UNIVERSITÀ DEGLI STUDI DI PADOVA
DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE
TESI DI LAUREA

TOWARDS LOCALITY AWARE DE NOVO DNA ASSEMBLY OF SHORT READS IN COLOUR SPACE

RELATORE: Enoch Peserico Stecchini Negri De Salvi

CORRELATORE: Federica Bogo

LAUREANDO: *Vincenzo Maria Cappelleri*

Padova, 13 Dicembre 2011
A.A. 2011/12

Abstract

This thesis presents a new approach towards de novo DNA assembly for short reads. Its two main contributions are a novel, robust filtering scheme for noisy reads that outperforms the (accuracy of the) widely used Sasson's filter, and a novel assembly algorithm that, minimizing space and maximizing locality of accesses, runs faster than all state-of-the-art algorithms even when on substantially cheaper hardware.

The thesis is organized as follows. Chapter 1 provides a brief introduction to sequencing technologies and existing assembly algorithms. Chapter 2 presents our filtering scheme, and experimentally evaluates its accuracy (comparing it to that of the Sasson's filter); this is joint work with G. Bilardi, F. Peruch and M. Schimd. Chapter 3 introduces our assembly algorithm, and provides a theoretical analysis of its correctness and asymptotic performance; this is joint work with E. Peserico. Chapter 4 experimentally evaluates the performance of our assembly algorithm combined; this is joint work with F. Bogo. Finally, Chapter 5 summarizes our results and examines directions of future research before concluding with the Bibliography.

Keywords: DNA, SOLiD, colour, filter, Sasson, assembly, de novo, Velvet, Abyss, locality, hash, memory.

Acknowledgement: This work was supported in part by FBK, INFN and PAT under project Aurora Science.

Contents

Chapter 1

DNA sequencing and assembly: an overview

This Chapter provides an overview of current state-of-the-art techniques in DNA sequencing and assembly. After introducing some preliminary notions (Section 1.1), we review the most widely employed DNA sequencing techniques (Section 1.2) and assembly algorithms (Section 1.3).

1.1 Preliminaries

This Section introduces some terminology and defines some essential concepts. We assume the reader has basic familiarity with the field of molecular biology, and provide only a few specialized definitions. This preliminary and general "glossary" will be integrated by more specific notions during discussion, when needed. First, we introduce the technical notions of DNA *primers*, DNA *templates* and *base-pairs*. Then, we clearly distinguish between DNA *sequencing* and *assembly* procedures, highlighting the differences and the relationships between the two. While discussing sequencing and assembly, we provide the auxiliary definitions of *read*, *contig* and $k-mer$.

A *DNA primer* is a short strand of chemically synthetized nucleotides, that serves as a starting point for DNA synthesis. Many of the laboratory techniques of biochemistry and molecular biology that involve DNA polymerase – such as DNA sequencing – require DNA primers. A *DNA template* is a recombinant DNA molecule made up of two regions: a binding zone (usually an adaptor sequence to which a primer can bind) and a target sequence (typically an unknown portion

to be sequenced). Genomes usually present a double-stranded DNA. A *base pair* (bp) corresponds to a linked pair of two nucleotides on opposite complementary strands; in particular, adenine forms a base pair with thymine and guanine forms a base pair with cytosine. The size of an individual gene or of an organism's entire genome is often measured in base pairs.

The goal of DNA sequencing is mapping the sequence of the nucleotides in a molecule of DNA, typically using biochemical methods. Starting from the 1970s, several sequencing technologies have been proposed (see [41] for a survey). In general, they can not read whole genomes in one run, but rather read small pieces of between $20 - 1000$ bases (b), depending on the technology used. These fragments are commonly called *reads*. DNA assembly *aligns* and *merges* the reads into longer composite sequences (known as *contigs*), in order to reconstruct the original (much longer) DNA sequence [12]. For efficiency, most assemblers generally introduce the notion of $k-$*mers*. A $k-$mer is a sequence of k bases (where k is a positive integer); usually, only consecutive bases are used. $k-$mers allow an efficient discovery of *overlapping* reads (that is, matching reads which should be aligned or merged into the same contig): reads with high sequence similarity share $k-$mers in their overlapping regions, and shared $k-$mers are generally easier to find than overlaps. Fast detection of shared $k-$mers dramatically reduces the computational cost of assembly, especially compared to all-against-all pair-wise read alignment. As we shall see in Subsection 1.2.4, k turns out to be a crucial parameter: it affects both efficiency and accuracy of the assembly process.

We distinguish two types of assembly [29]: *mapping* assembly and *de novo* assembly. Mapping assembly aligns and merges reads (or $k-$mers) against an existing backbone sequence, building a sequence that is similar but not necessarily identical to the backbone sequence itself. Mapping is useful when reads need comparing against a reference sequence – e.g., when comparing (reads from) an individual genome with a related genome. Mapped reads can reveal small-scale population differences such as substitutions and indels [18]. De novo assembly aligns and merges reads (or $k-$mers) to create full-length (sometimes novel) sequences. It refers to reconstruction in its pure form, without consultation of previously resolved sequences. In terms of complexity and time requirements, de novo assemblies are orders of magnitude slower and more memory intensive than mapping assemblies.

1.2 Sequencing technologies

In this Section, we describe the most widely employed sequencing techniques developed during the last 40 years. First, we provide an overview of these techniques – adhering to the classical distinction between *first generation* sequencing (Subsection 1.2.1) and *next generation* sequencing (Subsection 1.2.2). We emphasize how this distinction depends not only on temporal considerations (the former were developed a quarter of century ago, the latter became commercially available only in the last decade), but also on performance aspects (such as read length, bases per second, and costs). Then, we distinguish between sequencing data provided in *base space* and in *colour space* (Subsection 1.2.3) Finally, we discuss the opportunities and challenges these techniques present to de novo assembly (Subsection 1.2.4).

1.2.1 First generation sequencing

"First generation" sequencing methods are essentially two: the *Maxam–Gilbert* (or *chemical degradation*) method [26] and the *Sanger* (or *chain termination*) method [37,38]. They have been developed in the late 1970s and share a common approach. Both methods generate, in different ways, a nested set of single–stranded DNA fragments; these fragments are then separated according to their size by an electrophoresis procedure on high–resolution polyacryalmide gel.

[26] introduces the first widely employed DNA sequencing technique – the Maxam–Gilbert method – in 1976. In a nutshell, this procedure determines the nucleotide sequence of a terminally labeled DNA molecule by breaking it at each repetition of a base (adenine, guanine, cytosine or thymine) with chemical agents. It employs four different chemical cleavages, each specific to a base. The lengths of the labeled fragments then identify the positions of that base. This method presents three major drawbacks: its technical complexity (making it unsuitable for standard molecular biology kits), extensive use of hazardous chemicals and, mostly, difficulties with scaling (it permits the extraction of no more than 100 bases from the point of labeling) [27].

The Sanger method [37, 38] is more efficient and generates raw data that is easier to interpret. The Sanger method dominated the industry for almost two decades and led to a number of monumental accomplishments, including the

completion of the only *finished–grade* [1] human genome sequence.

Since its first, complete formulation in 1977 [38], the Sanger method has remained conceptually unchanged. It utilizes a DNA polymerase [2] to synthesize a complementary copy of a single–stranded DNA template in the presence of chain–terminating (inhibitor) nucleotides – the didexynucleotides (ddNTPs). ddNTPs, when incorporated, determine termination of the chain growth: in this way, the DNA synthesis reaction is randomly terminated whenever a ddNTP is added to the growing nucleotide chain. This produces truncated fragments of varying lengths, with an appropriate ddNTP at their terminus. The fragments are separated by size using polyacrylamide gel electrophoresis and the terminal ddNTPs are used to reveal the DNA sequence of the template strand.

Since the early 1980s, significant technical improvements have made the Sanger method both more precise and more efficient [27]. The use of fluorescently labelled DNA molecules allowed automatic detection of ddNTPs by laser-based technology; breakthroughs in polymer biochemistry increased sequencing efficiency. These advances in sequencing technology contributed to the relatively low error rate, *long read length*, and robust characteristics of modern Sanger sequencers. Nowadays, a commonly used automated high-throughput Sanger sequencing instrument from Applied Biosystems [3], the ABI 3730xl [4], can produce up to 96 kb in a 3-hour run. Despite the many advances in the chemistry-related technologies and the robust performance of instruments like the 3730xl, the Sanger sequencing is relatively expensive; its application to large sequencing projects has remained beyond the means of the typical grant-funded investigator. Next generation sequencing technologies address, to different degrees, this limitation.

1.2.2 Next generation sequencing

In the last few years, the Sanger method has been partially supplanted by the so called "next generation" sequencing (NGS) technologies. These newer technolo-

[1] a *finished–grade* sequence refers to a sequence exhibiting high base coverage as well as few errors and gaps.

[2] a *polymerase* is an enzime enabling the polymerization of new DNA from an existing DNA template.

[3] `www.appliedbiosystems.com`

[4] `www.appliedbiosystems.com/products/abi3730xlspecs.cfm`

gies shift away from the classic Sanger method for genome analysis. They combine various ad-hoc template preparation and sequencing techniques with intesive data analysis (in particular, with elaborate alignment and assembly methods). The major advantage of NG sequencing is a dramatic increase in cost-effective sequence throughput: in some cases, sequencers can produce in excess of one billion reads per instrument run [28]. This enormous – and cheap – volume of data comes at the expense of a shorter reads and often lower accuracy. Here, we discuss these advantages and disadvantages briefly reviewing the most popular, commercially available technologies: Roche/454, Illumina/Solexa, Polonator, Helicos BioSciences HeliScope and ABI SOLiD.

Roche/454 An inherent limitation of Sanger sequencing is the requirement of cloning into bacterial hosts the DNA fragments to be sequenced. This cloning step is prone to errors, lengthy and quite labour intensive. The 454 technology [25] – the first next–generation sequencing technology released to the market in 2005 – circumvents the cloning requirement by taking advantage of a highly efficient in vitro DNA amplification method, the *Emulsion Polymerase Chain Reaction*.

Emulsion PCR amplifies DNA inside water droplets in an oil solution. Each droplet contains a single DNA template attached to a *primer* bead; droplets act as individual amplification reactors, producing clonal copies of a unique DNA template per bead. Each template-containing bead is subsequently transferred into a well of a picotiter plate and the clonally related templates are analysed using a pyrosequencing reaction [30]. The use of the picotiter plate allows hundreds of thousands of pyrosequencing reactions to be carried out in parallel, massively increasing the sequencing throughput. The pyrosequencing approach is a *sequencing-by-synthesis* technique measuring the release of inorganic pyrophosphate (PPi) by chemiluminescence. The template DNA is immobilized, and solutions of ddNTPs are added one at a time; the release of PPi, whenever the complementary nucleotide is incorporated, is detectable by the light produced by a chemiluminescent enzyme present in the reaction mix. The sequence of DNA template is determined from a "pyrogram", which shows the order of correct nucleotides that have been incorporated.

This technology provides intermediate read length and price per base compared to Sanger sequencing on one end, and to later NGS techniques on the other.

The current 454 platform [5] marketed by Roche Applied Science [6] is capable of generating $80 - 120$ Mb of genome sequence in $200 - 300$ bp reads in a 4-hour run.

Illumina/Solexa The Illumina/Solexa approach [3] achieves cloning-free DNA amplification in two steps. First, it attaches single-stranded DNA fragments to primers through chemical adapters; then, it amplifies these DNA templates so that local clonal colonies are formed ("solid-phase bridge" amplification). After the amplification step, approximately 1000 clonal copies of a single template molecule are obtained. These copies are sequenced in a massively parallel fashion using a DNA sequencing-by-synthesis approach. Four types of ddNTPs – labeled with fluors of four different colours – are added to clonal copies. Such ddNTPs allow the distinction among the different bases at any given sequence position.

Compared to pyrosequencing, the Illumina approach produces shorter sequence reads; hence, it cannot resolve short sequence repeats. In addition, due to the use of modified DNA polymerases and reversible terminators, it may introduce a significant number of substitution errors [28]. Typically, the 1G genome analyzer [7] from Illumina, Inc. [8] is capable of generating 35 bp reads and producing at least 1 Gb per run (in $2 - 3$ days).

Polonator The Polonator G.007 instrument [9] relies on polony sequencing [42]. Polony sequencing technology is open and provides freely available software and protocols; it was ultimately incorporated into the Applied Biosystems SOLiD platform [10]. In a nutshell, polony sequencing relies on an accurate multiplex sequencing technique that can be used to "read" millions of immobilized DNA sequences in parallel. As in Roche/454, it employs emulsion PCR in order to obtain clonal colonies of an initial DNA template – such clones are then sequenced in parallel through enzymatic *ligation* reactions. We explain the *sequencing-by-ligation* scheme in greater detail when discussing the SOLiD technique, below.

[5] www.my454.com

[6] www.roche-applied-science.com

[7] www.illumina.com/systems/genome_analyzer_iix.ilmn

[8] www.illumina.com

[9] www.polonator.org/instrument/

[10] www.appliedbiosystems.com/absite/us/en/home/applications-technologies/solid-next-generation-sequencing.html

Helicos Biosciences HeliScope Helicos BioSciences [11] was the first group to commercialize a single-molecule sequencer, the HeliScope [46]. It requires no cloning or amplification in template preparation and, unlike the other NGS techniques, utilizes a one-colour termination method. The Helicos terminators are labeled with the same dye and dispensed individually in a predetermined order. Fluorescence imaging techniques allow the production of one-colour images, highlighting the sequencing data. While the lack of cloning mechanisms ensures non-bias representations of DNA templates, the adopted sequencing method introduces high error rates compared with other terminator chemistries.

ABI SOLiD SOLiD (Supported Oligonucleotide Ligation and Detection system) [32] from Applied Biosystems incorporates polony sequencing in order to achieve massively parallel sequencing by ligation.

Construction of sequencing templates for analysis on the SOLiD instrument begins with an emulsion PCR amplification step, similar to that used in the 454 technique. The amplification products are transferred onto a glass surface and a four-colour sequencing-by-ligation scheme is then applied to them – the same scheme adopted in polony sequencing. Through enzymatic ligation reactions, primers obtained during amplification are anchored to polymers labelled with four different fluorescent dyes. To the extent that the ligase discriminates for complementarity between polymers and bases, the fluorescent signal allows to infer the identity of the corresponding base. Using the four-dye encoding scheme, each position is probed twice; the identity of each nucleotide is determined by analysing the colour resulting from two successive ligation reactions. The two-base encoding scheme introduces robustness. In particular, it enables the distinction between a sequencing error and a sequence polymorphism: an error would be detected in only one particular ligation reaction, whereas a polymorphism would be detected in both. Newly released SOLiD instruments are capable of producing $1 - 3$ Gb of data in 35 bp reads over an 8-day run.

1.2.3 Base space and colour space

NG sequencers provide output data encoding it in *base space* or in *colour space*.

[11] `www.helicosbio.com`

The base space representation is the most common one. According to this encoding scheme, reads are sequences of characters; each character represents one nucleotide (i.e., A, C, G or T). It is common to refer to the individual characters in a read as *base calls* (to highlight the fact that, due to the presence of errors, a base call not correspond to an actual base).

SOLiD platforms introduced a novel, more effective way of representing sequenced data. Data are collected in *colour space*, a special type of two-base encoding. To reconstruct a DNA sequence, a further conversion is then required. Reads are sequences of *colours*; each colour is encoded by a digit. There are four possible digits: 0, 1, 2 and 3. Each digit corresponds to a *colour call*. Each colour is a *di*-base; it encodes one base in relation to its preceding base.

In particular, each colour can be conceptually interpreted as a representation of the *interstice* between bases. Table 1.1 shows the translation from *di-bases* to colours.

	A	**C**	**G**	**T**
A	0	1	2	3
C	1	0	3	2
G	2	3	0	1
T	3	2	1	0

Table 1.1: Di-base to colour translation table.

This two-base encoding requires dual interrogation of each base during SOLiD's sequencing process, helping to distinguish sequencing errors from true polymorphisms. In general, it is possible to gain robustness to error by switching from colour space to base space and viceversa; many assemblers for the SOLiD platform form alignments in colour space and then test their validity by conversion to base space.

1.2.4 NGS and de novo assembly: main challenges

NGS platforms can provide an enormous volume of data at a very low cost when compared against common Sanger sequencers. Unfortunately, this cost-effective throughput comes with two main drawbacks: much shorter reads and harder

retrieval of *mate pairs* [28]. Short reads probably constitute the more serious problem.

Table 1.2.4 directly compares the NGS technologies discussed in the previous Subsection, showing for each read length, data per run (in *Gb*), and run time.

