set $H_i$.

Obviously, there is a wide variety of reasonable objective functions to be used. We mentioned here a few ones that aim at characterising exons "as such", as well as in relation with other exons within the same gene. This characterisation is important in order to be able to select a good set of exons, which, in turn, are crucial for good expression levels estimation.

## 7.4 Results

Different objective functions were tested, varying both weight and penalty functions, and the coefficient of their linear combination. More precisely, by combining different choices of $w(\cdot)$ and $p(\cdot)$, based on Equations 7.3.6 and 7.3.7, we get 12 different objective functions. For each of them, we tried 13 couples of values for parameters $a$ and $b$.

The statistics showed in the following are average results obtained using different objective functions, on simulated RNA-Seq reads libraries. Some comparisons are reported with the tool Cufflinks, which performs both reads alignment and transcripts abundances estimation.

### 7.4.1 Datasets

We report here the results obtained on five RNA-Seq experiments simulated using Flux Simulator [57] on *Arabidopsis thaliana* genome (`TAIR10_cdna_20101214_updated`) and using annotation `TAIR10_GFF3_genes.gff` (see [65]). For each experiment, we generated paired reads datasets with read length 100 bp and insert size $\sim 250$ bp. All parameters for simulation were set to the default values, except for the exponent of the modified Zipf's law employed to generate transcripts expression profile [47]. More precisely, we set parameter `EXPRESSION_K` to five different values comprised between 0.6 and 0.9, as reported in the Supplementary Material [47] (namely, $-0.6$, $-0.65$, $-0.7$, $-0.75$, and $-0.8$, respectively). We simulated expression, library construction, and sequencing. In this way, we know exactly both the expression level of each isoform, and the location of each read in the transcript it has been extracted from.

As *A. thaliana* genes exhibit splicing patterns that are completely different from that of mammalian genes, we decided to perform experiment on human data as well [66, 78]. We expect to find differences in both isoforms number and alternative splicing complexity. Even though we expect much higher isoforms numerosity for a mammalian gene, we believe that our method would behave better on them, thanks to the alternative splicing complexity. It is quite difficult for us to output correct values for genes with a number of exons that is extremely higher than the number of isoforms: in our case, it would translate into a selection of very few exons (and corresponding cumulative expression levels) from a very larger set.

### 7.4.2 Results evaluation

In order to provide a comparison with existing tools, we launched Cufflinks [165] on the datasets described above, after alignment performed with TopHat [164].

We decided to quantify the results using the RPKM metric. Transcripts abundances was computed by firstly extracting the reads provenance, provided by Flux Simulator, and then calculating isoforms coverages with the method described in Section 7.2.2. Our pipeline outputs a mean coverage value for each isoform, so we needed to convert the output to RPKM. Let $T$ be a transcript. By denoting with $L$ the (average) read length, with $N$ the library size, and with $c$ the coverage of $T$, and recalling the definition of RPKM (see Equation 1.2.2, we have:

$$\text{RPKM} = \frac{m10^9}{NT}, \; c = \frac{mL}{T} \;\Rightarrow\; \text{RPKM} = \frac{c10^9}{LN}. \tag{7.4.1}$$

Clearly, the normalization obtained using the above equation introduces imprecisions in the results, due, for instance, to the fact that read lengths are not all the same within the library. In Figure 7.3 an overview of the dataset type is reported. The number of low expressed transcripts (*i.e.*, with RPKM ranging in $[1, 4[$) is the largest one and hence require a highly sensitive method for their quantification. Figure 7.4 depicts the response of our pipeline to different expression levels. The mean error rate $\bar{e}$ is defined as $100 \cdot |R - \hat{R}|/R$, where $R$ is the ground truth and $\hat{R}$ is the estimated RPKM. Values reported for $\bar{e}$ are computed by taking the average on the results obtained with the different objective functions. As espected, it drops as coverage increases. On Figure 7.5 the number of isoforms is reported as a function of the mean error rate. Notice that the high abundance

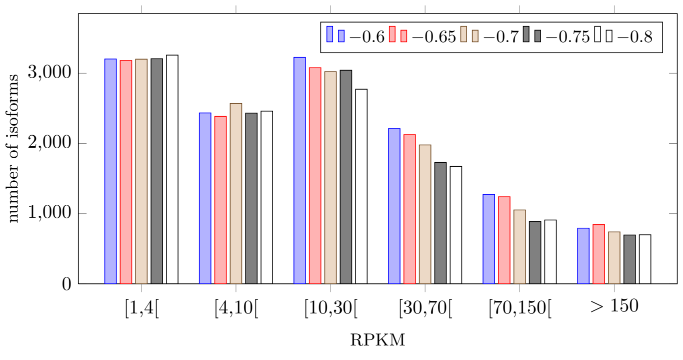


**Figure 7.3:** Number of isoforms as a function of RPKM value. Histograms refer to five experiments on simulated data. Parameter **EXPRESSION_K** ranges in $\{-0.6, -0.65, -0.7, -0.75, -0.8\}$. The higher **EXPRESSION_K**, the more variable transcripts abundancies of alternative isoforms referred to the same gene.

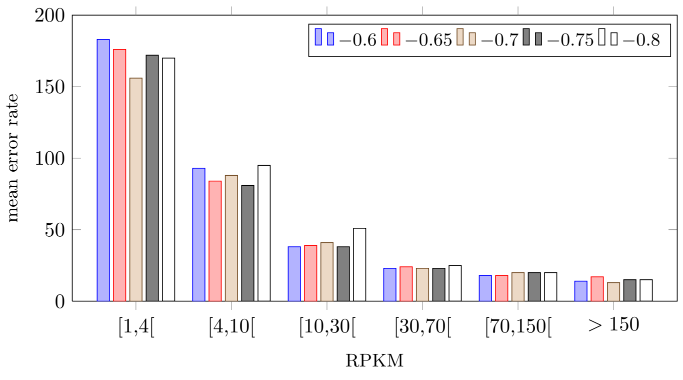


**Figure 7.4:** Error rate as a function of RPKM value. Histograms refer to five experiments on simulated data. Parameter **EXPRESSION_K** ranges in $\{-0.6, -0.65, -0.7, -0.75, -0.8\}$. Error rate refers to the percentage of error in predicted RPKM value with respect to ground truth.

of isoforms are estimated with an error rate $< 10\%$. It is proper to say here that some solutions cannot be found by our method as negative values for expression levels are reported. Their amount is $\sim 850$ over $40,000$ isoforms and they are due to imprecision in coverage estimates.

