THÈSE / ENS CACHAN - BRETAGNE

sous le sceau de l'Université européenne de Bretagne

pour obtenir le titre de

DOCTEUR DE L'ÉCOLE NORMALE SUPÉRIEURE DE CACHAN

Mention : Informatique

École doctorale MATISSE

présentée par

Rayan Chikhi

Préparée à l'Unité Mixte de Recherche 6074
Institut de recherche en informatique
et systèmes aléatoires

Computational Methods for de novo Assembly of Next-Generation Genome Sequencing Data

Thèse soutenue le 2 juillet 2012
devant le jury composé de :

Éric RIVALS,
DR, LIRMM / *rapporteur*

Sante GNERRE,
Group leader, Broad Institute of MIT and Harvard / *rapporteur*

Marie-France SAGOT,
DR, Inria Grenoble et Université de Lyon 1 / *examinateur*

Bertil SCHMIDT,
Professeur, Johannes Gutenberg-Universität Mainz / *examinateur*

Olivier JAILLON,
Chercheur, Genoscope-CNS (CEA) / *examinateur*

Dominique LAVENIER,
Professeur ENS Cachan - Bretagne et DR CNRS/ *directeur de thèse*

Résumé

Dans cette thèse, nous présentons des méthodes de calcul (modèles théoriques et algorithmiques) pour effectuer la reconstruction de séquences d'ADN. Il s'agit de l'assemblage de novo de génome à partir de lectures (courte séquences ADN) produites par des séquenceurs à haut débit. Ce problème est difficile, aussi bien en théorie qu'en pratique. Du point de vue théorique, les génomes sont structurellement complexes. Chaque instance d'assemblage de novo doit faire face à des ambiguïtés de reconstruction. Les lectures peuvent conduire à un nombre exponentiel de reconstructions possibles, une seule étant correcte. Comme il est impossible de déterminer laquelle, une approximation fragmentée du génome est retournée. Du point de vue pratique, les séquenceurs produisent un énorme volume de lectures, avec une redondance élevée. Une puissance de calcul importante est nécessaire pour traiter ces lectures. Le séquençage ADN évolue désormais vers des génomes et méta-génomes de plus en plus grands. Ceci renforce la nécessité de méthodes efficaces pour l'assemblage de novo.

Cette thèse présente de nouvelles contributions en informatique autour de l'assemblage de génomes. Ces contributions visent à incorporer plus d'information pour améliorer la qualité des résultats, et à traiter efficacement les données de séquençage afin de réduire la complexité du calcul. Plus précisément, nous proposons un nouvel algorithme pour quantifier la couverture maximale d'un génome atteignable par le séquençage, et nous appliquons cet algorithme à plusieurs génomes modèles. Nous formulons un ensemble de problèmes informatiques pour incorporer l'information des lectures pairées dans l'assemblage, et nous étudions leur complexité. Cette thèse introduit la notion d'assemblage localisé, qui consiste à construire et parcourir un graphe d'assemblage partiel. Pour économiser l'utilisation de la mémoire, nous utilisons des structures de données optimisées spécifiquement pour la tâche d'assemblage. Ces notions sont implémentées dans un nouvel assembleur de novo, Monument. Enfin, le dernier chapitre de cette thèse est consacré à des concepts d'assemblage dépassant l'assemblage de novo classique.

Abstract

In this thesis, we discuss computational methods (theoretical models and algorithms) to perform the reconstruction (de novo assembly) of DNA sequences produced by high-throughput sequencers. This problem is challenging, both theoretically and practically. The theoretical difficulty arises from the complex structure of genomes. The assembly process has to deal with reconstruction ambiguities. The output of sequencing predicts up to an exponential number of reconstructions, yet only one is correct. To deal with this problem, only a fragmented approximation of the genome is returned. The practical difficulty stems from the huge volume of data produced by sequencers, with high redundancy. Significant computing power is required to process it.
As larger genomes and meta-genomes are being sequenced, the need for efficient computational methods for de novo assembly is increasing rapidly.

This thesis introduces novel contributions to genome assembly, both in terms of incorporating more information to improve the quality of results, and efficiently processing data to reduce the computation complexity. Specifically, we propose a novel algorithm to quantify the maximum theoretical genome coverage achievable by sequencing data (paired reads), and apply this algorithm to several model genomes. We formulate a set of computational problems that take into account pairing information in assembly, and study their complexity. Then, two novel concepts that cover practical aspects of assembly are proposed: localized assembly and memory-efficient reads indexing. Localized assembly consists in constructing and traversing a partial assembly graph. These ingredients are implemented in a complete de novo assembly software package, the Monument assembler. Monument is compared with other state of the art assembly methods. Finally, we conclude with a series of smaller projects, exploring concepts beyond classical de novo assembly.