Platform	**Read length (b)**	**Gb per run**	**Run time (days)**
Roche/454	330	0.45	0.35
Illumina/Solexa	75	18	4
Polonator G.007	26	12	5
BioSciences HeliScope	32	37	8
ABI SOLiD	75	30	7

Table 1.2: Read length, Gb per run and run time of the most popular NG sequencers.

In general, while Sanger sequencing allows reads of up to 1000 bp, today's NGS reads rarely exceed lengths of 400 bp (Roche/454), and in many cases only allow reads of 100 bp (Illumina/Solexa and SOLiD) or less. Short reads poses novel challenges to de novo assembly. Shorter reads deliver less information per read, increasing the computational complexity of assembling reads into a longer DNA sequence. In particular, this shortcoming is an issue in sequencing new genomes and in sequencing highly modified segments – e.g, segments discovered in cancer genomes or in regions of structural variation.

A partial solution to short reads is *oversampling* the target genome with short reads from random positions. By providing higher coverage, NGS attempts to guarantee sufficient (detectable) overlap between reads. Unfortunately, this approach has some limitations. First, NGS can not always ensure a *uniform* coverage of the target sequence. Coverage variation is introduced by chance, by variation in the number of clones of source DNA molecules, and by compositional bias introduced by sequencing technologies during chemical treatments. While very low coverage introduces gaps in assembled sequences, coverage variability invalidates coverage-based statistical tests, and diminishes the effectiveness of coverage-based diagnostics. Furthermore, as a side effect, oversampling may both undermine assembly accuracy and intensify computational issues related to large data sets.

The huge number of reads and their short length make read accuracy critical for an effective assembly. For Sanger-sequenced contigs, the PHRED method [10, 11] is well established; for next generation technologies, no similar, well-defined approaches exist. In general, sequence quality and base accuracy remain highly uncertain.

Sanger platforms can deliver *paired-end* reads (or *mate pairs*) – that is, pairs of reads with an *estimated* constraint on their relative *orientation* and *separation* in the target sequence. Mate pairs' separation and orientation estimate is usually provided to assembly software through so-called libraries of reads. Early NG sequencers offered only *unpaired* reads; recently, however, platforms started supporting paired-end protocols. Thus, while early NG assembly software targeted only unpaired reads, more recent software exploits mate pairs in order to improve assembly effectiveness.

In general, this information can be of great help in reconstructing the distribution of fragment sizes. Also, a sufficient variety of mate pairs' separations can allow one to resolve low coverage read sets into single contigs, even if those contigs will sport "holes" whose contents are unknown and whose size is known only approximately. Finally, mate pairs can be of great help when dealing with repeated sequences [34] (see below).

Segments sharing perfect repeats can be indistinguishable, especially when repeats are longer than the reads. Careful repeat separation involves correlating reads by patterns in the different base calls they may have. This approach may be aided somewhat by high coverage but is seriously hampered by high error rates in extracted fragments. Repeat resolution depends on the presence of "spanner" reads – that is, single reads spanning a repeat instance with a long enough unique base sequence on either side of the repeat.

In some cases, repeats longer than the reads can be resolved by spanning mate pairs. Complete resolution usually requires two resources: pairs straddling the repeat with each end in unique sequence and pairs with exactly one end in the repeat. Sequencing errors may severely undermine the effectiveness of this approach. Assemblers must tolerate – to some extent – imperfect sequence alignments, in order to avoid missing true joins. However, excessively low thresholds in error tolerance lead to false positive joins. False-positive joins can induce *chimeric* assemblies (i.e., multiple valid solutions) and invalidate the correctness of whole

contigs. The assessment of an adequate sequencing error tolerance threshold is a problem especially when dealing with reads from inexact (polymorphic) repeats. Probably, only a decrease in the error-rate provided by NGS platforms could definitively increase the accuracy at which assembly software may operate.

1.3 De novo assembly algorithms

We divide de novo assembly algorithms into two broad categories, both based on graphs: *Overlap/Layout/Consensus* (OLC) methods and *de Bruijn Graph* (DBG) methods. All these methods rely on overlapping $k-$mers belonging to different reads, and the choice of k is crucial. The probability that a true overlap spans shared $k-$mers depends on the value of k, the length of the overlap, and the rate of error in the reads. An appropriate value of k should be large enough that most false overlaps do not share $k-$mers by chance, and small enough that most true overlaps do share $k-$mers. Further, the choice should be robust, taking into account variations in both read coverage and accuracy.

1.3.1 Overlap/Layout/Consensus assemblers

The OLC approach is typical of Sanger-data assemblers, such as Celera Assembler [31], Arachne [2] and PCAP [16]. We include it for completeness.

OLC assemblers try to organize the retrieved reads into the original DNA sequence by creating an *overlap* graph. An overlap graph represents the sequencing reads and their overlaps. Conceptually, the graph has nodes to represent the reads and edges to represent the overlaps. Paths through the graph are the potential contigs, and paths can be converted into a sequence. There are two ways to force paths to obey the semantics of double-stranded DNA. If the graph has separate nodes for read ends, then paths must exit the opposite end of the read they enter. If the graph has separate edges for the forward and reverse strands, then paths must exit a node on the same strand they enter.

Roughly speaking, OLC assembly is made up of three phases:

1. all-against-all, pair-wise read comparisons are performed. The assembly software pre-computes $k-$mer content across *all* reads, selects overlap candidates (i.e., reads sharing at least one $k-$mer), and finally estimates alignments using the $k-$mers as alignment seeds. The computational cost of

this operation is usually mitigated through heuristic algorithms (such as the seed and extend heuristic [29]).

Three parameters severely affect the effectiveness of overlap discovery: the value of k, the minimum overlap length and the minimum percent identity required for an overlap. Larger parameter values lead to higher accuracy but may generate shorter contigs. In particular, these parameters affect robustness in the face of base calling error and low-coverage sequencing.

In order to improve efficiency, overlap discovery can run in parallel by use of a matrix partition;

2. construction and manipulation of the resulting overlap graph leads to an approximate sequence layout;

3. multiple sequence alignment (MSA) determines the precise layout and then estimates the final sequence. There is no known method for efficiently reaching the optimal MSA consensus. Therefore, this phase uses progressive pair-wise alignments and runs in parallel, partitioned by contig. This phase is expensive in terms of both computational complexity and memory usage: all the base calls must be loaded into memory.

The OLC approach carries three main limitations, which make it not well suitable for NG sequencers. First, it can effectively incorporate a large amount of overlaps (connections) of various length only in presence of long-read data. Second, its computational complexity is very high, due to the all-against-all pair-wise read comparisons. Finally, when dealing with high-coverage data, it also requires large amounts of memory – since during the third phase it must keep in memory all the replicated sequences and thus an amount of data proportional to the input size rather than just to the genome size.

1.3.2 De Bruijn Graph assemblers

This Subsection briefly reviews the assembly approaches most widely used in conjunction with NG sequencing: the de Bruijn graph (DBG) assemblers. These approaches all rely on de Bruijn graphs and on a special kind of DBG specialization – the $k-mer$ *graphs*.

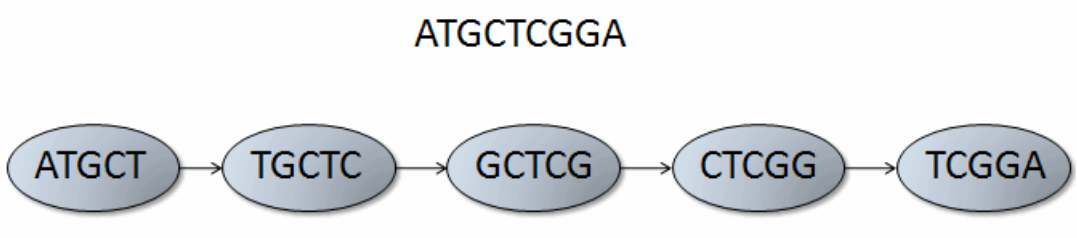

Figure 1.1: Representation of the read *atgctcgga* by a $k-$mer graph ($k = 5$): nodes correspond to the 5 distinct $k-$mers starting at each base, while edges connect any pair of $k-$mers overlapping by $k-1$ bases in the read.

De Bruijn graphs were originally used to represent strings from a finite alphabet. Nodes represent all possible fixed-length strings; edges represent suffix-to-prefix perfect overlaps.

A $k-$mer graph is a form of de Bruijn graph. Its nodes represent all the fixed-length subsequences drawn from a larger sequence; its edges represent all the fixed-length overlaps between subsequences that were consecutive in the larger sequence. In early formulations [35], one edge is drawn for each $k-$mer starting at each base (excluding the last $k-1$ bases); in this way, the nodes represent overlaps of $k-1$ bases. In other formulations [47], there is one node representing the $k-$mer starting at each base. An edge links any pair of nodes if the two $k-$mers the nodes represent are subsequent in at least one read. In other words, the edges represent overlaps of $k-1$ bases. The two representations are fundamentally equivalent.

By construction, the graph contains a path corresponding to the original sequence (see Figure 1.1): sequence assembly is performed by finding a path traversing all edges (i.e., an Eulerian path). $k-$mer graphs directly represent the input reads. Each read induces a path; reads with perfect overlaps induce a common path. Thus, perfect overlaps are detected implicitly without any (expensive) all-against-all pair-wise comparisons.

Redundancy and erroneous base calls in NG sequencers data may induce anomalous features in the resulting $k-$mer graphs. In particular, there are four kinds of anomalous features: *spurs*, *bubbles*, *converging-diverging paths* and *cycles* (see Figure 1.2). Spurs are short, dead-end divergences from the main path; they are induced by sequencing errors toward one end of a read. Bubbles are paths that first diverge, then converge; they are induced by sequencing errors toward

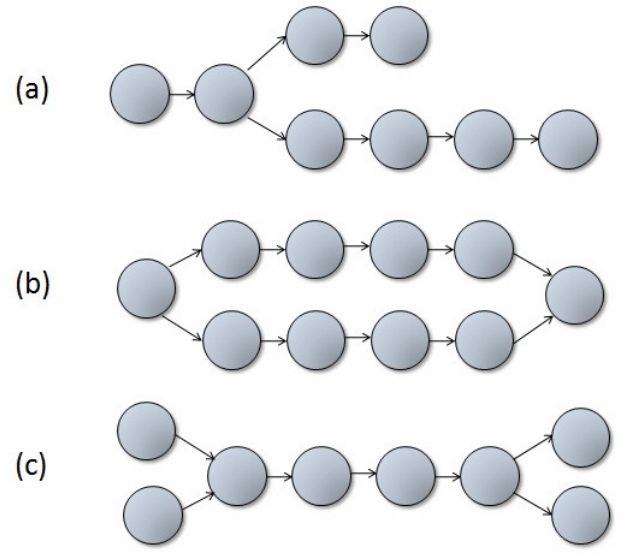

Figure 1.2: Complexity induced in $k-$mer grahps by redundancy and erroneous base calls: spurs (a), bubbles (b) and converging-diverging paths (c).

the middle of a read. Converging-diverging paths are paths first converging into a common shared subsequence and then diverging; they are induced by repeats in the sequenced data. Cycles are induced by repeats too.

Most DGB assemblers share a core set of common features:

- graph construction to represent reads and merge them into a longer sequence;
- error detection and correction based on sequence composition of the reads, during ad-hoc pre-processing or post-processing phases;
- removal of error-induced paths such as spurs or bubbles;
- collapse of polymorphism-induced complexity given by bubbles;
- reduction of converging-diverging paths to consensus sequences;
- in some cases, use of extra information outside the graph: mate pairs may simplify path reduction and act as constraints on path distance and path reduction.

During the last few years, several (and in most regards, similar) approaches have been proposed: Euler [35], Velvet [47], AllPaths [4, 23], SOAPdenovo [21],

ABySS [43]. Here, we focus on the most significant (and probably most widely employed) two: Velvet and ABySS. After describing Velvet and ABySS in more detail, we provide some final considerations valid for *all* DGB assemblers.

Velvet Velvet [47] is a DBG assembler proposed in 2008. The software targets de novo assembly from short reads from the Solexa platform. A newly released extension provides compatibility with SOLiD reads.

Velvet begins with the construction of a $k-$mer graph. Then, it applies a series of heuristics in order to reduce graph complexity. Extensive use of graph simplifications and other tricks simplify calculations and compress the amount of treated data without loss of information.

Velvet enters a simplification phase (which essentially consists in singleton elimination) during graph construction and again several times during the assembly process. A well-established parameter determines the minimum number of occurrences in the reads for a $k-$mer to qualify as a graph node. Spurs are removed iteratively. Bubbles are searched using a breadth-first approach, fanning out as much as possible, starting at nodes with multiple out-going edges. Since the assembly of real data can generate bubbles within bubbles, an exhaustive search for all bubbles is impractical. The search is bounded to make it tractable. In general, the criterion determining the target path is the higher read multiplicity: a consensus algorithm is usually adopted. This approach may lead to elimination of real sequence differences due to polymorphism in the original DNA as well as to over-collapse of near-identical repeats. Graph complexity is further reduced by read threading. This removes paths representing fewer reads than a threshold. This operation risks removing low-coverage subsequences but it is thought to remove mostly spurious connections induced by convergent sequencing errors.

The final graph reduction involves mate pairs. In a nutshell, Velvet operates on pairs of long contigs (simple paths) connected by mate pairs. Using the long contigs as anchors, it tries to fill the gap between them with short contigs. It gathers short contigs linked to the long contigs, and applies a breadth-first search through the DBG for a single path linking the long contigs and by traversing the short contigs.

Several "simplifying" iterations may be run per data set to optimize selection of three critical parameters: the value of k, the minimum frequency f_{k-mer} expected by a "genuine" $k-$mer, the estimated coverage c of the genome. f_{k-mer}

determines which $k-$mers are pruned a-priori. c controls spurious connection breaking.

To sum up, Velvet aims at providing a complete implementation of DBG assembly. It does not perform any error-correction preprocessing; however, it does have an error-avoidance read filter. Its heuristics exploit several aspects: graph topology, read coverage and – even if the approach could be still improved – mate pairs. However, locality is not adequately exploited by Velvet. Until now, the prohibitive memory requirements have precluded the use of Velvet in assembling very large genomes.

ABySS ABySS [43] is a distributed, parallel implementation of DBG assembly developed starting from 2009. It attemps to explicitely address memory limitations of DBG approaches in large genome assembly.

Storage required by the $k-$mer graph's nodes and by any computation over them is distributed over a computational grid formed by several interconnected nodes. In general, the overall storage capacity of such a grid is quite large. The algorithm tries to exploit CPU parallelism, partitioning the assembly problem into multiple independent instances and then merging the results.

ABySS uses a compact representation of the $k-$mer graph. Each graph node represents a $k-$mer and its reverse complement. Assembly partitions are performed at the level of individual graph nodes. Each graph node is processed separately; for efficiency, many graph nodes are assigned to each CPU. The assignment of a graph node to a CPU is accomplished by converting the $k-$mer to an integer. The formula is strand-neutral, so that a $k-$mer and its reverse complement map to the same integer. However, this approach does not guarantee that neighbouring $k-$mers are necessarily mapped to the same CPU. Each graph node keeps 8 bits of extra information: representing the existence or non-existence of each of the four possible one-letter extensions at each end. In this way, the graph edges are represented implicitly.

Paths along nodes are followed in parallel, starting at arbitrary graph nodes per CPU. From any node, ABySS finds its successor elegantly: the node's last $k-1$ bases, plus a one-base extension indicated by an edge, is converted numerically to the address (including CPU assignment) of the successor node. When a path traverses a node on a different CPU, the process emits a request for information. Since inter-CPU communication is typically slow, the process works on other

graph nodes while waiting for the response.

In order to compress the amount of processed data, ABySS essentially adopts the same graph simplification heuristics already proposed by Velvet – introducing a high degree of parallelism. Spurs shorter than a fixed threshold are iteratively removed. Bubbles are discovered (and eventually removed) by bounded search; paths supported by the highest number of reads are preferred. After obtaining the contigs from the final graph, ABySS performs a post-processing phase. During this phase, it exploits information about mate pairs in order to merge contigs into longer sequences.

As of 2011 ABySS is considered the most scalable and efficient assembly software for SOLiD short reads.

Considerations DBG assemblers provide a fairly robust approach to assembly. They rely to some extent on a set of overlaps between the input reads, representing such overlaps by means of a directed graph.