A comparison with Cufflinks performances is showed in Table 7.1. Our method introduces fewer false positives, however we cannot reach Cufflink's ability to capture down-regulated isoforms.

**Table 7.1:** FP and FN comparison statistics. Values for Cufflinks are computed on coverage and FPKM values reported on FPKM tracking file.

| | Our pipeline | | Cufflinks | |
|---|---|---|---|---|
| $k$ | FP | FN | FP | FN |
| −0.60 | 1409 | 6967 | 2938 | 861 |
| −0.65 | 1391 | 7121 | 2902 | 837 |
| −0.70 | 1294 | 7308 | 2868 | 839 |
| −0.75 | 1340 | 6919 | 2959 | 860 |
| −0.80 | 1117 | 7366 | 2756 | 914 |

Statistical significance of expression levels measurements is difficult to achieve, also because of the non-standardized metrics used to represent them. Indeed, RPKM is reported in different ways (*e.g.*, the count of reads mapping onto a trascript is normalized either to the total number of mapped reads or to the library size); moreover, Cufflinks reports FPKM instead of RPKM (using fragments instead of reads). A different approach can be that of evaluating expression levels with respect to coverage. In this context, at least for

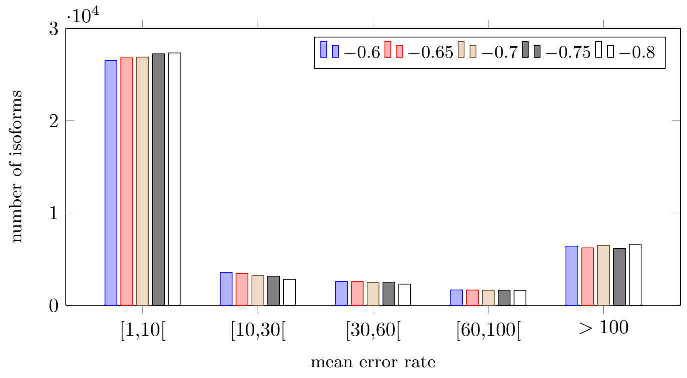


**Figure 7.5:** The number of isoforms as a function of mean error rate. Histograms refer to five experiments on simulated data. Parameter `EXPRESSION_K` ranges in $\{-0.6, -0.65, -0.7, -0.75, -0.8\}$. Notice that error rate is extremely high for low expressed transcripts, that explains the increase at error rate $> 100$.

methods that report it, the standard deviation from the mean coverage can be used as an indication of the seriousness of expression levels estimation errors. This reasoning can be used when exact expression levels are known. The idea is that, if exact expression is $x$ and its standard deviation is $\sigma$, then the goodness of the estimated expression $\hat{x}$ can be related to either the distance from $x$, and to the fact whether $\hat{x}$ belongs to $[x - k\sigma, x + k\sigma]$ or not (where $k$ is a parameter that can be varied in order to set the evaluation more or less strict).

Another statistic that can be computed is the consistence with global gene expression, that is, compute the error $|\sum_i \hat{x}_i - \sum_i x_i|$. In case the gene has a complex structure, such a statistics can assess whether the method succeeds at least in computing the gene expression accurately. We can also think of defining a sort of *entropy* of the splicing pattern of a gene. In our formulation, entropy $= 0$ should be assigned to a gene with a single exon; the highest entropy, instead, should be assigned to a gene with several isoforms, each of them differing a little from one another.

An indirect method for evaluating the results consists in looking at the variation in the expression profiles of the isoform within a gene. Thorough studies pointed out that, at least for human genes, very few isoforms are really important within a gene [33,44]. Hence, a possible way to compute indirect and unbiased statistics on expression level estimation methods may consist in evaluating the results with an eye of what realistic scenarios look like. Also, this point of view applied to experiments on simulated reads may help in evaluating a method exactly but, in some sense, less artificially.

## 7.5 Summary and conclusions

We developed a new annotation-guided method for quantification of isoforms expression levels. The elements of novelty of our approach lie on a layout-guided alignment technique, and to a combinatorial algorithm that finds a heuristic solution to the linear system associated to a gene by reducing the problem to that of finding an optimal solution to a high-quality sub-problem.

The pipeline we built consists in the following steps:

1) inserts assembly with GapFiller;

2) perform trusted contigs alignment against the reference genome;

3) compute exon coverages;

4) define the linear system for each gene and solve it.

Concerning step 1), we store insert-contigs layout (see 6.1.1). Alignment of insert-contigs performed in step 2) is then used to retrieve reads location on the genome, thanks to layout information. In order not to introduce coverage artifacts, at step 3) a single alignment for each read is retained, moreover, non-uniformity of coverage is reduced by neglecting exon subsequences exhibiting peaks in coverage. At step 4), a linear system is defined for each gene, following the definition of exons reported in Section 7.3.1 and employing coverage values computed at previous steps.

The last part of our pipeline allow for further development ad refinement. For instance, we can try different and more sophisticated objective functions. Most importantly, we must define a heuristic aimed at correcting solutions that exhibit negative components: this can be done by computing a set of solutions—each corresponding to a square sub-system—instead of a single one, or by refining coverage estimates within the neighborhood defined by coverage standard deviation. We believe that adding combinatorial information on the *biological* meaning of the problem is a key point that, possibly, can be refined at a second stage of RNA-Seq data analysis with suitable statistical methods.

# Conclusions

In this thesis we studied several problems concerned with the analysis of biological sequences. Paired reads play a key role in the analyses, allowing to connect contiguous stretches of DNA or RNA, validate an assembly, aid reads alignment with respect to a reference, and find structural variations among genomes. Sharpening pair information with insert assembly, makes GapFiller a useful tool for a number of biological applications, ranging from genomics to transcriptomics. The availability of the contiguous nucleotide sequence between two pairs allows, in fact, more accurate, *i.e.*, specific, predictions. To reach this goal, sequence correctness is of utmost importance and we adopt refined heuristic that allowed to get a certified output that does not need validation.

The application of GapFiller to *de novo* assembly as a preprocessing step yielded results comparable to that obtained with state-of-the-art assemblers on public real datasets. Moreover, sequence correctness has been extensively tested on simulated data and the result was that the false positives rate was always almost negligible.

Using paired reads to enhance reads alignment against a reference genome is a widely employed technique that allow to disambiguate multiple mapping locations and recognize pair-incompatible alignments. Concerning the task of mapping RNA-Seq reads against the reference genome, insert-size information is useless when paired reads come from different exons, moreover, gaps occurring within reads spanning splicing junctions make alignment harder and resource-demanding.