N° d'ordre :

École normale supérieure de Cachan - Antenne de Bretagne

Campus de Ker Lann - Avenue Robert Schuman - 35170 BRUZ

Tél : +33(0)2 99 05 93 00 - Fax : +33(0)2 99 05 93 29

Acknowledgments

My advisor, Dominique Lavenier, for immensely helpful guidance; Eric Rivals, Guillaume Rizk and Omar Chikhi for proof-reading the manuscript; my colleagues, for fruitful collaborations; Dorine and my family.

Contents

Chapter 1

Introduction

1.1 Introduction

DNA, sequence, genome, and sequencing

From a computational point of view, DNA sequences are long strings made of four different letters ($\{\mathtt{A}, \mathtt{C}, \mathtt{T}, \mathtt{G}\}$). In contrast, from a biological standpoint, DNA is a large molecule composed of repeated units (nucleotides), see Figure 1-1. The genome is the information one can extract from DNA, e.g. genes, variations between individuals, variations between species. Knowledge of a species genome is centrally important in biology. The genome of each individual is also likely to become increasingly important in the future, given the potential applications of personalized medicine [32]. Genome sequencing is essentially the process of bridging the biological object (DNA molecule) to the computational object (DNA sequence). A genome sequencer takes as input tangible DNA molecules, and outputs sequences in a textual format.

Sequencing returns fragments

However, this vision of sequencing as a black-box is an over-simplification. In practice, essentially due to technological constraints, the sequencing machine cannot output a complete DNA sequence. If it did, the textual sequence would exactly correspond to the sequence of nucleotides in the original molecule, and the story would end here.

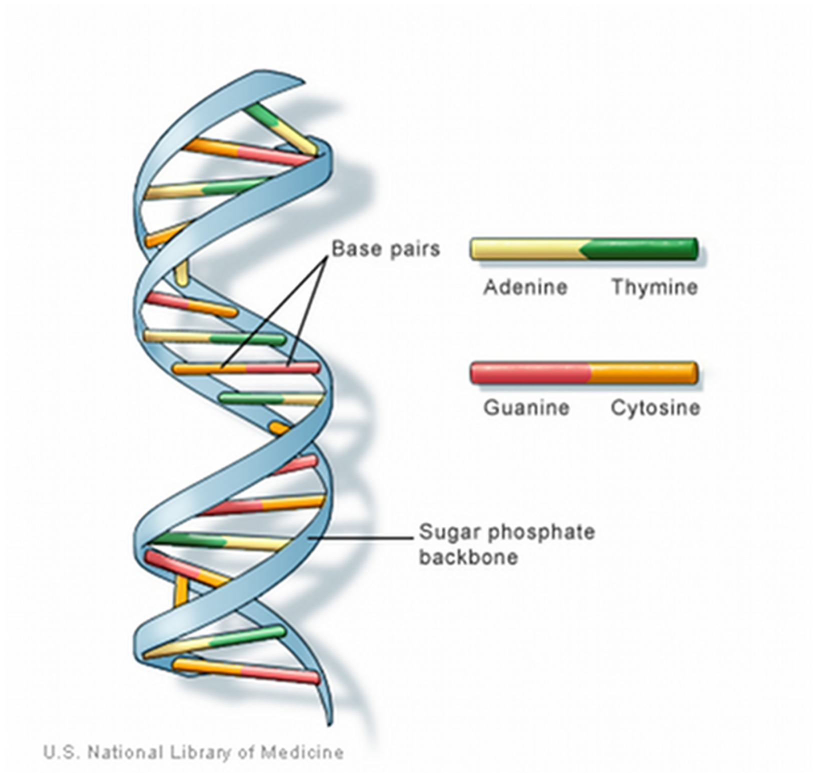

Figure 1-1: Structure of the DNA.

Instead, the sequencing machine outputs shorter, unordered fragments from random locations in the sequence. How short are these fragments? For the human genome, each fragment is only 0.000003% of the size of the genome [49]. This means that, to read each nucleotide of the genome at least once, hundreds of thousands of fragments are required.

A preliminary natural question is: is it even possible to recover the original sequence given only these short fragments? If the machine returns only one copy of the original sequence (each nucleotide is read exactly once), cut at random locations without any ordering information, the task would be impossible. But what if one is given, instead of one sequence cut at random locations, several copies of independently cut sequences?