DBG algorithms resemble old OLC approaches in many regards – the different graph representations adopted are similar, if not equivalent. The greatest difference between DBG and OLC approaches is their computational efficiency, which is strictly related to the sequencing technology they address. While OLC techniques directly incorporate connections (overlaps) of varying length, $k-$mer graphs are limited initially to short connections of uniform size (i.e., the shared $k-$mers). In general, DBG algorithms avoid computationally expensive all-against-all pair-wise read comparisons; furthermore, they avoid loading all the (replicated) sequences associated with high-coverage sequencing and thus tend to have a memory footprint proportional to the size of the genome times the size of a$k-$mer (rather than to the size of the input – which can be substantially larger when coverage is high).

1.3.3 Open issues

NG sequencers offer a huge amount of sequenced data with high throughput and low costs. This data is provided by fragments of sequenced genome – the reads. Reads are usually short. Further, no NG technology ensures error-free products: reads always contain an amount of erroneous base (or colour) calls. These shortcomings are usually addressed by ensuring high coverage and redundancy. There-

fore, de novo assembly algorithms must face two challenging problems: noisy data and storage and computational efficiency.

Noisy data is a crucial problem. All DBG approaches are extremely sensitive to sequencing errors – every wrong base call may introduce up to k erroneous nodes. Sequencing errors may induce false positive and false negative overlaps. Almost all current techniques employ pre-processing steps to filter or correct unconfirmed portions of reads, as well as post-processing to repair graphs by smoothing and threading. In general, these techniques are computationally expensive. We address this problem by introducing a novel, effective approach to read filtering. Chapter 2 presents our filtering technique in detail.

Current DBG assemblers proliferated with the introduction of short reads. In general, they target uniformly sized reads in the 25-100 bp range; they rely on redundant, high coverage data to mitigate this shortcoming. This requires assemblers to deal with an impressive amount of data – posing strict constraints on storage and computational efficiency. This is unlikely to change in the near future. Almost certainly, data volume will continue to increase as sequencing costs decline. The next-generation technology will surely be applied to larger and larger genomes [8]. Reads of the near future may be intermediate-sized, matching more closely the expectations of OLC assemblers. Even very long reads may become feasible, while short reads may become even more affordable.

Our novel, locality-aware assembly algorithm presented in Chapters 3 and 4, with its very low memory requirements and cache-efficient data access, is a perfect match for the requirements of current – and future – sequencing technologies.

Chapter 2

Robust filtering of sequenced data

This Chapter introduces an innovative technique for filtering reads (i.e., for distinguishing between reads with and without errors) in colour space. We validate the effectiveness of our scheme both theoretically – through formal analysis and direct comparison against other commonly utilized schemes – and experimentally – performing extensive experiments over a SOLiD dataset containing more than 50 million reads. After briefly contextualizing the filtering problem and presenting a widely employed technique (Sasson's filter), Section 2.1 introduces our scheme, the *probabilistic filter*. Section 2.2 experimentally compares Sasson's method against ours; in particular, we show how the latter clearly outperforms the former over a wide range of crucial parameters (such as the $k-$mer length).

2.1 Filtering SOLiD output

Filters aim at discriminating between *correct* and *erroneous* reads. In general, effective filters help assemblers in both bounding resource consumption and improving final output quality. After contextualizing the filtering problem for SOLiD technology output (Subsection 2.1.1), we describe the commonly used Sasson's filter (Subsection 2.1.2). Then, we present our probabilistic filter in its basic formulation (Subsection 2.1.3). Finally, we extend our technique with a preprocessing phase in order to improve its effectiveness (Subsection 2.1.4).

2.1.1 SOLiD reads and quality values

SOLiD platforms do not prefilter low-quality reads. All identified reads, even if with poor quality, are reported after primary analysis. This becomes a crucial problem when dealing with de novo assembly of short read sequences, which are highly sensitive to sequencing errors. Further, the elimination of reads containing errors may noticeably improve efficiency of assembly procedures. A common approach is to mitigate these errors prior to assembly, making use of additional information provided by SOLiD output – the *quality values*.

The sequencing errors commonly observed are of two types: polyclonal/correlated errors and independent, erroneous colour calls [44]. Polyclonal and correlated errors occur when the entire read is of poor quality or missequenced. Polyclonal reads may lead to assembled sequences which are hybrid, with no match in the true genome. Single colour call errors are independent and can occur multiple times in the sequence also leading to an inaccurate sequence.

The probability of single colour call errors can be calculated by means of *quality values* (QVs). The SOLiD platform outputs two types of files after primary analysis: a sequence file in *colour space* (see Chapter 1) and a quality file containing the corresponding QVs. QVs are calculated by training the sequencing process parameters (such as image intensity and a noise to signal value) against several annotated datasets. In a nutshell, they represent the probability for colour calls to be inaccurate: the higher the QV, the higher the confidence in the colour call accuracy. More specifically, given a quality value q_i for the ith colour call, its error probability p_i is given by:

$$p_i = 10^{-\frac{q_i}{10}} \tag{2.1}$$

Consequently, $1 - p_i$ represents the probability for the ith colour call of being correct. Quality values are theoretically drawn from the $[0, \infty)$ interval but, in practice, values above 40 are rarely obtained.

2.1.2 Sasson's filter

Sasson's filter [39] attempts to optimize the assembly preprocessing phase by identification and removal of reads containing errors for the SOLiD platform. In order to do this, it makes intensive use of the QVs provided from the SOLiD's primary analysis.

In a nutshell, its goal is to eliminate low-quality reads and thus pass only high-quality data into the assembly applications. Sasson's filter targets both the types of errors that can occur during SOLiD sequencing: polyclonal/correlated errors and erroneous colour calls (see Subsection 2.1.2).

The working principle it follows is quite straightforward; it implements a simple *boolean* filter that discriminates correct reads on the basis of a set of *filter conditions* – such as polyclonal analysis or error analysis quality values thresholds. When applied to an input dataset containing n reads, it generates two output sets:

- the set R_f of reads satisfying the filter conditions;
- the set $R_{\bar{f}}$ of reads violating the filter conditions.

Obviously $|R_f| + |R_{\bar{f}}| = n$.

An adequate assessment of filter conditions is crucial. In general, filter conditions depend on a set of tunable parameters. In the following, we will refer to these parameters with the following notation:

- α is the quality value threshold for polyclonal analysis;
- ℓ_c is the maximum number of colours below α tolerated;
- β is the quality value threshold for error analysis;
- e_c is the maximum number of colours below β tolerated;
- t is the number of colours considered for error analysis. Only the *first* t colours of the read are considered for error analysis, and only the first 10 are for polyclonal analysis.

Filter conditions immediately follow from these parameters. In particular, a m-length read r constituted by colours $c_1 \dots c_m$ satisfies filter conditions if and only if both the following statements hold:

- At most ℓ_c colours in $c_1 \dots c_{10}$ have quality values smaller than α;
- At most e_c colours in $c_1 \dots c_t$ have quality values smaller than β.

Some experimental results obtained by this setup will be reported in Section 2.2.

2.1.3 Probabilistic filter

Our approach adopts a probabilistic perspective. In a nutshell, we consider the error probability associated to each colour call (see Equation 2.1); on this basis, we estimate the expected number $\tilde{n}$ of error free reads in the whole dataset. In its basic version, our filtering scheme directly utilizes the QVs provided by the SOLiD's primary analysis. In its extended version, it adjusts such QVs during a preprocessing phase; we shall discuss this extension in Subsection 2.1.4.

Our *probabilistic* filter relies on a series of simple hypotheses and consists of three main steps. First, we assign every read a error probability p_r; then, we sort all the reads in the input dataset according to their error probabilities; finally, we estimate a filtering threshold discriminating the correctness of any read.

Our approach makes a series of assumptions:

- error probabilities are independent from each other. Therefore, within a single read, any quality value is an assessment on the reliability of the corresponding colour *only*;
- the quality value assigned to a given colour is *independent* from the colour itself. That is, given a quality value q_i associated with the colour c_i, we may state that c_i is wrong with probability p_i (where p_i is calculated according to (2.1)) and correct with probability $1 - p_i$. For instance, let us suppose that c_i is equal to 1; if the quality value q_i for c_i is equal to 20, then we have that $p_i = 0.01$. Consider now that c_i is equal to 3, while q_i remains unchanged; the estimated probability q_i would remain unchanged as well.

It is important to observe that neither of these two assumptions is *completely* true in practice. However, with these two assumptions we can construct a simple model for the overall error probability p_r for a m-long read $r = c_1 c_2 \dots c_m$ that is easily tractable and (as we shall see later in this Chapter) not too inaccurate. Since (single colour) error probabilities are independent, we calculate p_r as follows:

$$p_r = 1 - \prod_{i=1}^{m} (1 - p_i) \tag{2.2}$$

Clearly r is correct with probability equal to $1 - p_r$.

Error probabilities associated to reads – and not only to colours – let us adopt a well defined sorting criterion for reads. In particular, given any pair of reads

r and s, we state that r is *better* than s if and only if $p_r < p_s$. In this way, we organize *all* the reads belonging to the input dataset into an ordered list. The better the read, the lower is the probability that it contains no errors.

As a final step, we define a threshold value as a *rank* in the list. Reads ranked better are considered error free; reads ranked worse are considered to contain errors. Our threshold rank is obtained in a very simple but effective fashion. It simply corresponds to the *expected* number of error free reads. More precisely, consider a sequence of n reads $r_1, r_2, \ldots, r_n$. On the basis of our assumptions, given any pair i, j, p_{r_i} and p_{r_j} are independent. The expected number $\tilde{n}$ of correct reads is:

$$\tilde{n} = \sum_{i=1}^{n} p_{r_i} \tag{2.3}$$

Our filtering threshold is then set equal to $\tilde{n}$.

2.1.4 Preprocessing $k-$mers

We improve the effectiveness of our probabilistic filter by introducing a preprocessing phase. In a nutshell, we try to *guess* whether different reads are actually samples of the *same* region in the original genome – that is, if they are "*twin*" reads. We perform this guess by looking for *identical* $k-$mers belonging to *distinct* reads. Then, we use the retrieved information in order to adjust the quality values of the colour calls in twin reads. As we shall see, this preprocessing can noticeably improve the performance of our filtering scheme.

Our preprocessing phase focuses on $k-$mers rather than on reads. The whole input dataset is decomposed into $k-$mers; then, an all-against-all comparison is performed among these $k-$mers, partitioning them into equivalence classes of identical $k-$mers (note that two $k-$mers may be identical but have different quality values associated to their colours).

For a generic set of m identical $k-$mers belonging to m distinct reads, let q_i^j be the quality value associated to the i^{th} colour c_i in the j^{th} $k-$mer. We assign to the i^{th} colour of the m $k-$mers a *partial pseudoquality* Π_i equal to the sum of the m qualities of that colour in the m $k-$mers:

$$\Pi_i = \sum_{j=1}^{m} q_i^j \tag{2.4}$$

Then, we recalculate the error probability of each read by utilizing the partial pseudoqualities, instead of the qualities, of the colours appearing in its $k-$mers. Note that multiple $k-$mers taken from the same read may "cover" the same colour – for example, the 3^{rd} colour in a read with $r \geq k+2$ colours falls for $k \geq 3$ within 3 $k-$mers (starting, respectively, with the first, second and third colour of the read). *The maximum partial pseudoquality becomes the pseudoquality of the corresponding colour in the read.* We then recompute for each read its error probability using the pseudoqualities of its colours instead of the actual qualities, and, as in Subsection 2.1.3, rank the reads and discard those below the rank threshold – *which, importantly, is not recomputed.*

Thus, if we have multiple copies of the same $k-$mer, we assess the probability that the i^{th} colour is incorrect as the product of the probabilities that it would be incorrect in each copy of the $k-$mer (remember that the quality is proportional to the logarithm of the error probability). Note that, in addition to disregarding error correlations between different reads (as in our basic probabilistic filter), this both underestimates and overestimates the real error probability. The underestimation stems from disregarding the evidence of $k-$mers that are not quite, but almost, identical, and in particular coincide on a given colour. The overestimation stems from the fact that only positive evidence contributes to our probability estimation – in other words, as the number of reads grows, the probability of witnessing a large number of copies of *every* $k-$mer, whether "correct" or not, also grows. However, we are interested only in a *ranking* of the various reads, which mitigates the effects of the two errors.

Finally, note that the assessment of the pseudoquality of a colour in a read as the maximum of the pseudoquality of the colour in the corresponding $k-$mers is also a very simplistic heuristic that handwaves the necessity of carefully balancing the strong positive correlations between the pseudoqualities of a colour in partially overlapping $k-$mers. But, again, the "unbiased" nature of this error paired with our interest only in the ranking of the various reads suggests its effects might be limited; and in the next Subsection we show this is indeed the case.

2.2 Experimental evaluation

This Section compares the effectiveness of Sasson's filter against that of our filter in discriminating correct reads. After describing the dataset our experiments rely on, we present the results obtained by Sasson's filter and by our "basic" and "extended" probabilistic filter.

Dataset The input dataset we adopt is a 600× [1] of the *DH10B* fragment of *Escherichia Coli*, provided by the Applied Biosystems SOLiD System Open Source Software Tools framework [2]. It contains 57254192 reads, paired in half as many mate pairs. Each read is 50 bases long.

Sasson's filter results We run Sasson's filter with the following parameters (see Subsection 2.1.2 for an exhaustive description of them):

$$\alpha = 25 \quad \beta = 10 \quad \ell_c = 3 \quad e_c = 2 \quad t = 2$$

These values correspond to the recommended ones [39]. Table 2.2 summarizes the results we obtained, also represented in Figure 2.1 and Figure 2.2

	Reads	**Pairs**
Filtered reads	7242058	7242058
Aligned reads	4076674	1509278
% of aligned reads	56.291%	20.8405%

Table 2.1: Reads filtered and aligned by the Sasson's filter.

Filtered reads denote reads satisfying the filter conditions; *aligned* reads denote reads matching actual subsequences belonging to the original genome. We provided the alignment results for single reads and for mate pairs as well. Only half of the filtered reads are actually aligned reads. Further, note that only 7M reads (about 12.9% of the entire dataset) are contained in the filter output dataset. However, these values do not take into account *orphan* reads – that is, reads which satisfy the filter conditions but whose associated mate violates the conditions themselves.

[1] 600× denotes the estimated (provided) coverage

[2] `http://solidsoftwaretools.com/gf/project/`

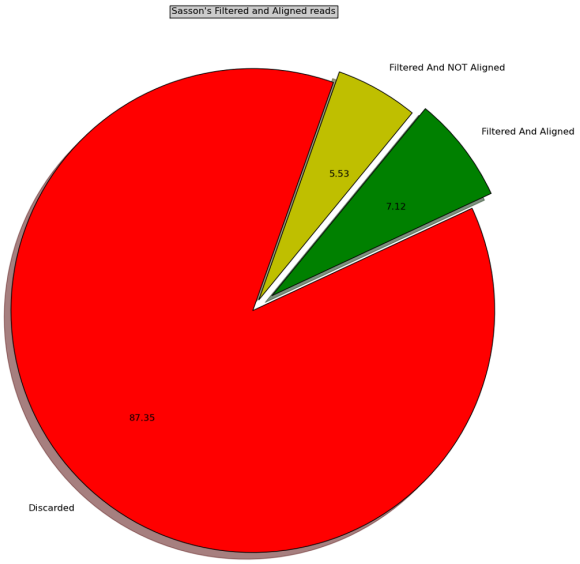

Figure 2.1: In red (the largest slice of the pie) the fraction of reads discarded by Sasson's filter. In green (the lowest of the two smaller slices of the pie) the fraction of filtered reads actually aligning to the reference genome.

Basic probabilistic filter results Results obtained by our filter are provided in Figure 2.3. The solid curves are obtained by first sorting reads and then counting at position i (on the x-axis) how many reads, out from the i top ranked, align to the reference. The green dash and dots curve is the theoretical value obtained using (2.3) with p_{r_i} obtained from (2.2). Note how the two curves are similar but do not perfectly match; the gap between red and blue curves suggests a *pessimistic estimation* of the quality values by the sequencer.

The dotted blue line represents an ideal "perfect" filter that would rank any error free read better than any read containing at least one error; slightly less than 20% of all reads turn out to be error-free (i.e. perfectly aligned to the genome). Note that the best 20% of all reads according to the probabilistic filter capture about two thirds of these error-free reads.