Concerning applications to RNA-Seq data analysis, we employ insert information to both guide RNA-Seq reads alignment, employing the layout of reconstructed inserts, and recognize pair-compatible alignments. We deeply analyzed the ability of our alignment method to annotate splicing junctions, without requiring prior annotation information, but using instead a combination of inserts alignment and dinucleotides check on putative splicing sites. By aligning inserts and then retrieving reads position on the genome, the method turns out to be the highest specific one among the set of tools we tested, and it has shown high alignment accuracy by correctly recognizing spliced and unspliced reads. Application of our RNA-Seq reads alignment method to the computation of expression levels of alternative transcripts is outlined within a linear system that represents the exon-isoform structure of each gene. Still focusing on accuracy, we proposed a parsimony approach to the problem that employs only the necessary information on exons coverage. To this aim, we introduced a general method to compute such information on a set of high-quality exons. Even though this last method needs further improvement in the detection of low expressed isoforms, we observed good precision in the computation of RPKM values. Moreover, the proposed method is very general, hence it can be extended towards various directions.

Summarizing the main findings, we stress again the importance of taking advantage of information coming from data to the highest possible level. We focused on the massive usage of paired reads, and found out that this approach allows to obtain accurate predictions, by implicitly eliminating what is inconsistent (*i.e.*, pair-incompatible) with the data.

# Bibliography

[1] G. P. Alamancos, E. Agirre, and E. Eyras. Methods to study splicing from high-throughput RNA Sequencing data.

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

[3] S. Altschul, W. Gish, W. Miller, E. Myers, and D. B. l. a. s. t. Lipman. Basic local alignment search tool. *J. Mol. Biol.*, 215:403–410, 1990.

[4] A. Ameur, A. Wetterbom, L. Feuk, and U. Gyllensten. Global and unbiased detection of splice junctions from RNA-seq data. *Genome Biology*, 11(3):R34, 2010.

[5] S. Anders. Analysing rna-seq data with the "deseq" package. *Mol Biol*, pages 1–17, 2010.

[6] P. N. Ariyaratne and W. K. Sung. Pe-assembler: de novo assembler using short paired-end reads. *Bioinformatics*, 27(2):167–74, 2011.

[7] K. F. Au, H. Jiang, L. Lin, Y. Xing, and W. H. Wong. Detection of splice junctions from paired-end RNA-seq data by SpliceMap. *Nucleic Acids Res.*, 38(14):4570–8, 2010.

[8] H. Bao, Y. Xiong, H. Guo, R. Zhou, X. Lu, Z. Yang, Y. Zhong, and S. Shi. Mapnext: a software tool for spliced and unspliced alignments and snp detection of short sequence reads. *BMC genomics*, 10(Suppl 3):S13, 2009.

[9] S. Batzoglou, D. B. Jaffe, K. Stanley, J. Butler, S. G. Gnerre, E. Mauceli, B. B. Berger, J. P. Mesirov, and E. S. Lander. Arachne: A whole-genome shotgun assembler. *Genome Res*, 12:177–189, 2002.

[10] D. Bentley. Whole-genome re-sequencing. *Curr. Opin. Genet. Dev.*, 16:545–552, 2006.

[11] D. R. Bentley, S. Balasubramanian, H. P. Swerdlow, G. P. Smith, J. Milton, C. G. Brown, K. P. Hall, D. J. Evers, C. L. Barnes, H. R. Bignell, et al. Accurate whole human genome sequencing using reversible terminator chemistry. *Nature*, 456(7218):53–59, 2008.

[12] I. Birol, S. D. Jackman, C. B. Nielsen, J. Q. Qian, R. Varhol, G. Stazyk, R. D. Morin, Y. Zhao, M. Hirst, J. E. Schein, D. E. Horsman, J. M. Connors, R. D. Gascoyne, M. A. Marra, and S. J. Jones. De novo transcriptome assembly with ABySS. *Bioinformatics*, 25:2872–2877, 2009.

[13] 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):1519–33, 2010.

[14] P. Bonizzoni, R. Rizzi, and G. Pesole. Aspic: a novel method to predict the exon-intron structure of a gene that is optimally compatible to a set of transcript sequences. *BMC Bioinformatics*, 6:244, 2005.

[15] K. R. Bradnam, J. N. Fass, A. Alexandrov, P. Baranay, M. Bechner, İ. Birol, S. Boisvert10, J. A. Chapman, G. Chapuis, R. Chikhi, et al. Assemblathon 2: evaluating de novo methods of genome assembly in three vertebrate species. *arXiv preprint arXiv:1301.5406*, 2013.

[16] A. Bräutigam, T. Mullick, S. Schliesky, and A. P. M. Weber. Critical assessment if assembly strategies for non-model species mRNA-Seq data and application of next-generation sequencing to the comparison of $C_3$ and $C_4$ species. *Journal of Experimental Botany*, 62(9):3093–3102, 2011.

[17] D. W. Bryant, R. Shen, H. D. Priest, W.-K. Wong, and T. C. Mockler. Supersplat—spliced rna-seq alignment. *Bioinformatics*, 26(12):1500–1505, 2010.

[18] 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.

[19] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. 1994.

[20] J. Butler, I. MacCallum, M. Kleber, I. A. Shlyakhter, M. K. Belmonte, E. S. Lander, C. Nusbaum, and D. B. Jaffe. ALLPATHS: de novo assembly of whole-genome shotgun microreads. *Genome Res.*, 18:810–820, 2008.

[21] M. O. Carneiro, C. Russ, M. G. Ross, S. B. Gabriel, C. Nusbaum, and M. A. DePristo. Pacific biosciences sequencing technology for genotyping and variation discovery in human data. *BMC Genomics*, 13:375, Aug. 2012.

[22] M. Chaisson, P. Pevzner, and H. Tang. Fragment assembly with short reads. *Bioinformatics*, 20:2067–2074, 2004.

[23] J. A. Chapman, I. Ho, S. Sunkara, S. Luo, G. P. Schroth, and D. S. Rokhsar. Meraculous: De novo genome assembly with short paired-end reads. *PLoS ONE*, 6(8):e23501, 2011.

[24] K. Chen, J. W. Wallis, M. D. McLellan, D. E. Larson, J. M. Kalicki, C. S. Pohl, S. D. McGrath, M. C. Wendl, Q. Zhang, D. P. Locke, X. Shi, R. S. Fulton, T. J. Ley, R. K. Wilson, L. Ding, and E. R. Mardis. Breakdancer: an algorithm for high-resolution mapping of genomic structural variation. *Nature methods*, 6(9):677–681, 2009.