Toy example of assembly

In this case, recovering the original molecule given only fragments is sometimes possible. Consider a toy example with a made-up sequence, `GATTACA`. Assume that the machine returns random fragments from a single copy of the sequence, in this case, `GATT` and `ACA`. Since the order of the fragments is not known, the original sequence could be either `GATTACA` or `ACAGATT`. Instead, if the machine returned two copies cut at random locations, such a set of fragments would be more helpful: {`GATT`, `ACA`, `GAT`, `TACA`}. Given this set, one can immediately rule out the solution `ACAGATT`, because it does not agree with the fourth fragment, `TACA`. Hence, the only solution is `GATTACA`.

This example is a simplified instance of the genome assembly problem, which will be the central topic of this thesis. In actual sequencing, one has to deal with millions or billions of fragments, yielding a potentially enormous number of candidate reconstructions. It should come to no surprise that genome assembly requires very efficient computational methods. Improving the quality of assemblies and lowering computational resources requirements is a very active research topic [35].

Toy example of re-sequencing

Genome sequencing essentially returns fragments of the original sequence. For some applications, knowing only fragments is sufficient; reconstructing the original sequence is unnecessary. Indeed, prior knowledge of sequences from other organisms/individuals can be used. Assume that $\texttt{GAT}\underline{\texttt{T}}\texttt{ACA}$ is the sequence of individuals of type A and $\texttt{GAT}\underline{\texttt{G}}\texttt{ACA}$ is the sequence of individuals of type B. The only difference between both types is a single nucleotide change at the fourth position (underlined). Then, sequencing an unknown individual and deciding its type is an easier problem than reconstructing its genome.

For instance, assume that an unknown individual (guaranteed to belong to either type A or B) is sequenced and the following set of fragments is returned: {$\texttt{GAT}\underline{\texttt{T}}$, `ACA`, `GAT`, $\underline{\texttt{T}}\texttt{ACA}$}. Fragments `ACA` and `GAT` are uninformative, as they are present in the sequence of both types. However, both $\texttt{GAT}\underline{\texttt{T}}$ and $\underline{\texttt{T}}\texttt{ACA}$ are sequences specific to type A, hence the unknown individual is of type A. This example is a simplified instance of re-sequencing a known genome to find variations.

Fragments length, error rate and coverage

As seen in the previous examples, genome assembly and re-sequencing appear possible given only a set of fragments, as long as useful fragments are sequenced. Since fragments originate from random locations, how can one guarantee that the sequencing machine will produce useful fragments with high enough probability?

First, fragments need to be long enough, as very short fragments tend to be uninformative. The extreme case is a fragment length of 1: knowing that the genome contains a `A` is certainly not useful. Similarly, given a length of 2, any string (say, `GA`) is likely to appear at plenty of locations in the genome. For very large genomes such as the human genome, fragments need to be of length of at least 16 nucleotides in order not to be trivially uninformative[1].

Second, sufficiently many copies of the genome need to be sequenced. This point was critical for the toy example of assembly. For the re-sequencing toy example, the motivation for many copies does not emerge clearly. However so far, no mention has been made of the accuracy of fragments; fragments were assumed to be perfect sub-strings of the original genome. In practice, the sequencing machine sometimes erroneously skips, inserts or changes a nucleotide. Fortunately, the observed rate of errors is typically low, below 2% of sequences nucleotides are erroneous in most sequencing machines [49]. Then, the same genome location needs to be sequenced multiple times, in order to rule out (by a majority vote) the possibility of having an error at any nucleotide.

In practice, the sequencer returns fragments in large quantities, exceeding the length of the original sequence by a factor of 5 to 200 [49]. This factor is said to be the sequencing coverage.

[1]Based on the expected number of occurrences of a random DNA string of length k inside a random genome of length $n = 3 \cdot 10^9$: $(n - k + 1) \cdot \left(\frac{1}{4}\right)^k < 1 \iff k > 15$.

Next-generation sequencing

From now on, we will refer to fragments originating from the sequencer as *reads*, as it is the most widely used term. Early sequencing machines (known as Sanger-generation sequencers) enabled low-coverage sequencing with relatively long reads, of length up to 900 nucleotides [46]. Since 2007, next-generation sequencing machines significantly increased the sequencing coverage while yielding shorter reads (36 to 500 nucleotides). Figure 1-2 shows the evolution of read lengths, and volume of sequences produced by a single run, for two leading next-generation sequencing technologies.