Extended probabilistic filter results We performed a preliminary experiment in order to evaluate the actual effectiveness of our preprocessing phase. We divided all the reads belonging to the input dataset into 255 classes – on the basis of the error probabilities obtained from their pseudoquality values. For

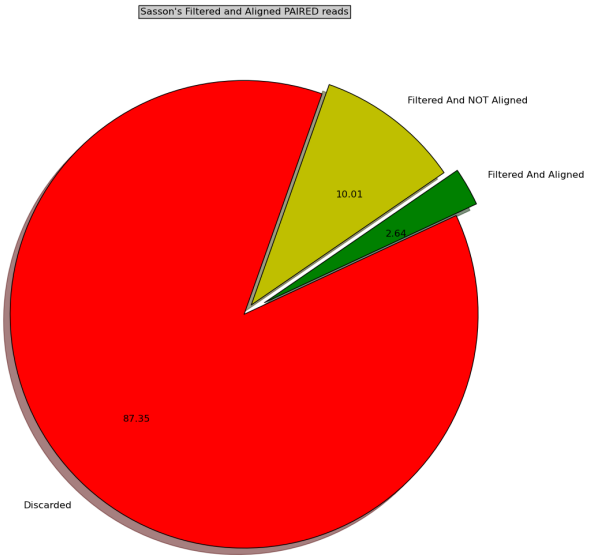

Figure 2.2: In red (the largest slice of the pie) the fraction of reads discarded by Sasson's filter. In green (the smallest slice of the pie) the fraction of reads that are not discarded and can be aligned to the reference genome together with their mate.

each of these classes, we counted the number of reads it contains and the number of aligned reads it contains. Figure 2.4 shows the results obtained upon a subset of the 600× *E. Coli* dataset containing a million reads (approximately 10× coverage). Figure 2.5 and Figure 2.6 present the results we have obtained for a subset of 5M reads (approximately 50× coverage) and for the whole dataset. We performed multiple experiments, adopting different values for k.

Values drawn in Figure 2.6 are reported in Table 2.2, for a $k-$mer length equal to 25. Considering only the reads belonging to the 255^{th} class (which is the best one), we have a set of about 8 million reads (slightly more than 14% of the whole 600× dataset). Interestingly, reads belonging to this class align to the reference genome in more than 95% of the cases.

On the basis of these results, we performed a series of experiments using our extended probabilistic filter. We compared the results we have obtained against

Quality class	Occurrences	Alignments	Percentage of Alignments
0	758002499	177477	0.000234
1	22673312	89559	0.003950
2	10075851	37948	0.003766
3	5905308	33773	0.005719
4	4337040	10048	0.002317
5	3385776	8944	0.002642
6	3067492	8471	0.002762
7	2281080	7816	0.003426
...	...	...	...
108	18595	4067	0.218715
109	13043	3626	0.278004
110	12896	3719	0.288384
...	...	...	...
255	8055499	7658329	0.950696

Table 2.2: Occurrences and alignments of reads clustered according to their quality classes. Percentage of alignment corresponds to the ratio between alignments and occurrences.

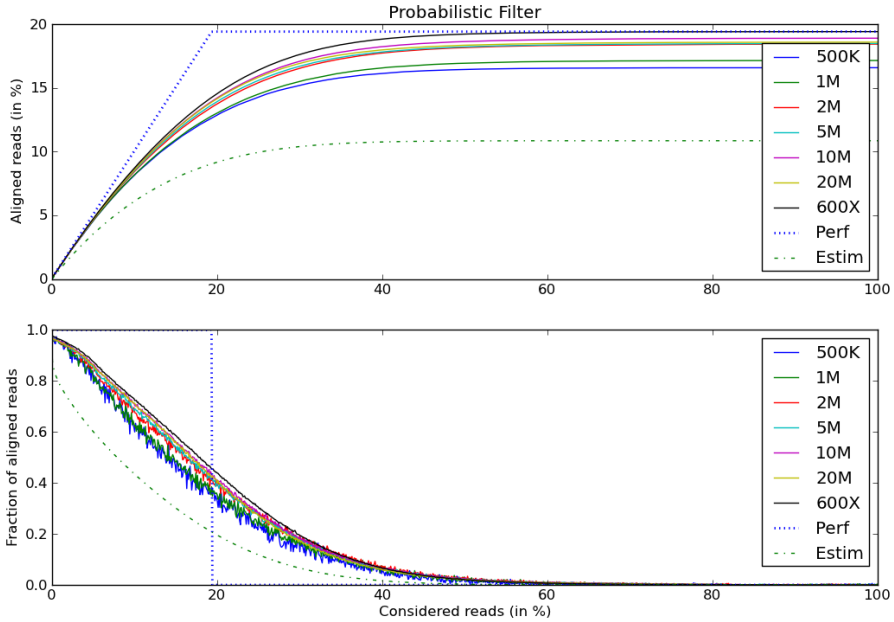

Figure 2.3: Theoretical (dash and dots line) vs. measured (solid) values for the number of correct reads. The dotted line represents an hypothetical perfect filter.

those provided by the "basic" filter and by the ideal perfect filter for different values of k. Here, we present the results obtained on the whole dataset (see Figure 2.10) and on subsets of $500K$ reads (approximately $5\times$ coverage – see Figure 2.7), $5M$ reads (approximately $50\times$ coverage – see Figure 2.8) and $20M$ reads (approximately $200\times$ coverage – see Figure 2.9).

In addition to a function corresponding to that of Figure 2.3, each figure also shows (below) what is effectively the function's derivative – the fraction of reads ranked at a given percentile that can be aligned to the genome. Note that for the perfect filter this means the totality of all reads up to the percentage of correct reads ($\approx 20\%$) and no reads beyond the point. For comparison, we also show the results of Sasson's filter.

The more is the size of the dataset to filter, the higher is the impact of preprocessing. In any case, preprocessing introduces an improvement; but its effects are not evident when the coverage is small ($5\times$). There is a simple explanation for this. Even if $k-$mers, being significantly shorter than a read, have a lower error probability, the majority of $k-$mers of 20 or more colours are incorrect. The

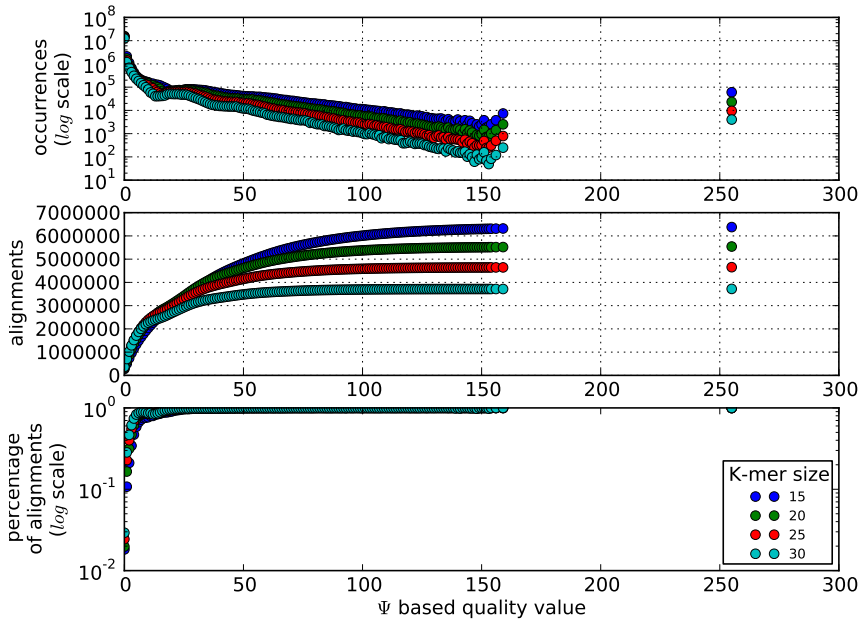

Figure 2.4: Number of occurrences, alignment and percentage of aligned reads in a set of 1M reads after applying the extended probabilistic filter.

effects of preprocessing only begin to show when at least a few ("correct") copies of each $k-$mer are present and can boost each other's pseudoqualities, which does not happen at $5\times$ coverage.

The case shown in Figure 2.10 (with $k = 15$) deserves particular attention. A slight, but noticeable degeneration in the overall performance appears. $k-$mers of this length are sufficiently short that the "same" $k-$mer begins to have a significant probability of appearing in two unrelated reads – and in particular between an error-free one and one containing at least one error. This allows a minor fraction of incorrect reads to see the pseudo-quality of their colours boosted by the identical $k-$mers of a correct read, bringing their rank within the acceptance threshold.

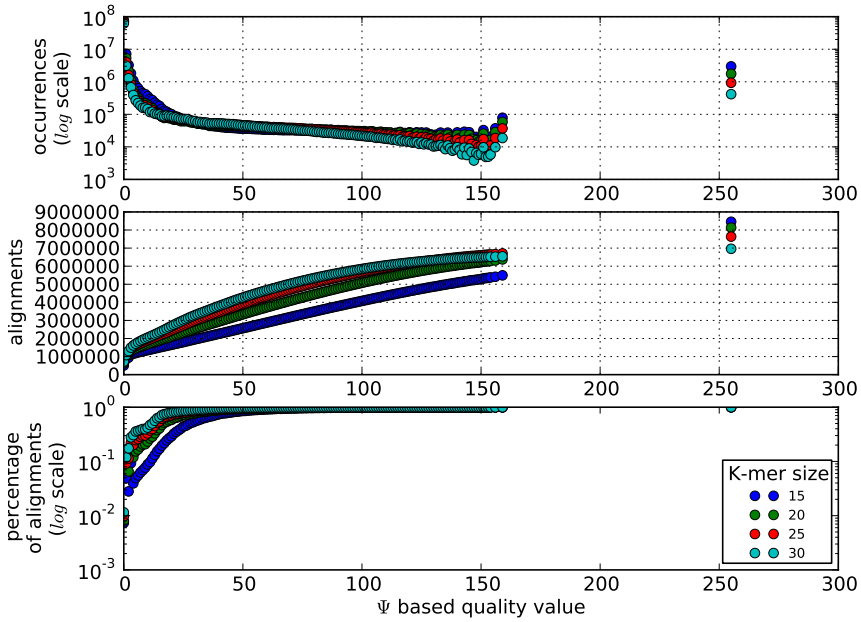

Figure 2.5: Number of occurrences, alignment and percentage of aligned reads in a set of 5M reads after applying the extended probabilistic filter.

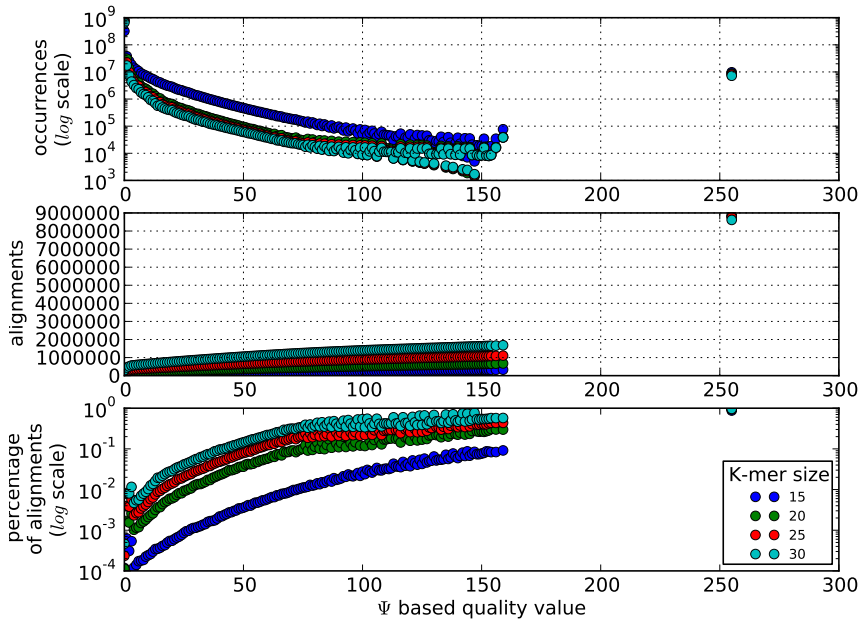

Figure 2.6: Number of occurrences, alignment and percentage of aligned reads in the whole input dataset after applying the extended probabilistic filter.

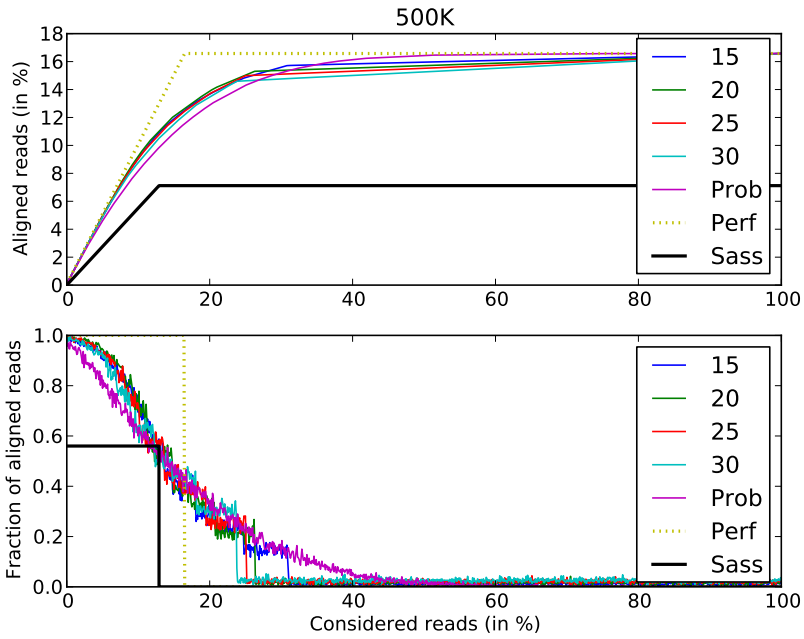

Figure 2.7: Comparison among perfect (dotted line), basic and extended probabilistic filters over a set of $500k$ reads, for different values of k (solid curves). "Sass" depicts the behaviour of Sasson's Filter.

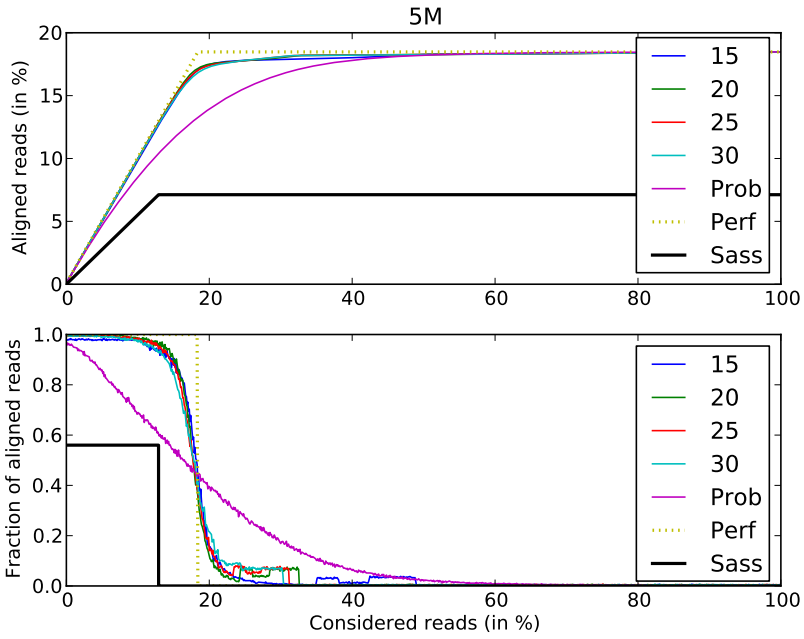

Figure 2.8: Comparison among perfect (dotted line), basic and extended probabilistic filters and Sasson's filter over a set of $5M$ reads, for different values of k (solid curves). "Sass" depicts the behaviour of Sasson's Filter.

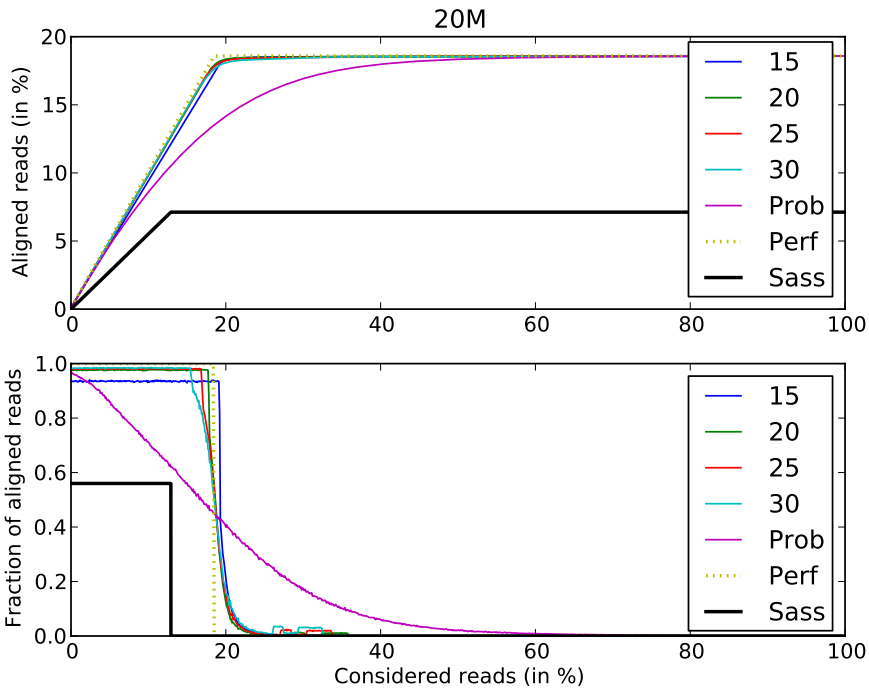

Figure 2.9: Comparison among perfect (dotted line), basic and extended probabilistic filters and Sasson's filter over a set of $20M$ reads, for different values of k (solid curves). "Sass" depicts the behaviour of Sasson's Filter.