[25] B. Chevreux, T. Pfisterer, B. Drescher, A. J. Driesel, W. E. G. Müller, T. Wetter, and S. Suhai. Using the miraEST assembler for reliable and automated mRNA transcript assembly and SNP detection in sequenced ESTs. *Genome Res.*, 14:1147–1159, 2004.

[26] B. Chevreux, T. Wetter, and S. Suhai. Genome sequence assembly using trace signals and additional sequence information. In *Computer Science and Biology: Proceedings of the German Conference on Bioinformatics GCB*, page 45–56, 1999.

[27] C.-S. Chin. The origin of the Haitian cholera outbreak strain. *N. Engl. J. Med.*, 364:33–42, 2011.

[28] J. Cocquet, A. Chong, G. Zhang, and R. A. Veitia. Reverse transcriptase template switching and false alternative transcripts. *Genomics*, 88:127–131, 2006.

[29] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2 edition, 2001.

[30] F. De Bona, S. Ossowski, K. Schneeberger, and G. Rätsch. Optimal spliced alignments of short sequence reads. *BMC Bioinformatics*, 9(Suppl 10):O7, 2008.

[31] M. de la Bastide and W. R. McCombie. *Assembling genomic DNA sequences with PHRAP*, volume Unit 11.4., chapter Chapter 11. 2007.

[32] M. T. Dimon, K. Sorber, and J. L. DeRisi. HMMSplicer: a tool for efficient and sensitive discovery of known and novel splice junctions in RNA-Seq data. *PLoS One*, 5(11):e13875, 2010.

[33] S. Djebali, C. A. Davis, A. Merkel, A. Dobin, T. Lassmann, A. M. Mortazavi, A. Tanzer, J. Lagarde, W. Lin, F. Schlesinger, et al. Landscape of transcription in human cells. *Nature*, 489(7414):101, 2012.

[34] A. Dobin, C. A. Davis, F. Schlesinger, J. Drenkow, C. Zaleski, S. Jha, P. Batut, M. Chaisson, and T. R. Gingeras. STAR: ultrafast universal RNA-seq aligner. *Bioinformatics*, 29(1):15–21, 2013.

[35] J. Dohm, C. Lottaz, T. Borodina, and H. Himmelbauer. Substantial biases in ultra-short read data sets from high-throughput DNA sequencing. *Nucleic Acids Res.*, 36:e105, 2008.

[36] 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*, 17:1697–1706, 2007.

[37] D. Earl, K. Bradnam, J. S. John, A. Darling, D. Lin, J. Fass, H. O. K. Yu, V. Buffalo, D. R. Zerbino, M. Diekhans, et al. Assemblathon 1: A competitive assessment of de novo short read assembly methods. *Genome research*, 21(12):2224–2241, 2011.

[38] S. R. Eddy. Hidden markov models. *Current opinion in structural biology*, 6(3):361–365, 1996.

[39] J. Eid, A. Fehr, J. Gray, K. Luong, J. Lyle, G. Otto, P. Peluso, D. Rank, P. Baybayan, B. Bettman, et al. Real-time dna sequencing from single polymerase molecules. *Science*, 323(5910):133–138, 2009.

[40] P. G. Engström, T. Steijger, B. Sipos, G. R. Grant, A. Kahles, G. Rätsch, N. Goldman, T. J. Hubbard, J. Harrow, R. Guigó, et al. Systematic evaluation of spliced alignment programs for rna-seq data. *Nature methods*, 2013.

[41] N. A. Fonseca, J. Rung, A. Brazma, and J. C. Marioni. Tools for mapping high-throughput sequencing data. *Bioinformatics*, 28(24):3169–3177, 2012.

[42] M. B. Gerstein, C. Bruce, J. S. Rozowsky, D. Zheng, J. Du, J. O. Korbel, O. Emanuelsson, Z. D. Zhang, S. Weissman, and M. Snyder. What is a gene, post-encode? history and updated definition. *Genome research*, 17(6):669–681, 2007.

[43] S. Gnerre, I. MacCallum, D. Przybylski, F. J. Ribeiro, J. N. Burton, B. J. Walker, T. Sharpe, G. Hall, T. P. Shea, S. Sykes, A. M. Berlin, D. Aird, M. Costello, R. Daza, L. Williams, R. Nicol, A. Gnirke, C. Nusbaum, E. S. Lander, and D. B. Jaffe. High-quality draft assemblies of mammalian genomes from massively parallel sequence data. *Proceedings of the National Academy of Sciences*, 108(4):1513–1518, Dec. 2010.

[44] M. Gonzalez-Porta, A. Frankish, J. Rung, J. Harrow, and A. Brazma. Transcriptome analysis of human tissues and cell lines reveals one dominant transcript per gene. *Genome Biology*, 14:R70, 2013.

[45] M. G. Grabherr, B. J. Haas, M. Yassour, J. Z. Levin, D. A. Thompson, I. Amit, X. Adiconis, L. Fan, R. Raychowdhury, Q. Zeng, Z. Chen, E. Mauceli, N. Hacohen, A. Gnirke, N. Rhind, F. di Palma, B. W. Birren, C. Nusbaum, K. Lindblad-Toh, N. Friedman, and A. Regev. Full-length transcriptome assembly from RNA-seq data without a reference genome. *Nature biotechnology*, 29(7):644–652, 2011.

[46] G. R. Grant, M. H. Farkas, A. D. Pizarro, N. F. Lahens, J. Schug, B. P. Brunk, C. J. Stoeckert, J. B. Hogenesch, and E. A. Pierce. Comparative analysis of rna-seq alignment algorithms and the rna-seq unified mapper (rum). *Bioinformatics*, 27(18):2518–2528, 2011.

[47] T. Griebel, B. Zacher, P. Ribeca, E. Raineri, V. Lacroix, R. Guigó, and M. Sammeth. Modelling and simulating generic RNA-Seq experiments with the flux simulator. *Nucleic Acids Res.*, 40(20):10073–83, 2012.

[48] M. Guttman, M. Garber, J. Z. Levin, J. Donaghey, J. Robinson, X. Adiconis, L. Fan, M. J. Koziol, A. Gnirke, C. Nusbaum, et al. Ab initio reconstruction of transcriptomes of pluripotent and lineage committed cells reveals gene structures of thousands of lincRNAs. *Nature biotechnology*, 28(5):503, 2010.