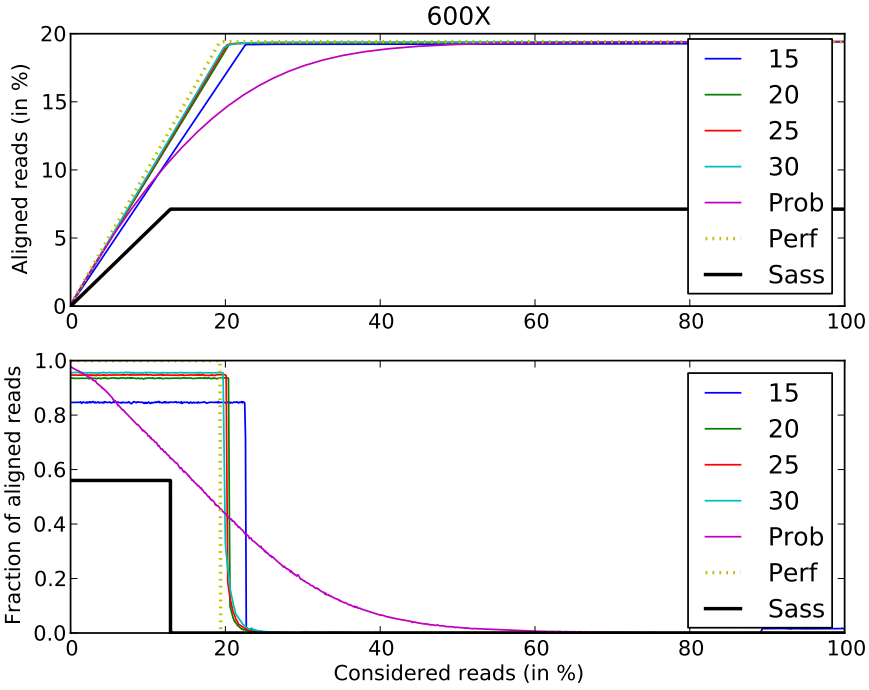

Figure 2.10: Comparison among perfect (dotted line), basic and extended probabilistic filters and Sasson's filter over the whole dataset, for different values of k (solid curves). "Sass" depicts the behaviour of Sasson's Filter.

Chapter 3

A novel algorithm for de novo DNA assembly

This Chapter introduces a novel algorithm for de novo DNA assembly, and provides a theoretical evaluation. Compared to existing algorithms, our algorithm focuses on a careful exploitation of the memory hierarchy; this allows it to run considerably faster and on cheaper hardware than the competition. The algorithm is fairly complex; thus, to ease the burden on the reader, after a brief review (in Section 3.1) of the notation and terminology used throughout the rest of the Chapter we begin by presenting (in Section 3.2) and analyzing (in Section 3.3) a simplified version of our algorithm that assumes no repeated $k-$mers in the target DNA and works in base space (rather than in colour space). We then show (in Section 3.4) how this assumption can be lifted with only a modest performance overhead; and we sketch how the algorithm can be easily translated to colour space, and parallelized. All results in this Chapter are of a theoretical nature. The implementation details and the experimental performance evaluation can be found in the following Chapter.

3.1 Notation

Throughout the rest of the Chapter, we denote by:

- r the length of a read;
- k the length of a $k-$mer; this means that the number of different $k-$mers in a read is $r - k + 1$, which we assume is of the same order as r;

- g the length of the genome;
- n the total number of reads;
- $c = \frac{n \cdot r}{g}$ the *coverage* of the genome; we assume $c >> 1$ (values between 10 and 1000 are all realistic with modern short read technology).

All sizes are expressed in bases. Two bits are sufficient to represent a base, but in some cases, including many "standard" formats used in DNA-databases, a full 1 byte `char` is used to represent a base. This does not change the asymptotic complexity of the analysis, but it can mean a substantial performance difference in practice – see the next Chapter for more details.

Finally, given a generic set (of not necessarily distinct) $k-$mers S, we denote by $\rho(S)$ the set of reads with at least one $k-$mer in S. We shall also abuse the notation slightly and denote, for any give $k-$mer s, by $\rho(s)$ the corresponding read.

3.2 Assembly in the absence of repeated $k-$mers

As briefly mentioned in Chapter 1, one fundamental limitation of algorithms based on $k-$mers and De Bruijn graphs is high memory consumption paired with fairly "scattered" accesses to the memory space. In a nutshell, this depends on the fact that the De Bruijn graph approach generates, for each read, $r - k + 1$ $k-$mers, each of which then requires random access in a memory space of size $\Theta(k \cdot g)$ (since the De Bruijn graph has one node, corresponding to a $k-$mer, for each base in the genome). The goal of our algorithm is twofold: to reduce the number of $k-$mers generated (and stored), and to increase the spatial locality of accesses.

To do so, we examine only a small fraction of all the $k-$mers at a time. In particular, we assume the existence of a hash function that, roughly speaking, maps the set of all $k-$mers uniformly into the set of the first h non-negative integers with $h \approx k$. More formally:

Definition 1. *Consider a set R of n reads. Denote by K the set of all (not necessarily distinct) $n(r - k + 1)$ $k-$mers of R. Consider a function $\mathcal{H}_h : K \to 0, \ldots, h-1$. $\mathcal{H}_h$ is a* balanced $h-$hash *for R if, for each $i = 0, \ldots, h-1$,*

1. $|\mathcal{H}_h^{-1}(i)| \leq \frac{2|K|}{h}$

2. $|\rho(\mathcal{H}_h^{-1}(i))| \geq \frac{|R|}{2}(\min(1, \frac{r-k+1}{h}))$.

Note that a balanced $h-$hash will not, in general, exist for every possible set of reads in a genome; in fact, it may not even exist for *any* set of reads of a particular genome (this is obvious in the case of a genome consisting of a single base repeated g times). However, if both bases and reads within the genome are sufficiently "well distributed" there are many simple heuristics that appear to work well (see the next Chapter 4). We can now define the crucial notion of (level i) anchor.

Definition 2. *Consider a set R of n reads. Let K be the set of all (not necessarily distinct) $n(r-k+1)$ $k-$mers of R, and let $\mathcal{H}$ be a balanced $h-$hash for R. A $k-$mer x is* an anchor at depth *i for its read if $\mathcal{H}(x) = i$.*

A high level view of our algorithm is as follows. First, it chooses a balanced $h-$hash, with $h \approx k$. The exact choice of h makes no difference in terms of asymptotic performance as long as it is within a constant factor of k. Next, our algorithm partitions all reads based on the depth of their deepest anchor(s). Then, it begins by merging those reads anchored at level $h-1$ into contiguous snippets of DNA, which we call (depth $h-1$) *bundles*, with the property that every read anchored at depth $h-1$, and no other, falls entirely within a level $h-1$ bundle. The algorithm then repeats the process at depth $h-2$, considering those reads whose anchors all lie at depth $h-2$ or less; it then attempts to merge the depth $h-1$ and depth $h-2$ bundles. This process of "bundling" reads and merging bundles is then repeated out all the way to depth 1.

Let us now look in greater detail at the creation of the depth $h-1$ bundles. All reads involved have at least one depth $h-1$ anchor; we refer to those with more than one such anchor as *bridge* reads. Note that sharing a bridge is a symmetric relationship between depth $h-1$ anchors; the reflexive and transitive closure of such a relationship partitions the set of all depth $h-1$ anchors, and the individual partitions coincide with what we refer to as bundles. More formally:

Definition 3. *A depth i* bundle *is a non-empty set $\mathcal{B}$ of depth i anchors, such that, for any two depth i anchors x' and x'' sharing a bridge, $x' \in \mathcal{B} \rightarrow x'' \in \mathcal{B}$.*

Algorithm 1 A high level view of our algorithm

Choose a balanced $h-$hash with $h = \Theta(k)$

for all reads **do**

Determine a read's deepest anchor(s)

end for

for i=$h-1, \ldots, 0$ **do**

Merge reads anchored at depth i (and not deeper) into temporary i-bundles.

Merge temporary i-bundles and final $(i+1)$-bundles into final i-bundles.

end for

In a dual fashion, this also partitions the reads anchored at depth $h-1$, with each read being "attached" to a single bundle (since all its anchors belong to the same bundle).

Note that the genome implicitly defines a total ordering between different anchors – the order in which we would encounter them if we scanned the genome e.g. from left to right. To assemble the bundles, the algorithm maintains a compact data structure associated to each (depth $h-1$) anchor; these data structures are kept in a hash table indexed by the anchors themselves. Each entry holds:

1. The anchor.
2. Up to $r-k$ bases to the left of the anchor (which we refer to as the *left support* of the anchor).
3. Up to $r-k$ bases to the right of the anchor (which we refer to as the *right support* of the anchor).
4. A pointer to the depth $h-1$ anchor in the same bundle immediately to the left, if it exists and has already been identified.
5. A pointer to the depth $h-1$ anchor in the same bundle immediately to the right, if it exists and has already been identified.
6. The offset between the anchor and the anchor of the previous point (note that this offset is between 1 and $r-k$ – and if it is less than $h-1$ the two anchors may partially overlap)

For efficiency reasons, the right support of an anchor is truncated at the beginning of the next anchor to the right (if any); if the next anchor to the right has offset $r - k$ or less it is omitted entirely. Similarly, if the next anchor to the left has been identified, the left support is omitted (any intervening bases are recorded in that anchor's right support). Whenever a new read is processed, its anchor(s) are examined; the corresponding entries are entered into the hash table or, if already present, possibly updated in terms of support and/or pointers.

Algorithm 2 Temporary depth i bundle formation

```
for all reads anchored at depth i and no deeper do
  for all depth i anchors in read do
    if anchor is in HashTable then
      update anchor
    else
      insert anchor in HashTable
    end if
  end for
end for
```

Merging depth i temporary bundles with depth $i + 1$ final bundles to obtain the depth i final bundles is extremely easy, and hinges on the notion of *extremal* read.

Definition 4. *The extremal reads of a bundle B are the leftmost and rightmost r bases of B (note that these two sequences may coincide, if the bundle spans only r bases; and that in any case each corresponds to an actual read merged into the bundle).*

All one has to do is to add to the depth i bundling process, all reads that are currently *extremal* for some depth $i + 1$ bundle and are also anchored at level i. If one such extremal read shares an anchor with a read anchored at depth i, the respective bundles are merged. All final depth $i + 1$ bundles and all temporary depth i bundles that are not merged at this stage become final depth i bundles, too. Note that, in practice, this step only requires one to carry out the extremal read merge mentioned above; although we are conceptually "promoting" bundles, this requires no actual updates to the data structures.

Algorithm 3 Final depth i bundle formation

for all extremal reads of final depth $i+1$ bundles anchored at level i **do**
attempt merge into a temporary depth i bundle, linking the two bundles.
end for

3.3 Theoretical analysis

This Section provides a theoretical analysis of our algorithm, proving its correctness and providing asymptotic bounds on its performance. The cornerstone of our correctness analysis is the following:

Theorem 1. *Every final depth i bundle represents a contiguous sequence of bases that does not overlap with any other final depth i bundle nor with any temporary depth j bundle with $j < i$ save possibly for its extremal reads.*

Proof. To prove that every final depth i bundle represents a contiguous sequence of bases, consider two genome positions corresponding to two bases b' and b'' both in the same bundle. If b' and b'' both belong to the same depth i read anchored to some anchor in the bundle, the result is immediate. Otherwise, they belong to two distinct reads with depth i anchors x' and x'' both in the same bundle. By the definition of bundle, there exists a sequence of anchors $x_1, \ldots, x_m$ such that $x' = x_0$, $x'' = x_m$, and there exists a bridge read z_i in the bundle between x_i and x_{i+1} for all positive i less than m. Then, denoting by w_i the infix of z_i between x_i and x_{i+1}, the sequence $x_1, w_1, \ldots, x_{m-1}, w_{m-1}, x_m$ is a sequence of contiguous bases in the bundle stretching from x' to x''. Therefore, the bundle holds a sequence of contiguous bases stretching between b' and b''.

Moreover, there is a sequence of contiguous bases stretching from any anchor x' in the same read as b' and any anchor x'' in the same read as b'', with the property of being $r-k$ *spaced* – i.e. the distance between the leftmost bases of two consecutive anchors in the sequence is at most $r-k$, the length of a read minus that of a $k-$mer. Note that in an $r-k$-spaced sequence, *any* contiguous subsequence of length r (or greater) contains at least one full anchor. This helps us prove that the sequence of contiguous bases corresponding to a final depth i bundle does not overlap with that of any other final depth i bundle, except for (at most) the bases in their respective extremal reads.

Suppose the overlap did indeed happen, and suppose that a base of the first bundle b' that is not part of its extremal reads were part of a read (extremal

or not) in the second bundle b''. Then, there would exist a (contiguous) $r-k$-spaced subsequence of the genome represented by the first bundle beginning with an anchor entirely to the left of b'' and ending with an anchor entirely to the right of b''. Thus, any read of the second bundle containing b'' would contain at least one full anchor from such a sequence; and thus, by the definition of bundle, would be part of the first bundle.

By the same argument, we can easily prove that a final depth i bundle does not overlap with any temporary depth $j < i$ bundle, expect for (at most) the bases in their respective extremal reads. Were it not the case, there would be a read from depth j bundle containing a full depth i anchor from the final depth i bundle. □

Armed with Theorem 1, we now proceed to show that a sufficiently "dense" read set will allow the reconstruction of the full genome. In particular, we can prove the following:

Theorem 2. *Assume there is a sequence of reads $z_1, \ldots, z_m$, such that, for $1 \leq i < m$, the leftmost base of z_{i+1} is to the right of the leftmost base of z_i, but to the left of its $(k-1)^{th}$ rightmost base. Then our algorithm at some point produces a bundle containing all of $z_1, , \ldots, z_m$.*

In other words, if there is a sequence of reads, with each immediately to the right of the previous, but overlapping by at least a k-mer, then the entire portion of the genome spanning from the first to the last of those reads is fully reconstructed by our algorithm; and in particular, if the first read holds the first base of the genome and the last read the last base, the entire genome is reconstructed.

Proof. We begin with two simple observations. First, since every $k-$mer is an anchor at *some* depth, every read will eventually become part of some bundle. Second, since bundles never split, if two reads share the same final depth j bundle, they will also share the same final depth ℓ bundle for all $\ell < j$. Then, since z_i and z_{i+1} for $1 \leq i < m$ share at least one $k-$mer, they share a depth j anchor for some j anchor, and thus will become part of the same final depth ℓ bundle for all $\ell < j$. □

We now proceed to examine the time and space requirements of our algorithm. The first step in this direction is to introduce a simple model for our hardware.

This is crucial, because modern computing devices are not well-described by the "classic" RAM-model, where every operation and/or access to an elementary datum has cost 1. Instead, the memory system of any such device is usually formed by several layers of progressively larger size but also progressively higher access cost (in terms of both time and energy); with the ratio between sizes and access costs spanning several orders of magnitude. Furthermore, the size of the elementary data unit that can be accessed at different levels[1] can also vary considerably – from a few bytes at the processor register level, to a megabyte or more at the disk level.

Striking a balance between simplicity and accuracy, we shall assume that our machine is formed by 3 layers of storage. The first is the processor cache, of capacity C_p, whose elementary data unit is a single integer, and with access cost 1 – the same cost of carrying out an elementary operation such as an integer comparison or an arithmetic operation. The second is the (main) memory, of capacity $C_m >> C_p$, whose elementary data unit is the cache line of size $B_m >> 1$ integers, and with access cost A_m (with $\frac{A_m}{B_m} >> 1$). The third is the disk, of capacity $C_d >> C_m$, whose elementary data unit is the disk block of size $B_d >> B_m$, and with access cost A_d(with $\frac{A_d}{B_d} >> \frac{A_m}{B_m}$).

We assume that the input initially resides on disk; and similarly that the output must eventually be written to disk. Note that the input is c times larger than the output, and thus it dominates the total cost *if the output can be written sequentially* – a total cost that in this case is $\Theta(\frac{A_d}{B_d} \cdot cg)$.

The first crucial observation in the analysis of our algorithm is the following:

Lemma 1. *The space required by a bundle is within a constant factor of the space required by the sequence represented by the bundle.*

Proof. The proof is immediate noting that:

1. Each base appears at most once in the bundle outside of the anchors.

2. If we are using a balanced $h-$hash with $h = \Theta(k)$, the total size of the anchors of a bundle is on average within a constant factor of the size of the sequence represented by the bundle.

[1]With elementary data unit we mean the largest datum that can be accessed at a cost at most a factor 2 larger than accessing a single bit.

3. There are at most $O(1)$ pointers per anchor.

4. The size of a pointer can be kept to $O(\lg g))$ bits; while the size of a $k-$mer must be at least $\Theta(\lg(n))$ bits to avoid repeated $k-$mers. Thus, the size of a pointer is within $O(1)$ times that of a $k-$mer.

□

In practice (see the following Chapter) it turns out that the constant factor is relatively small – about a factor 10 (and it could be reduced further with more aggressive optimizations). This means the main memory of a typical PC is sufficient to assemble a human genome ($\approx 3 * 10^9$ bases, i.e. less than $1GB$). The crucial point is that the size of any bundle depends only on the size of the associated genomic subsequence, and is *independent* of the number of reads involved – higher coverage does not mean higher space occupation, since multiple reads are "squeezed" together. Note that, because of Theorem 1, one can effectively assemble the bundles of different, temporary depths in parallel, without worrying about interference. Thus, we have proved the following:

Theorem 3. *Disregarding the storage required to hold the input, the space required by our algorithm is $O(g)$ – i.e. proportional to the genome size, independently of the coverage or of the size of the $k-$mers.*

Note that, in contrast, "standard" De Bruijn algorithms require space that is $\Theta(kg)$ – since they must store, independently, each of the g $k-$mers of the genome. In practice, this means our algorithm uses an order of magnitude less space than the state of the art competition!

We now turn our attention to the total time required. To simplify the analysis, we make two (realistic) assumptions. First, we assume that the size of a $k-$mer is $O(1)$ times the size of an integer. This is realistic, in that $k-$mers should be chosen as small as possible while still avoiding $k-$mer repetitions – which, as pointed out in the proof of Lemma 1, is approximately the size of a machine word. Second, we assume that $r = O(B_m)$ – i.e. a read is relatively small compared to the size of a cache line. Considering that a typical processor cache line may be 128 or 256 bytes, while even with Sanger technology reads rarely exceed a few hundred bases, this second assumption is realistic, too.

We can then easily prove the following:

Theorem 4. *For a sufficiently large memory capacity $C_m = O(g)$ our algorithm performs $O(\frac{c \cdot g}{B_d})$ accesses to disk; $O(n)$ accesses to memory; and $O(n(r-k))$ accesses to the processor's cache.*

Proof. The condition on the main memory size is essentially that all the bundles should fit in main memory (see Theorem 4). If this is the case, the Theorem follows easily from the following observations:

1. Each read is read once from disk into main memory.
2. Since reads can be read sequentially, one can read B_d/r reads at a time, amortizing the disk access cost.
3. Similarly, the genome can be written sequentially in output, amortizing the disk access cost and making the output cost $\approx 1/c$ times the input.
4. The first time a read is accessed, its $r-k$ $k-$mers are computed and hashed; this requires a total of $O(n(r-k))$ accesses to the processor's cache.
5. The cost to "tie" a read to a bundle of appropriate depth is $O(1)$ accesses to main memory (since a read has, on average, $O(1)$ maximum depth anchors), for a total of $O(n)$ accesses.
6. On average, at each depth a fraction bounded away from 0 of all reads become non-extremal reads of final depth bundles, and can thus need be considered no further. Therefore the total number of accesses of the algorithm can be bound to within an $O(1)$ factor of the accesses at maximum depth.

□

It is interesting to compare the results from Theorem 4 with those of "standard" De Bruijn graph algorithms. The latter perform (asymptotically) the same number of accesses to disk (assuming the main memory is sufficiently large – namely $\Theta(k)$ times larger than in our case): basically in both cases the disk is accessed only to load the input and output the assembled genome. They also perform asymptotically the same number of processor accesses, i.e. $r-k$ per read (to extract the $k-$mers). But they perform $\Theta(n(r-k))$ accesses to main memory (asymptotically as many as to the processor cache) vs. our $\Theta(n)$. Thus,

if memory accesses are the bottleneck, our algorithm is faster by up to a factor $\Theta(r-k)$.

Another way of seeing it is that our algorithm concentrates the accesses in such a way as to maximize their locality (hence the title of this work), and thus, although performing the same number of accesses, all but a fraction $\frac{1}{r-k}$ are cache accesses, whereas those of "standard" De Bruijn graph algorithms are predominantly memory accesses.

3.4 Extensions

This section shows how our algorithm can be extended to deal with repeated $k-$mers, parallelized, and translated to colour space.

Repeated $k-$mers. Most genomes sport fairly long sequences that appear more than once. Thus, even if $k-$mers are sufficiently long to avoid "accidental" repeats of $k-$mer in two totally unrelated sequences, it is crucial to be able to deal with repeated $k-$mers. The fundamental issue in this case is that, if repeated $k-$mers are present, the genome no longer appears as a linear sequence of bases, but instead as an oriented "graph of bases", with loops indicating a repeated sequence.

This can lead to ambiguities: for example, the sequence $< s_0, s, s_1, s, s_2, s, s_3 >$ cannot be distinguished from the sequence $< s_0, s, s_2, s, s_1, s, s_3 >$ if the subsequences s_1, s_2 and s are all sufficiently long compared to the size r of a read. Sometimes mate pair information can be used to solve the ambiguity, but not always; in many cases, one simply has to accept that the output will not be a linear sequence of bases, but instead a labeled, oriented graph. Nodes are labeled with linear sequences of bases, and a node v has an arc to another node u if the sequence associated to v is immediately followed, in the genome, by the sequence associated to u. Obviously nodes with a single incoming arc or a single outgoing arc can be collapsed.

With this caveat, our algorithm can work in an almost identical fashion. The crucial difference is that, in case of a *divergence* (the same $k-$mer appearing followed by two different bases in two different reads) one must chose one of the two continuations as the "main" one, and instead have the other continuation become the prefix of a new linear sequence. Symmetrically, in case of a *convergence* (the

same $k-$mer appearing preceded by two different bases in two different reads), one must choose one of the two converging sequences to be the "main" one that will continue into the $k-$mer, and have the other become the suffix of a new linear sequence. This adds only some minor bookkeeping to our algorithm, and does not asymptotically change its complexity.

Parallelization. Our algorithm is particularly easy to parallelize, though care must be taken to avoid having the parallelization process destroy our carefully built locality: in other words, it often makes no sense to perform two operations on the same datum on two different processors, because the time required to move the datum to the second processor far outweighs the advantage of carrying out the two operations in parallel. Thus, different parts of the computation can be parallelized at different points in the memory hierarchy.

To parallelize the computation at the cache level, one can exploit the simultaneous multi-processing and multi-threading capabilities of the individual processor core – in a nutshell,the ability of modern processors to carry out up to w operations in parallel (today typical values of w fall into the $2-16$ range, although the higher end of the range is growing) as long there are no data or control dependencies. This is ideal for carrying out the computation of the different $k-$mers of a read, and can yield up to a $min((r-k), w)$ speedup if the cache accesses are the bottleneck.

To parallelize the computation at the memory level, one can process reads in parallel, although care must be taken to avoid conflicts. Also, if access to the memory is the bottleneck, this parallelization must take place across different motherboards – adding the load of multiple cores or processors to the same memory controller will, if anything, deteriorate performance.

Finally, should the I/O be the bottleneck, one can certainly parallelize it across an array of disks.

Colours vs. bases. Some sequencing technologies (notably SOLiD of Applied Biosystems) produce sequences not of bases, but of "colours" – in a nutshell sequences with the same information content, but where each bit depends on multiple bases (see Chapter 1). While this tends to provide greater resilience to errors, it is also creates numerous problems when errors do appear, because the effects of a single error can propagate to large distances.

It should be noted, however, that *in the absence of errors, one can operate*

in colour space exactly as in base space. And since our approach decouples error correction/detection from the actual assembly, it can be automatically translated to colour space with virtually no modifications.

Chapter 4

Assembler implementation and experimental results

In order to experimentally prove the effectiveness of our assembly algorithm, we actually implemented it. This Chapter describes the design and implementation of our SyntAssemblerII software and evaluates its performance both in computational time and resource consumption. Section 4.1 describes the guidelines we followed during the development of our software and provides an overview of its architecture. Section 4.2 illustrates the experimental setup which brought to the results presented in Section 4.3. In these last two Sections we provide context to the performance of our assembler by performing a direct comparison with Velvet, a widely employed de novo assembler.

4.1 Assembler implementation

In this Section, we describe the tools we adopted during the development of our assembler, briefly motivating our choices (Subsection 4.1.1). Then, we provide an overview of the SyntAssemblerII architecture (Subsection 4.1.2).

4.1.1 Development tools

We developed our assembler in C++, since C++ provides a combination of both high-level and low-level language features. In particular, it offers both the powerful semantics typical of object-oriented languages, paired with very low level data manipulation. Furthermore, C++ provides the developer with direct memory

management; given the typical memory footprint of *de novo* assembly computation problems, this clearly turns out to be a highly desirable feature.

During development, we followed some Extreme Programming methodologies such as *Test Driven Development* (TDD) and *Keep It Simple Stupid!* (KISS). Test Driven Development (TDD) requires the developer to set up a suite of unit tests. The purpose of these tests is to check the correct execution of (almost) every single program method, *before* writing the actual code. More precisely, the development process in TDD relies on the iteration of a very short development cycle. Every iteration consists of two steps. First, the developer identifies a desired improvement or a new function, and designs an automated test case for it (obviously, the test case will fail at this stage). Then, the code to pass that test is produced and finally refactored in order to match acceptable standards. Every iteration enriches the software with novel functionalities.

Although TDD could seem an excessively time consuming methodology, it is really effective in preventing bugs: due to the isolation of single test cases, one can immediately identify the portion(s) of code no longer work after a modification of the existing code base. And since TDD requires developers to think of the software in terms of small units, with each small unit written and tested independently, the resulting code is highly modularized, flexible and extensible .

The philosophy of TDD naturally leads to the so called Keep It Simple Stupid (KISS) software design methodology. KISS practice focuses on making easily understandable (ideally self-explanatory) objects and methods. In general, this speeds up the revision of already written portions of code and their integration with new ones. KISS principle states that most systems work better if they are kept simple – simplicity should be a key goal in design and unnecessary complexity should be avoided at all costs.

Our choice for a development environment was Eclipse [13] with its CDT plugin for C/C++ development. The unit tests suite is powered by CUTE [14] (C++ Unit Testing Easier). CUTE features an Eclipse plugin; this noticeably speeds up the time required for writing a test case and subsequently executing the suite. When simple unit testing was not enough – e.g., in memory leak detection and correction – we used Valgrind [9].

Our assembler is made up of approximatively 6000 lines of code – plus approximately 10000 lines of code for the unit test suite. Its development required

approximatively 400 hours (270 spent writing code and 130 hours spent debugging).

4.1.2 Architecture overview

The methodologies we adopted led to a simple and effective software design. In particular, there are four main classes:

- `ContigInterpreter`: objects of this class handle reads and, more in general, genomic subsequences. They provide immediate conversions from the "1-byte-per-symbol" representation to the "2-bit-per-symbol" one and vice-versa, by using a specifically designed internal data compressor. Since the possible colours (or bases) are only four, they can be unambiguously represented using only two bits. Further, `ContigInterpreter` objects embed methods managing the "fusion" and comparison operations between genome (sub)sequences. "Fusion" is obtained by merging two (or more) subsequences into a single one which covers their whole span.

- `AnchorPoint`: the goal of these objects is to provide a quick reference to an *anchor* (a special $k-$mer selected by the means of a hash, as described in Chapter 3) and to the reads anchored to it. It provides methods to tie a subsequence to this *anchor*, recording little additional information in order to simplify the selection of those subsequences which extend the anchor's support (while the others can be discarded).

- `AnchorBundle`: this class provides wrap objects. They translate a set of `AnchorPoint` instances into *bundle* abstractions and take care of linking the various `AnchorPoint` with each other. Objects of this type also provide utility methods able to join more *bundles* into a single one. Finally, this class includes methods to retrieve the *bundle* support (using the "fusion" methods of `ContigInterpreter`).

- `BundleManager`: objects of this type track all bundles (i.e., all the `AnchorBundle` objects) and anchors (i.e., all `AnchorPoint` objects) encountered during the assembly procedure. They handle the processing of genome subsequences, by attaching them to the correct `AnchorPoint(s)` and joining them into a single `AnchorBundle` when needed.

Our assembler instantiates objects from these classes in order to execute a simple procedure. After reading the input data from the disk, it instantiates a `BundleManager` object and uses it to handle the first (deepest) anchoring depth. Then, it retrieves from this manager the support for the anchors at this depth. A new `BundleManager` is instantiated for the input that is not anchored at this depth, and is used to carry out the computation at the next depth. This procedure goes on until all depths are covered and the final depth 0 bundles are written to the output.

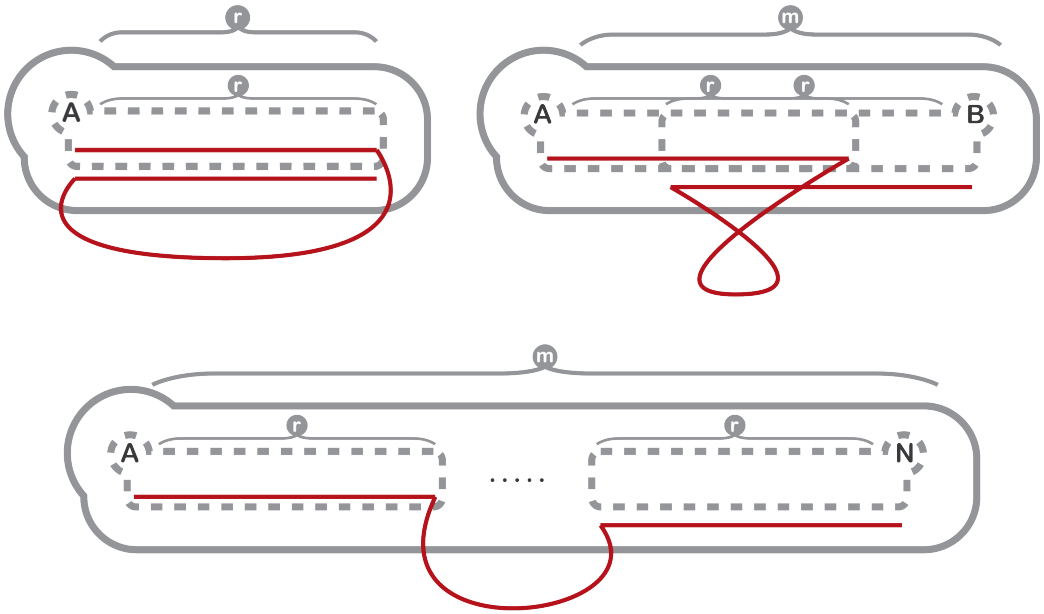

Figure 4.1: The `ContigInterpreter` class provides a smart representation of genomic subsequences, avoiding the need of storing unnecessary information.

We emphasize how a key aspect of our software design is the abstract concept of contig, implemented through the `ContigInterpreter` class (see Figure 4.1). Recall the theory developed in Chapter 3: while at each depth the continuous sequences of bases or colours (which we call *contigs* are expected to grow, the overlapping zones are localized only in the extremal reads. Thus, we do not have hold information about the "middle" portion of a contig, as the software "ascends" to lesser depths. But this information is not lost: it can be retrieved

by considering the original contigs which where merged into the new contig.

4.2 Experimental setup

The first set of experiments involved testing the quality of different candidates for $h-$hashes. We evaluated three different fast hash functions over the set of all $k-$mers in the E. Coli genome, one $k-$mer for each distinct starting position in the genome. The evaluation was in terms of both speed and key distribution.