[49] M. Guttman, M. Garber, J. Z. Levin, J. Donaghey, J. Robinson, X. Adiconis, L. Fan, M. J. Koziol, A. Gnirke, C. Nusbaum, J. L. Rinn, E. S. Lander, and A. Regev. Ab initio reconstruction of cell type-specific transcriptomes in mouse reveals the conserved multi-exonic structure of lincRNAs. *Nat. Biotech.*, 28:503–510, 2010.

[50] M. Heber, Steffen abd Alekseyec, S.-H. Sze, H. Tang, and P. A. Pevzner. Splicing graphs and EST assembly problem. *Bioinformatics*, 18(Suppl. 1):S181–S188, 2002.

[51] D. Hernandez, P. Francois, L. Farinelli, M. Osteras, and S. J. De novo bacterial genome sequencing: millions of very short reads assembled on a desktop computer. *Genome Res*, 18:802–809, 2008.

[52] M. S. Hossain, N. Azimi, and S. Skiena. Crystallizing short-read assemblies around seeds. *BMC Bioinformatics*, 10(Suppl 1):S16, 2009.

[53] B. E. Howard and S. Heber. Towards reliable isoform quantification using rna-seq data. *BMC Bioinformatics*, 11(Suppl 3):S6, 2010.

[54] http://assemblathon.org/.

[55] http://blast.ncbi.nlm.nih.gov/.

[56] http://cnag.bsc.es/.

[57] http://flux.sammeth.net/.

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

[59] http://genome.ucsc.edu/encode/.

[60] http://samtools.sourceforge.net/.

[61] http://sourceforge.net/projects/gapfiller/.

[62] http://web.ornl.gov/sci/techresources/Human_Genome/project.

[63] http://www.454.com.

[64] http://www.appliedbiosystems.com.

[65] http://www.arabidopsis.org/.

[66] http://www.ebi.ac.uk/gxa/.

[67] http://www.ensembl.org.

[68] http://www.gencodegenes.org/.

[69] http://www.gencodegenes.org/rgasp.

[70] http://www.genome.umd.edu/masurca.html.

[71] http://www.illumina.com.

[72] http://www.ncbi.nlm.nih.gov/.

[73] http://www.sanger.ac.uk/resources/software/smalt/.

[74] http://www.vmatch.de/.

[75] S. Huang, J. Zhang, R. Li, W. Zhang, Z. He, T.-W. Lam, Z. Peng, and S.-M. Yiu. SOAPsplice: genome-wide ab initio detection of splice junctions from RNA-Seq data. *Frontiers in genetics*, 2:46, 2011.

[76] D. H. Huson, K. Reinert, and E. W. Myers. The greedy path-merging algorithm for contig scaffolding. *Journal of the ACM (JACM)*, 49(5):603–615, 2002.

[77] W. R. Jeck, J. A. Reinhardt, D. A. Baltrus, M. T. Hickenbotham, V. Magrini, E. R. Mardis, J. L. Dangl, and C. Jones. Extending assembly of short DNA sequences to handle error. *Bioinformatics*, 23:2942–2944, 2007.

[78] M. Kapushesky, I. Emam, E. Holloway, P. Kurnosov, A. Zorin, J. Malone, G. Rustici, E. Williams, H. Parkinson, and A. Brazma. Gene expression atlas at the European bioinformatics institute. *Nucleic Acids Res.*, 38:D690–8, 2010.

[79] R. Karp and M. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Develop.*, 31(2):249–260, 1987.

[80] D. R. Kelley, M. C. Schatz, and S. L. Salzberg. Quake: quality-aware detection and correction of sequencing errors. *Genome biology*, 11(11):R116, Nov. 2010.

[81] S. Kerkstra. Transcription-initiation complexes and their role in cell behavior. 2007.

[82] D. E. Knuth. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms.* Addison-Wesley, 1973.

[83] E. V. Koonin. Does the central dogma still stand? *Biology Direct*, 7:27, 2012.

[84] S. Koren, M. C. Schatz, B. P. Walenz, J. Martin, J. T. Howard, G. Ganapahty, Z. Wang, D. A. Rasko, W. R. McCombie, E. D. Jarvis, and A. M. Pillippy. Hybrid error correction and *de novo* assembly of single-molecule sequencing reads. *Nature Biotechnology*, 30(7):693–700, 2012.

[85] I. Korf. Gene finding in novel genomes. *BMC Bioinformatics*, 5(1):59, 2004.

[86] S. Kumar and M. L. Blaxter. Comparing de novo assemblers for 454 transcriptome data. *BMC Genomics*, 11:571, 2010.

[87] S. Kurtz, A. Phillippy, A. L. Delcher, M. Smoot, M. Shumway, C. Antonescu, and S. L. Salzberg. Versatile and open software for comparing large genomes. *Genome Biology*, 5:R12, 2004.

[88] E. S. Lander and M. S. Waterman. Genomic mapping by fingerprinting random clones: A mathematical analysis. *Genomics*, 2:231–239, 1988.

[89] B. Langmead, C. Trapnell, M. Pop, S. L. Salzberg, et al. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biol*, 10(3):R25, 2009.

[90] B. Li, V. Ruotti, R. M. Stewart, J. A. Thomson, and C. N. Dewey. Rna-seq gene expression estimation with read mapping uncertainty. *Bioinformatics*, 26(4):493–500, 2010.

[91] H. Li and R. Durbin. Fast and accurate short read alignment with burrows–wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[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, J. Ruan, and R. Durbin. Mapping short dna sequencing reads and calling variants using mapping quality scores. *Genome research*, 18(11):1851–1858, 2008.

[94] 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.

[95] R. Li, C. Yu, Y. Li, T.-W. Lam, S.-M. Yiu, K. Kristiansen, and J. Wang. Soap2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009.

[96] R. Li, H. Zhu, J. Ruan, W. Qian, X. Fang, Z. Shi, Y. Li, S. Li, G. Shan, K. Kristiansen, S. Li, H. Yang, J. Wang, and J. Wang. De novo assembly of human genomes with massively parallel short read sequencing. *Genome research*, 20(2):265–72, Feb. 2010.

[97] W. Li, J. Feng, and T. Jiang. IsoLasso: a LASSO regression approach to RNA-Seq based transcriptome assembly. *J Comput Biol.*, 18(11):1693–707, Nov. 2011.

[98] Y. Li, H. Li-Byarlay, P. Burns, M. Borodovsky, G. E. Robinson, and J. Ma. True-sight: a new algorithm for splice junction detection using rna-seq. *Nucleic acids research*, 41(4):e51–e51, 2013.

[99] Y. Liao, G. K. Smyth, and W. Shi. The Subread aligner: fast, accurate and scalable read mapping by seed-and-vote. *Nucleic acids research*, 41(10):e108–e108, 2013.

[100] R. Lindner and C. C. Friedel. A comprehensive evaluation of alignment algorithms in the context of RNA-Seq. *PLoS ONE*, 7(12):e52403, 12 2012.

[101] L. Liu, Y. Li, S. Li, N. Hu, Y. He, R. Pong, D. Lin, L. Lu, and M. Law. Comparison of next-generation sequencing systems. *Journal of Biomedicine and Biotechnology*, 2012:251364, 2012.

[102] T. Liu, C.-H. Tsai, W.-B. Lee, and J.-H. Chiang. Optimizing information in next-generation-sequencing (ngs) reads for improving ¡italic¿de novo¡/italic¿ genome assembly. *PLoS ONE*, 8(7):e69503, 07 2013.

[103] S.-K. Lou, B. Ni, L.-Y. Lo, S. K.-W. Tsui, T.-F. Chan, and K.-S. Leung. Abmapper: a suffix array-based tool for multi-location searching and splice-junction mapping. *Bioinformatics*, 27(3):421–422, 2011.

[104] T. Magoč and S. L. Salzberg. FLASH: fast length adjustment of short reads to improve genome assemblies. *Bioinformatics (Oxford, England)*, 27(21):2957–63, Nov. 2011.

[105] C. A. Maher, N. Palanisamy, J. C. Brenner, X. Cao, S. Kalyana-Sundaram, S. Luo, I. Khrebtukova, T. R. Barrette, C. Grasso, J. Yu, R. J. Lonigro, G. Schroth, C. Kumar-Sinha, and A. M. Chinnaiyana. Chimeric transcript discovery by paired-end transcriptome sequencing. *Proc Natl Acad Sci*, 106(30):12353–12358, 2009.

[106] D. Maier and J. Storer. A note on the complexity of the superstring problem. In *12th Annual Conference on Information Science and Systems*, page 52–56, 1978.

[107] G. Marçais and C. Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics (Oxford, England)*, 27(6):764–770, Jan. 2011.

[108] E. R. Mardis. The impact of next-generation sequencing technology on genetics. *Trends in Genetics*, 24(3):133–141, Feb. 2008.

[109] E. R. Mardis. Next-generation dna sequencing methods. *Annu Rev Genomics Hum Genet.*, 9:387–402, 2008.

[110] M. Margulies, M. Egholm, W. E. Altman, S. Attiya, J. S. Bader, L. A. Bemben, J. Berka, M. S. Braverman, Y.-J. Chen, Z. Chen, et al. Genome sequencing in microfabricated high-density picolitre reactors. *Nature*, 437(7057):376–380, 2005.

[111] J. Martin, V. M. Bruno, Z. Fang, X. Meng, M. Blow, T. Zhang, G. Sherlock, M. Snyder, and Z. Wang. Rnnotator: an automated de novo transcriptome assembly pipeline from stranded RNA-Seq reads. *BMC genomics*, 11(1):663, 2010.

[112] A. J. Matlin, F. Clark, and C. W. J. Smith. Understanding alternative splicing: towards a cellular code. *Nature Reviews Molecular Cell Biology*, 6:386–398, may 2005.

[113] P. Medvedev and M. Brudno. Ab initio whole genome shotgun assembly with mated short reads. In *Research in Computational Molecular Biology*, pages 50–64. Springer, 2008.

[114] P. Medvedev, E. Scott, B. Kakaradov, and P. Pevzner. Error correction of high-throughput sequencing datasets with non-uniform coverage. *Bioinformatics*, 27(13):i137–i141, 2011.

[115] P. Medvedev, M. Stanciu, and M. Brudno. Computational methods for discovering structural variation with next-generation sequencing. *Nature methods*, 6:S13–S20, 2009.

[116] J. R. Miller, A. L. Delcher, S. Koren, E. Venter, B. P. Walenz, A. Brownley, J. Johnson, K. Li, C. Mobarry, and G. Sutton. Aggressive assembly of pyrosequencing reads with mates. *Bioinformatics*, 24(24):2818–2824, 2008.

[117] J. R. Miller, S. Koren, and G. Sutton. Assembly algorithms for next-generation sequencing data. *Genomics*, 95(6):315–327, June 2010.

[118] A. Mortazavi, B. A. Williams, K. McCue, L. Schaeffer, and B. Wold. Mapping and quantifying mammalian transcriptomes by RNA-Seq. *Nature Methods*, 5(7):621–8, July 2008.

[119] K.-R. Muller, S. Mika, G. Ratsch, K. Tsuda, and B. Scholkopf. An introduction to kernel-based learning algorithms. *Neural Networks, IEEE Transactions on*, 12(2):181–201, 2001.

[120] J. C. Mullikin and Z. Ning. The Phusion assembler. *Genome Res.*, 13(1):81–90, 2003.

[121] E. W. Myers. The fragment assembly string graph. *Bioinformatics*, 21(Suppl. 2):ii79–ii85, 2005.

[122] F. Nadalin, S. Scalabrin, and A. Policriti. Estimating expression levels of gene isoforms: a combinatorial approach based on accurate reads mapping via local transcriptome assembly. In *Proceedings on the 10th annual meeting of the Bioinformatics Italian Society (BITS 2013)*, Udine, May 2013.

[123] F. Nadalin, F. Vezzi, and A. Policriti. Gapfiller: a *de novo* assembly approach to fill the gap within paired reads. *BMC Bioinformatics*, 13(Suppl 14):S8, 2012.

[124] F. Nadalin, F. Vezzi, and A. Policriti. A multi-objective optimisation approach to the design of experiment in de novo assembly projects. In *Proceedings of the 2012 23rd International Workshop on Database and Expert Systems Applications*, DEXA '12, pages 213–217, Washington, DC, USA, 2012. IEEE Computer Society.

[125] F. Nadalin, F. Vezzi, S. Scalabrin, and A. Policriti. Identification, reconstruction, and validation of insertions in resequencing projects with GapFiller (poster). In *Proceedings on the 9th annual meeting of the Bioinformatics Italian Society (BITS 2012)*, Catania, May 2012.