We then performed a set of experiments on a synthetic colour-space genome of 10 million bases. Its error-free reads contain no repetitions longer than 25 colours. From these reads, we extracted a sequence of subsets. One of these subsets was made of 1250000 "consecutive", partially overlapping reads, in order to grant the possibility of a complete reconstruction of the original genome. The other subsets were constituted by 2, 5, 6, 15, 25, 50 and 75 millions of *randomly positioned* reads. Despite the random positioning, due to their high coverage, the last three subsets contain enough information to reconstruct the original genome.

Another set of experiments was performed on the same synthetic colour-space genome to compare the number and length of contigs obtained from our software and our competitors. In this set of tests we also measured the number of bases from the original genome left uncovered by the set of contigs that the different software produced in output. These experiments were performed on sets of randomly positioned reads resulting in coverage respectively equal to $5\times$, $10\times$, $15\times$, $20\times$, $25\times$, $30\times$ and $35\times$.

All tests were performed on a commodity Personal Computer equipped with an Intel Core i7-2720QM CPU, with clock frequency of 2.20 GHz, an amount of 8 Giga Bytes of DDR III 1.333MHz RAM and a S.ATA II Hard Disk with capacity of 320 Giga Bytes, disk spin speed of 7200RPM and a 16 Mega Bytes buffer.

Initially, on the synthetic genome we run our assembler, Velvet [47] and ABySS [43]. Unfortunately, we could not gather a significant amount of experimental results for ABySS: it succeeded in the assembly task only in one of the presented cases (the small set of sequential reads). In this case, ABySS correctly produced one contig of 9999991 bases (only 9 from the original genome are missing) in 102 seconds. For this reason, in the following Section we consider only our assembler and Velvet.

4.3 Experimental results

The first and the second hash functions we tested were based on Knuth's well known multiplicative hash [17], that effectively multiplies each $k-$mer (read as an unsigned integer) by $\frac{\sqrt{5}-1}{2}$, takes the fractionary part and multiplies that by h, rounding down. The first function set h to 32; the second to 31. Using a power of 2 instead of a prime for h yielded, as expected, a lower quality hash (see Figures 4.3 and 4.3) but a faster execution time (5 seconds vs. 8 to execute 10^10 hashes). The third function we tested involved multiplying a $k-$mer by a smallish prime – in our case 2043, small enough that the result still fit within a 64 bit word but much larger than k – and taking the result modulus 31. This third hash had a quality comparable to the better of the two hashes based on Knuth's method (see Figures 4.3) and was slightly faster (7 seconds to execute 10^10 hashes).

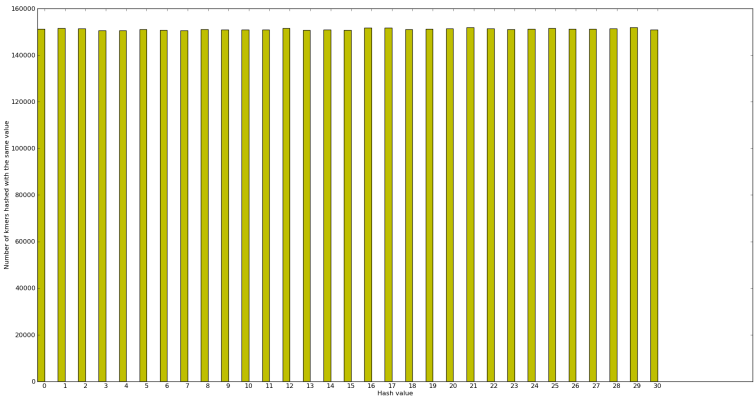

Figure 4.2: Distribution of E. Coli's $k-$mers according to Knuth's multiplicative hash, into 31 sets.

In terms of performance of SyntassemblerII, the experimental results support the theoretical analysis carried out in Chapter 3. Memory usage is a crucial parameter when dealing with *de novo* assembly. Figure 4.5 shows how the larger is the size of input set, the larger are the memory requirements of Velvet. Instead, our software maintains a constant occupation, due to the proportionality to the size of genome (which obviously is the same for all the tests). In particular, we can focus on the red line; the green one incorporates a 500 Mega Bytes Input/Output

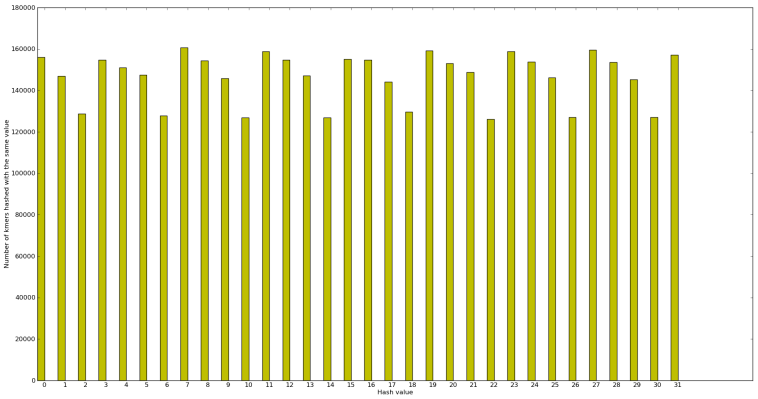

Figure 4.3: Distribution of E. Coli's $k-$mers according to Knuth's multiplicative hash, into 32 sets.

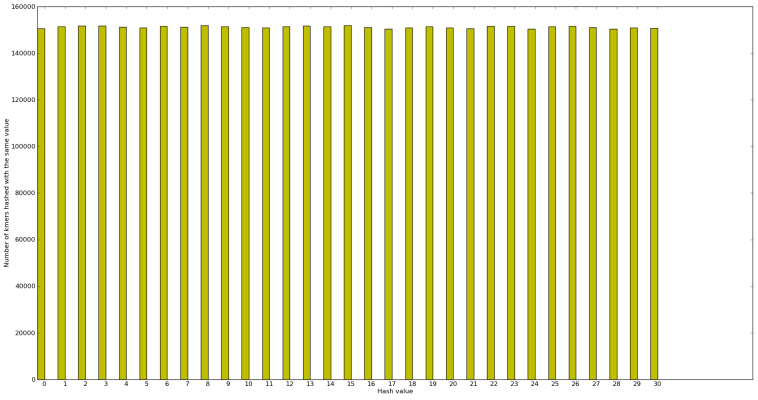

Figure 4.4: Distribution of E. Coli's $k-$mers into 31 sets, by multiplying each $k-mer$ by 2043 and taking the result modulus 31.

buffer to improve disk access performance – by introducing some optimizations, this buffer could be reduced or even avoided.

One of the major strengths of our approach resides in a clever data representation and in the subsequent input size reduction. We store only data bringing actual, useful information and discard all unnecessary and repeated reads (e.g., reads coming from oversampled regions of the the original genome). In Figure 4.6, the blue line shows how the input size dramatically decreases during our assem-

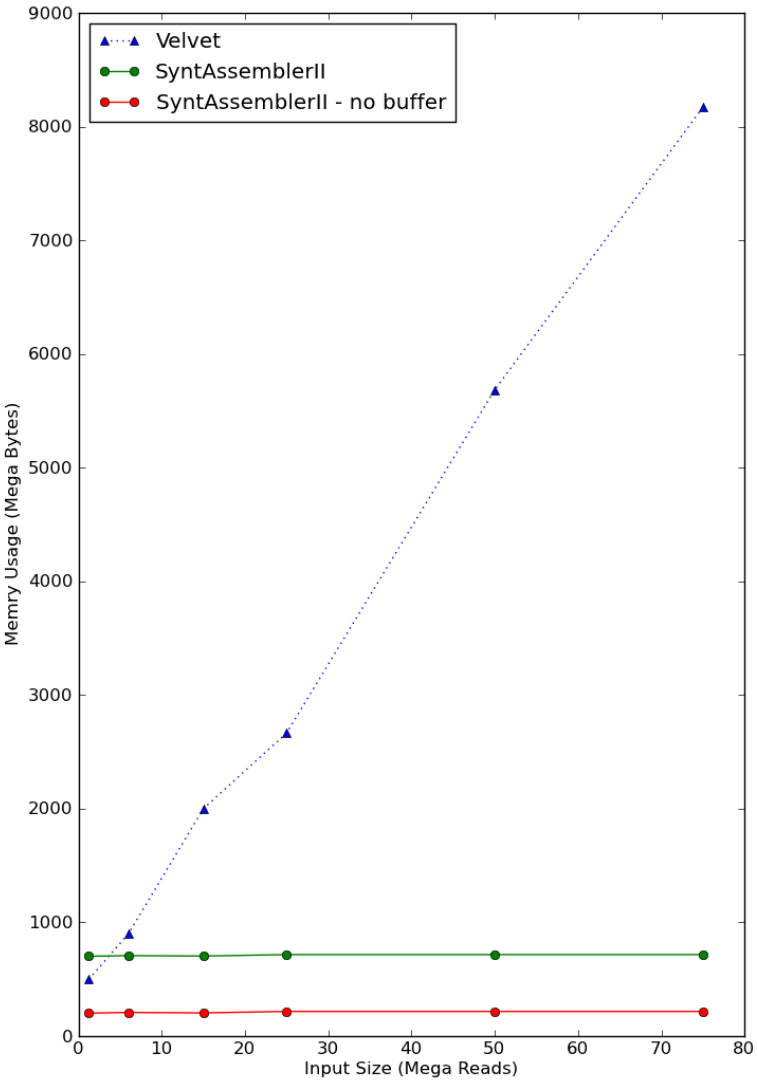

Figure 4.5: Comparison between SyntAssemblerII (with I/O buffering – higher solid line – and without I/O buffering – lower solid line) and Velvet (dotted line) memory usage peaks in function of the input size.

bler execution, depth level by depth level. This is one of the keys leading to the performance shown in Figure 4.8 and Figure 4.5. As a side effect, it affects the disk occupation after a run is performed (see Figure 4.7).

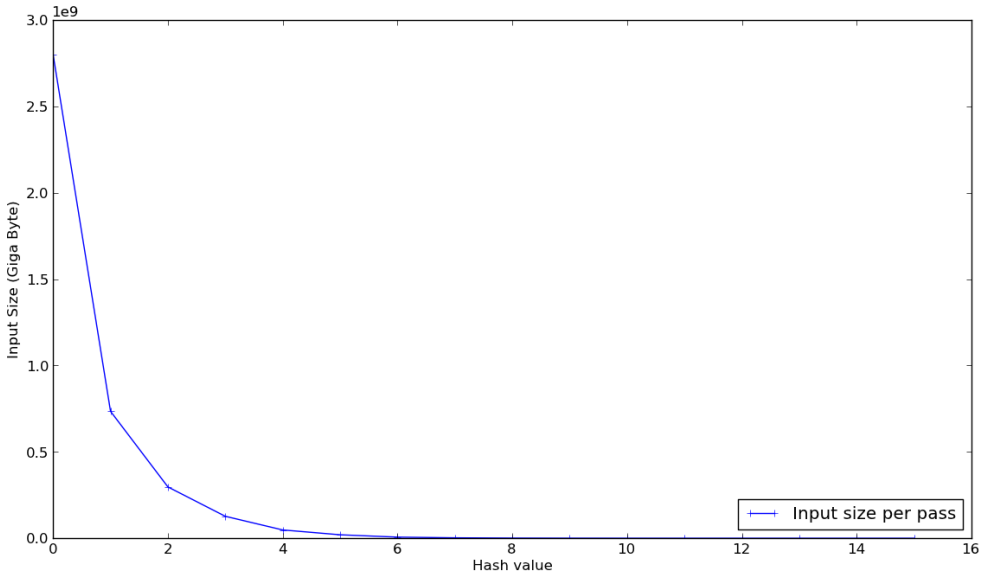

Figure 4.6: Shrink of input size for every depth level during execution. The initial size is 2.9GBytes, corresponding to 50 million reads.

Since our software carefully manages memory memory and makes use of a clever data representation, during the assembly process it has to deal with substantially less data – which in turn reduces its running time. Figure 4.8 compares our software's running time with that of Velvet – the sharp increase in Velvet's at 75 million reads is probably due to its outstripping the main memory capacity and having to resort to frequent accesses to disk. In a nutshell, our software clearly outshines Velvet in terms of raw performance: it exhibits shorter execution times and smaller memory and disk footprints – thus running faster even when on cheaper hardware.

Last but not least, when dealing with an input set with small coverage, our software provides a surprisingly higher quality output than Velvet. As the dataset coverage increases, the number of genome subsequences produced as output by our software drops more rapidly than that produced by Velvet (indicating a more rapid convergence to a single, unfragmented genome) – see Figure 4.9. Similarly, the size of the longest assembled subsequences grow more rapidly – as can be

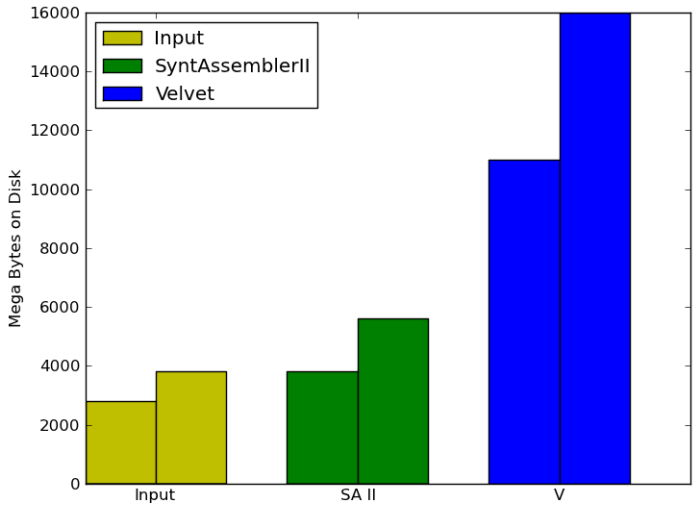

Figure 4.7: Comparison between the memory footprint of two sets of 50 and 75 million randomly positioned reads: initial input size, memory occupation after one pass by our software and after one pass by Velvet.

observed in Figure 4.10. And while Velvet's output always leaves many bases uncovered, our program reaches a 100% coverage (even if scattered among the contigs) as soon as the input allows this (see Figure 4.11).

Let us consider a single example. On a 6 million read input, Velvet completed execution providing 100 contigs. Among these 100 contigs, the longest one was about 498938 bases long and the total number of bases from all the contigs in output was 9999682. This means that 318 bases are missing in the final output. Our software completed the task by giving only 2 contigs: the first 9274904 bases long and the second 725115 bases long. The total number of bases was 10000019 and simple inspection showed that the two contigs overlapped exactly on 19 bases. That is, no bases from the original genome are missing and a complete reconstruction is possible with little additional work.

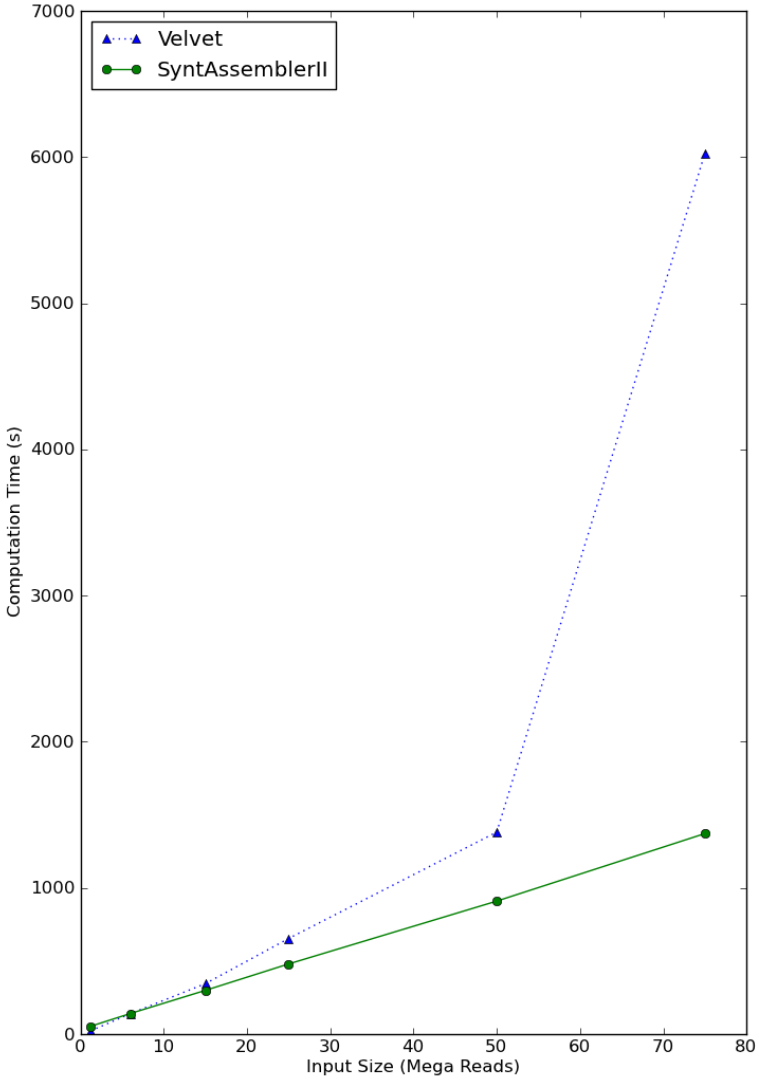

Figure 4.8: Comparison between SyntAssemblerII (solid line) and Velvet (dotted line) computation times as a function of the input size.