[126] N. Nagarajan and M. Pop. Parametric complexity of sequence assembly: Theory and applications to next generation sequencing. *Journal of Computational Biology*, 16:897–908, 2009.

[127] G. Narzisi and B. Mishra. SUTTA: Scoring-and-Unfolding Trimmed Tree Assembler. 2008.

[128] G. Narzisi and B. Mishra. Comparing De Novo Genome Assembly: The Long and Short of It. *PLoS ONE*, 6(4):e19175, Apr. 2011.

[129] T. W. Nilsen and B. R. Graveley. Expansion of the eukaryotic proteome by alternative splicing. *Nature*, 463(7280):457–463, 2010.

[130] B. Nystedt, N. R. Street, A. Wetterbom, A. Zuccolo, Y.-C. Lin, D. G. Scofield, F. Vezzi, N. Delhomme, S. Giacomello, A. Alexeyenko, et al. The norway spruce genome sequence and conifer genome evolution. *Nature*, 2013.

[131] F. Ozsolak, A. R. Platt, D. R. Jones, J. G. Reifenberger, L. E. Sass, P. McInerney, J. F. Thompson, J. Bowers, M. Jarosz, and P. M. Milos. Direct rna sequencing. *Nature*, 461(7265):814–818, 2009.

[132] G. Pertea, X. Huang, F. Liang, V. Antonescu, R. Sultana, S. Karamycheva, Y. Lee, J. White, C. Foo, B. Parvizi, J. Tsai, and J. Quackenbush. TIGR gene indices clustering tools (TGICL): a software system for fast clustering of large EST datasets. *Bioinformatics*, 19(5):651–652, 2003.

[133] G. Pesole. What is a gene? an updated operational definition. *Gene*, 417:1–4, 2008.

[134] P. Pevzner, H. Tang, and W. M. S. An eulerian path approach to DNA fragment assembly. *Proc. Natl. Acad. Sci. U. S. A.*, 98:9748–9753, 2001.

[135] N. Philippe, M. Salson, T. Commes, E. Rivals, et al. CRAC: an integrated approach to the analysis of RNA-seq reads. *Genome Biology*, 14:R30, 2013.

[136] A. M. Phillippy, M. C. Schatz, and M. Pop. Genome assembly forensics: finding the elusive mis-assembly. *Genome Biology*, 9:R55, 2008.

[137] A. Policriti, A. I. Tomescu, and F. Vezzi. A randomized numerical aligner (rna). *Language and Automata Theory and Applications*, 603:512–523, 2010.

[138] M. Pop. Genome assembly reborn: recent computational challenges. *Briefings in Bioinformatics*, 10(4):354–366, 2009.

[139] M. A. Quail, M. Smith, P. Coupland, T. D. Otto, S. R. Harris, T. R. Connor, A. Bertoni, H. P. Swerdlow, and Y. Gu. A tale of three next generation sequencing platforms: comparison of ion torrent, pacific biosciences and illumina miseq sequencers. *BMC genomics*, 13(1):341, 2012.

[140] D. A. Rasko, D. R. Webster, J. W. Sahl, A. Bashir, N. Boisen, F. Scheutz, E. E. Paxinos, R. Sebra, C. S. Chin, D. Iliopoulos, A. Klammer, P. Peluso, L. Lee, A. O. Kislyuk, J. Bullard, A. Kasarskis, S. Wang, J. Eid, D. Rank, J. C. Redman, S. R. Steyert, J. Frimodt-Møller, C. Struve, A. M. Petersen, K. A. Krogfelt, J. P. Nataro, E. E. Schadt, and M. K. Waldor. Origins of the E. coli strain causing an outbreak of hemolytic–uremic syndrome in Germany. *N. Engl. J. Med.*, 365:709–717, 2011.

[141] M. D. Robinson, D. J. McCarthy, and G. K. Smyth. edger: a bioconductor package for differential expression analysis of digital gene expression data. *Bioinformatics*, 26(1):139–140, 2010.

[142] S. Rodrigue, A. C. Materna, S. C. Timberlake, and *et al.* Unlocking short read sequencing for metagenomics. *PLoS ONE*, 5(7):e11840, July 2010.

[143] S. L. Salzberg, A. M. Phillippy, A. V. Zimin, D. Puiu, T. Magoc, S. Koren, T. Treangen, M. C. Schatz, A. L. Delcher, M. Roberts, G. Marcais, M. Pop, and J. A. Yorke. GAGE: A critical evaluation of genome assemblies and assembly algorithms. *Genome Research*, 22(3):557–67, Mar. 2012.

[144] F. Sanger, S. Nicklen, and A. Coulson. Dna sequencing with chain-terminating inhibitors. *Proc. Natl. Acad. Sci. USA*, 74:5463–5467, 1977.

[145] S. Scalabrin, A. Policriti, F. Nadalin, C. Del Fabbro, M. Miculan, S. Pinosio, F. Cattonaro, E. Vendramin, V. Aramini, I. Verde, L. Rossini, R. Testolin, and M. Morgante. A catalog of molecular diversity within prunus germplasm inferred from next-generation sequencing data: bioinformatic approaches and challenges (poster). In *AGI-SIBV-SIGA joint meeting*, 2011.

[146] S. Scalabrin, A. Policriti, F. Nadalin, S. Pinosio, F. Cattonaro, E. Vendramin, V. Aramini, I. Verde, D. Bassi, R. Pirona, L. Rossini, G. Cipriani, R. Testolin, and M. Morgante. A catalog of molecular diversity of prunus germplasm gathered from aligning NGS reads to the peach reference sequence: bioinformatic approaches and challenges. In *XIII Eucarpia Symposium on Fruit Breeding and Genetics*, 2013.

[147] M. Schena, D. Shalon, R. W. Davis, and P. O. Brown. Quantitative monitoring of gene expression patterns with a complementary DNA microarray. *Science*, 270(5235):467–70, 1995.

[148] S. Schliesky, U. Gowik, A. P. Weber, and A. Bräutigam. RNA-seq assembly — are we there yet? *Frontiers in Plant Science*, 3:220, 2012.

[149] B. Schmidt, R. Sinha, B. Beresford-Smith, and S. J. Puglisi. A fast hybrid short read fragment assembly algorithm. *Bioinformatics*, 25:2279–2280, 2009.

[150] M. H. Schulz, D. R. Zerbino, M. Vingron, and E. Birney. Oases: robust *de novo* RNA-seq assembly across the dynamic range of expression levels. *Bioinformatics*, 28(8):1086–1092, 2012.

[151] J. Shendure and H. Ji. Next-generation DNA sequencing. *Nature biotechnology*, 26(10):1135–1145, 2008.

[152] J. T. Simpson and R. Durbin. Efficient construction of an assembly string graph using the FM-index. *Bioinformatics*, 26(12):i367–i373, 2010.

[153] J. T. Simpson and R. Durbin. Efficient de novo assembly of large genomes using compressed data structures. *Genome Res.*, 22(3):549–56, 2012.

[154] J. T. Simpson, K. Wong, S. D. Jackman, J. E. Schein, S. J. Jones, and B. I. Abyss: A parallel assembler for short read sequence data. *Genome Res*, 19:1117–1123, 2009.

[155] N. Siva. 1000 genomes project. *Nature biotechnology*, 26(3):256–256, 2008.

[156] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–7, Mar. 1981.

[157] D. D. Sommer, A. L. Delcher, S. L. Salzberg, and M. Pop. Minimus: a fast, lightweight genome assembler. *BMC Bioinformatics*, 8:64, 2007.

[158] T. Steijger, J. F. Abril, P. G. Engström, F. Kokocinski, T. J. Hubbard, R. Guigó, J. Harrow, P. Bertone, R. Consortium, et al. Assessment of transcript reconstruction methods for rna-seq. *Nature methods*, 2013.

[159] Y. Surget-Groba and J. I. Montoya-Burgos. Optimization of *de novo* transcriptome assembly from next-generation sequencing data. *Genome research*, 20:1432–1440, 2010.

[160] G. G. Sutton, O. 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.

[161] R. J. Taft, M. Pheasant, and J. S. Mattick. The relationship between non-protein-coding dna and eukaryotic complexity. *Bioessays*, 29(3):288–299, 2007.

[162] A. I. Tomescu, A. Kuosmanen, R. Rizzi, and V. Mäkinen. A novel min-cost flow method for estimating transcript expression with rna-seq. *BMC bioinformatics*, 14(Suppl 5):S15, 2013.

[163] C. Trapnell, D. G. Hendrickson, M. Sauvageau, L. Goff, J. L. Rinn, and L. Pachter. Differential analysis of gene regulation at transcript resolution with RNA-seq. *Nat Biotechnol.*, 31(1):46–53, 2013.

[164] C. Trapnell, L. Pachter, and S. L. Salzberg. Tophat: discovering splice junctions with RNA-Seq. *Bioinformatics*, 25(9):1105–11, May 2009.

[165] C. Trapnell, B. A. Williams, G. Pertea, A. Mortazavi, G. Kwan, M. J. van Baren, S. L. Salzberg, B. J. Wold, and L. Pachter. Transcript assembly and quantification by RNA-Seq reveals unannotated transcripts and isoform switching during cell differentiation. *Nature Biotechnology*, 28(5):511–515, 2010.

[166] V. E. Velculescu, L. Zhang, B. Vogelstein, and K. W. Kinzler. Serial analysis of gene expression. *Science*, 270(5235):484–7, 1995.

[167] J. C. Venter, M. D. Adams, E. W. Myers, P. W. Li, R. J. Mural, G. G. Sutton, H. O. Smith, M. Yandell, C. A. Evans, R. A. Holt, et al. The sequence of the human genome. *science*, 291(5507):1304–1351, 2001.

[168] F. Vezzi. *Next Generation Sequencing Revolution Challenges: Search, Assemble and Validate Genomes*. PhD thesis, University of Udine, 2011.

[169] F. Vezzi, C. Del Fabbro, A. I. Tomescu, and A. Policriti. rNA: a fast and accurate short reads numerical aligner. *Bioinformatics (Oxford, England)*, 28(1):123–124, 2012.

[170] F. Vezzi, G. Narzisi, and B. Mishra. Feature-by-Feature — Evaluating *De Novo* Sequence Assembly. *Plos One*, 7(2):e31002, 2012.

[171] F. Vezzi, G. Narzisi, and B. Mishra. Reevaluating assembly evaluations with feature response curves: GAGE and Assemblathons. *PLoS ONE*, 7(12):e52210, 2012.

[172] K. Wang, D. Singh, Z. Zeng, S. J. Coleman, Y. Huang, G. L. Savich, X. He, P. Mieczkowski, S. A. Grimm, C. M. Perou, et al. MapSplice: accurate mapping of RNA-seq reads for splice junction discovery. *Nucleic acids research*, 38(18):e178–e178, 2010.

[173] Z. Wang, M. Gerstein, and M. Snyder. RNA-Seq: a revolutionary tool for transcriptomics. *Nature Reviews Genetics*, 10(1):57–63, 2008.

[174] R. L. Warren, G. G. Sutton, S. J. Jones, and R. A. Holt. Assembling millions of short DNA sequences using SSAKE. *Bioinformatics*, 23:500–501, 2007.

[175] J. D. Watson. *The double helix; a personal account of the discovery of the structure of DNA*. Atheneum, 1968.

[176] J. Wu, O. Anczuków, A. R. Krainer, M. Q. Zhang, and C. Zhang. OLego: fast and sensitive mapping of spliced mRNA-Seq reads using small seeds. *Nucleic acids research*, 41(10):5149–5163, 2013.

[177] T. D. Wu and S. Nacu. Fast and snp-tolerant detection of complex variants and splicing in short reads. *Bioinformatics*, 26(7):873–881, 2010.

[178] Y. Xing, A. Resch, and C. Lee. The multiassembly problem: reconstructing multiple transcript isoforms from est fragment mixtures. *Genome Research*, 14(3):426–441, 2004.

[179] D. R. Zerbino and E. Birney. Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome Res*, 18:821–829, 2008.

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

[181] Y. Zhang, E.-W. Lameijer, P. A. 't Hoen, Z. Ning, P. E. Slagboom, and K. Ye. PASSion: A pattern growth algorithm based pipeline for splice junction detection in paired-end RNA-Seq data. *Bioinformatics*, 28:479–486, 2012.

[182] Q.-Y. Zhao, Y. Wang, Y.-M. Kong, D. Luo, X. Li, and P. Hao. Optimizing *de novo* transcriptome assembly from short-read RNA-Seq data: a comparative study. *BMC Bioinformatics*, 12(Suppl 14):S2, 2011.

[183] A. V. Zimin, G. Marçais, D. Puiu, M. Roberts, S. L. Salzberg, and J. A. Yorke. The masurca genome assembler. *Bioinformatics*, 29(21):2669–2677, 2013.