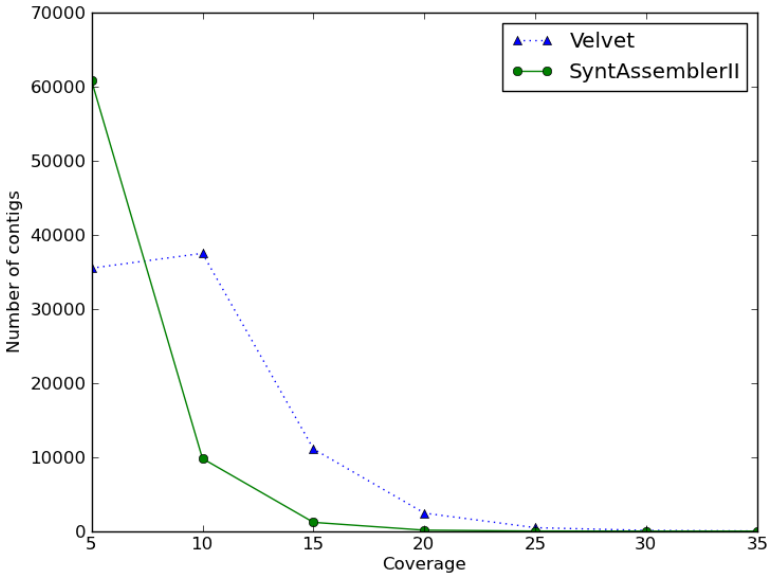

Figure 4.9: The number of contigs obtained as output from some execution of our software and Velvet.

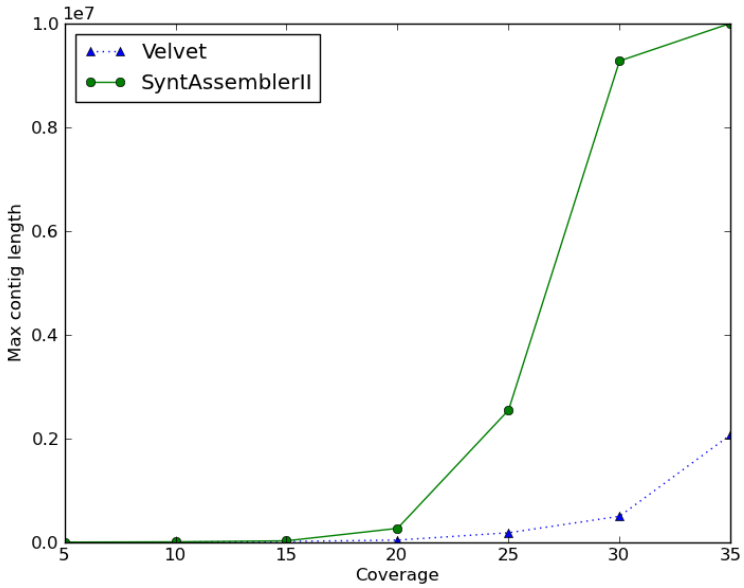

Figure 4.10: Size of the longest contig obtained as output from some execution of our software and Velvet.

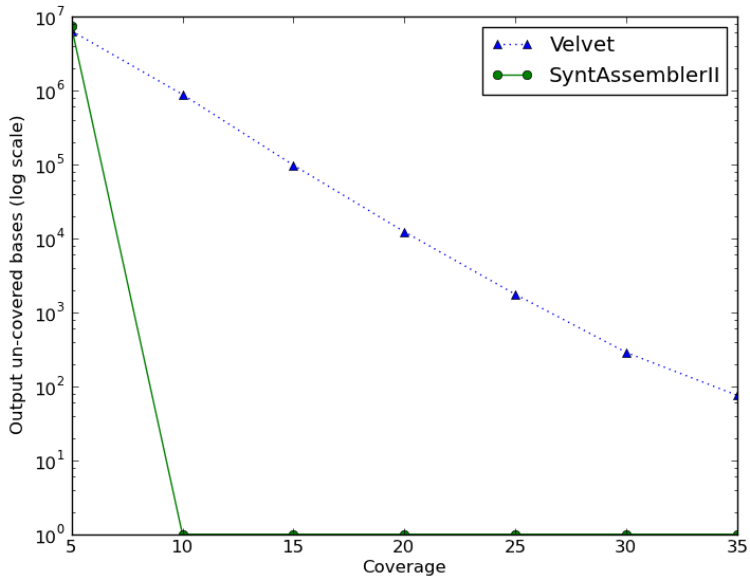

Figure 4.11: The number of bases not covered by the output of assembly process (in logarithmic scale).

Chapter 5

Conclusions and future work

This thesis presented a two-pronged approach to more efficient de novo DNA assembly.

The first, preliminary contribution is an improved filter for noisy reads (which technically can be of use even outside de novo assembly – e.g. in mapping). A simple filter based on a probabilistic model of a read's errors already outperforms Sasson's filter (the "industry standard") on the E.Coli genome. But the true power of our approach emerges when exploiting (unlike Sasson's filter) cross-read information with medium to high coverage: at $50\times$ coverage our filter has almost perfect accuracy in identifying error-free reads. In this regard, it is important to observe that such a level of coverage is the minimum necessary, anyway, to provide reasonable guarantees that every base in genome is covered by at least one error-free read. A crucial advantage of such a high quality filter is that it removes many of the complications that dealing simultaneously with errors and colour-based reads entails: without errors, assembly in base and colour space essentially coincide.

The second contribution of this thesis is an algorithm for de novo DNA assembly that vastly outperforms state-of-the-art algorithms based on De Bruijn graphs. From a theoretical point of view, the memory used by our algorithm is a factor $\Theta(k)$ less (where k is the $k-$mer size – at least $15 - 25$ to allow correct reconstruction of a genome). Furthermore, while our algorithm performs asymptotically the same number of operations as the competition, it *m*aximizes the locality of those accesses: the number of non-cache accesses is reduced, again, by a factor $\Theta(k)$ (meaning that, if the ratio between the access cost of cache and main memory and cache is ϵ, our algorithm runs $min(k, 1/\epsilon)$ times faster).

The advantages of our algorithm are not, however, purely theoretical. A sim-

ple implementation in C++ of our algorithm that required only a few hundred hours of coding vastly outperforms a state-of-the-art assembly software like Velvet, that has been under development for several years. Our assembler runs several times faster, and sports a memory footprint several times smaller (and thus requires considerably cheaper hardware to run). Also, it seems that the "cleaner" approach of our algorithm provides better quality output with a higher coverage of the genome.

There are many directions in which this work may be extended. In terms of filtering, the greatest challenge is probably to move from error detection to error correction. In our tests, less than a read out of 5 was completely error free; but those reads with 1 or 2 errors can still provide valuable information. In this regard, the cross-read approach of our extended probabilistic filter appears particularly promising.

In terms of assembly, while our algorithm correctly deals with repeated $k-$mers, support for them in the actual software is still at an early stage of development. Our software also does yet not incorporate information from mate pairs to assist in the scaffolding of low coverage read sets – though adding this functionality seems to pose relatively few challenges and, in fact, would seem extremely easy to develop in our "anchor and bundle" framework. Yet another interesting line of research would be to experiment with "dynamic" $k-$mers: in a nutshell, as the number of distinct subsequences grows at every iteration, the minimum length of $k-$mers to avoid mistakenly "stitching" together unrelated subsequences can decrease – allowing one to reduce fragmentation in the case of low coverage read sets. And there are still many optimizations that can be incorporated in the software to further reduce by a factor $2-4$ at least both the running time and the memory footprint.

Finally, we would stress that many of the techniques and results in this thesis should be relatively easy to "port" to the investigation of other problems in computational genomics, such as mapping onto a reference genome or transcriptome analysis.

Bibliography

[1] P.N. Ariyaratne and W.K. Sung. Pe-assembler: de novo assembler using short paired-end reads. *Bioinformatics*, 27(2):167–174, 2011.

[2] S. Batzoglou, D.B. Jaffe, K. Stanley, J. Butler, S. Gnerre, E. Mauceli, B. Berger, J.P. Mesirov, and E.S. Lande. Arachne: a whole-genome shotgun assembler. *Genome Research*, (12):177–189, 2002.

[3] S. Bennet. Solexa ltd. *Pharmacogenomics*, 5(4):433-438, 2004.

[4] 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 research*, 18(5):810–820, 2008.

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

[6] M. J. Chaisson, P. A. Pevzner, and H. Tang. Fragment assembly with short reads. *Bioinformatics*, 20:2067–2074, 2004.

[7] M.J. Chaisson and P.A. Pevzner. Short read fragment assembly of bacterial genomes. *Genome Res.*, 18(2):324–330, 2008.

[8] International Human Genome Sequencing Consortium. Initial sequencing and analysis of the human genome. *Nature*, 409:860–921, 2011.

[9] Valgrind Developers. Valgrind. `http://valgrind.org/`, 2011. [Online; accessed 10-December-2011].

[10] B. Ewing and P. Green. Base-calling of automated sequencer traces using phred. ii. error probabilities. *Genome Res.*, 8(3):186–194, 1998.

[11] B. Ewing, L. Hillier, M.C. Wendl, and P. Green. Base-calling of automated sequencer traces using phred. i. accuracy assessment. *Genome Res.*, 8(3):175–185, 1998.

[12] P. Flicek and E. Birney. Sense from sequence reads: methods for alignment and assembly. *Nature Methods*, 6, 2009.

[13] The Eclipse Foundation. Eclipse. `http://www.eclipse.org/`, 2011. [Online; accessed 10-December-2011].

[14] HSR Hochschule fur Technik Rapperswil Institut fur Software. CUTE: C++ Unit Testing Easier. `http://cute-test.com/`, 2011. [Online; accessed 10-December-2011].

[15] O. Harismendy, C. Pauline, R. Strausberg, X. Wang, T. Stockwell, K. Beeson, N. Schork, S. Murray, E. Topol, S. Levy, and K. Frazer. Evaluation of next generation sequencing platforms for population targeted sequencing studies. *Genome Biology*, 10(3), 2009.

[16] X. Huang and S.P. Yang. Generating a genome assembly with pcap. *Current Protocols in Bioinformatics*, (11):177–189, 2005.

[17] Donald E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[18] B. Langmead, C. Trapnell, M. Pop, and S. Salzberg. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biol.*, 10(3), 2009.

[19] H. Li and R. Durbin. Fast and accurate long-read alignment with burrows-wheeler transform. *Bioinformatics (Oxford, England)*, 26(5), 2010.

[20] H. Li and N. Homer. A survey of sequence alignment algorithms for next–generation sequencing. *Brief Fioinfogm*, 11(5):473–483, 2010.

[21] 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):265–272, 2009.

[22] Y. Lin, J. Li, H. Shen, L. Zhang, C. Papasian, and H. Deng. Comparative studies of de novo assembly tools for next-generation sequencing technologies. *Bioinformatics (Oxford, England)*, 27(15):2031–2037, 2011.

[23] I. Maccallum, D. Przybylski, S. Gnerre, J. Burton, I. Shlyakhter, A. Gnirke, J. Malek, K. McKernan, S. Ranade, T.P. Shea, L. Williams, S. Young, C. Nusbaum, and D.B. Jaffe. Allpaths 2: small genomes assembled accurately and with high continuity from short paired reads. *Genome biology*, 10(10):R103+, 2009.

[24] E.R. Mardis. The impact of next-generation sequencing technology on genetics. *Trends Genet.*, 24(3):133–141, 2008.

[25] M. Margulies and M. Egholm. Genome sequencing in open microfabricated high density picoliter reactors. *Nature*, 437(7054):376-380, 2005.

[26] A.M. Maxam and W. Gilbert. A new method for sequencing dna. *PNAS*, 74(2):560–564, 1977.

[27] M.L. Metzker. Emerging technologies in dna sequencing. *Genome Res.*, 15:1767–1776, 2005.

[28] M.L. Metzker. Sequencing technologies – the next generation. *Nat. Rev. Genet.*, 11(1):31–46, 2010.

[29] J. R. Miller, S. Koren, and G. Sutton. Assembly algorithms for next-generation sequencing data. *Genomics*, 95(6):315–327, 2010.

[30] 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*, (15), 2008.

[31] E.W. Myers, G.G. Sutton, A.L. Delcher, I.M. Dew, D.P. Fasulo, M.J. Flanigan, S.A. Kravitz, C.M. Mobarry, K.H. Reinert, K.A. Remington, E.L. Anson, R.A. Bolanos, H.H. Chou, C.M. Jordan, A.L. Halpern, S. Lonardi, E.M.

Beasley, R.C. Brandon, L. Chen, P.J. Dunn, Z. Lai, Liang Y., D.R. Nusskern, M. Zhan, Q. Zhang, X. Zheng, G.N. Rubin, M.D. Adams, and J.C. Venter. A whole-genome assembly of drosophila. *Science*, 287:2196–2204, 2000.

[32] V. Pandey, R.C. Nutter, and E. Prediger. *Applied Biosystems SOLiD System: Ligation-Based Sequencing*, Wiley, 2008.

[33] K. Paszkiewicz and D.J. Studholme. De novo assembly of short sequence reads. *Briefings in Bioinformatics*, 11(5):457–472, 2010.

[34] P.A. Pevzner, H. Tang, and G. Tesler. De novo repeat classification and fragment assembly. *Genome Res.*, 14(9):1786–1796, 2004.

[35] P.A. Pevzner, H. Tang, and M.S. Waterman. An eulerian path approach to dna fragment assembly. *Proceedings of the National Academy of Sciences*, 98(17):9748–9753, 2001.

[36] M. Pop. Genome assembly reborn: recent computational challenges. *Briefings in bioinformatics*, 10(4):354–366, 2009.

[37] F. Sanger and A. R. Coulson. A rapid method for determining sequences in dna by primed synthesis with dna polymerase. *Journal of Molecular Biology*, 95(3):447–448, 1975.

[38] F. Sanger, S. Nicklen, and A.R. Coulson. Dna sequencing with chain-terminating inhibitors. *PNAS*, 74(12):5463–5467, 1977.

[39] A. Sasson and T.P. Michael. Filtering error from solid output. *Bioinformatics (Oxford, England)*, 26(6):849–850, March 2010.

[40] J. Shendure and J. Hanlee. Next-generation dna sequencing. *Nature Biotechnology*, 26:1135–1145, 2008.

[41] J. Shendure, R.D. Mitra, C. Varma, and G.M. Church. Advanced sequencing technologies: methods and goals. *Nat Rev Genet.*, 5(5):335–344, 2004.

[42] J. Shendure, G.J. Porreca, N.B. Reppas, X. Lin, J.P. McCutcheon, A.M. Rosenbaum, M.D. Wang, K. Zhang, R.D. Mitra, and G.M. Church. Accurate multiplex polony sequencing of an evolved bacterial genome. *Science*, 309(5741):1728–1732, 2005.

[43] J. T. Simpson, K. Wong, S.D. Jackman, J.E. Schein, S.J. Jones, and I. Birol. Abyss: A parallel assembler for short read sequence data. *Genome Research*, 19(6):1117–1123, 2009.

[44] A. Valouev, J. Ichikawa, T. Tonthat, J. Stuart, S. Ranade, H. Peckham, K. Zeng, J.A. Malek, G. Costa, K. McKernan, A. Sidow, A. Fire, and S.M. Johnson. A high-resolution, nucleosome position map of c. elegans reveals a lack of universal sequence-dictated positioning. *Genome Res.*, 18:1051–1063, 2008.

[45] N. Whiteford, N. Haslam, G. Weber, A. Prugel-Bennett, J.W. Essex, P.L. Roach, M. Bradley, and C. Neylon. An analysis of the feasibility of short read sequencing. *Nucleic Acids Res.*, 33, 2005.

[46] Business Wire. Helicos biosciences enters molecular diagnostics collaboration with renowned research center to sequence cancer-associated genes. *Genetic Engineering and Biotechnology News*, 2008.

[47] D. Zerbino and E. Birney. Velvet: Algorithms for de novo short read assembly using de bruijn graphs. *Genome Research*, 18(5):821–829, 2008.

List of Figures