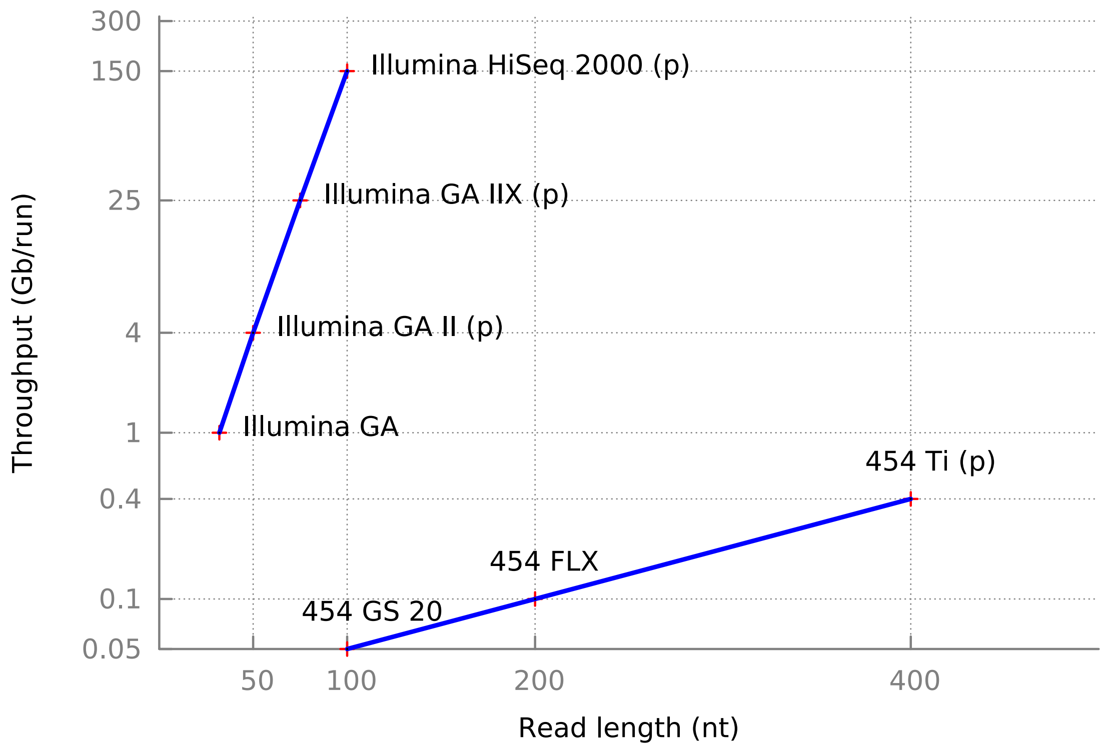

Figure 1-2: Evolution of DNA sequencing technologies, 2007-2011, in terms of throughput and read length. Data taken from companies websites.

Short fragments and sequencing errors are two practical aspects of genome sequencing. There exists other biases, such as uneven coverage, and non-uniform error profile.

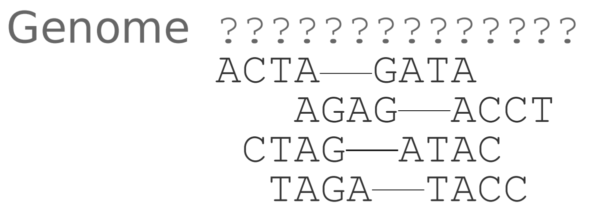

Figure 1-3: Sequencing a toy genome with paired reads of length 5 (inserts are of length 12).

Paired reads

Sequencers are able to produce *paired* reads. Paired reads are pairs of reads which are separated by a known distance in the genome. They are produced by sequencing both extremities of a long fragment. This long fragment will be referred to as the *insert*. For instance, in Figure 1-3, assume that the insert `ACTAGAGATA` is being sequenced, sequencing both its extremities with reads of length 5 produces the paired read $(\texttt{ACTA}, \texttt{GATA})$. There are two different sequencing processes that enable the production of paired reads. One process uses short inserts, of length typically not exceeding 500 nucleotides, which produces paired reads referred to as *paired-end* reads in the literature. The other process uses longer inserts (of length ranging from $1,000$ to $40,000$ nucleotides [54]), producing the so-called *mate-pairs*.

The concept of paired reads is central to this thesis, as several chapters focus on the difference between paired and unpaired reads, for re-sequencing and assembly applications.

1.2 Genome assembly

1.2.1 Earlier works

Early works on genome assembly considered the similarities between genomic sequence reconstruction and a well-known computational problem, the shortest common super-string problem [43]. As the name suggests, the latter problem consists in constructing a shortest possible string that contains all the given sub-strings. Consider for example the set of strings $S = \{\texttt{GAT}, \texttt{ATT}, \texttt{TTA}, \texttt{TAC}, \texttt{ACA}, \texttt{CAT}, \texttt{CAA}\}$, one can check that $s =$ `GATTACATCAA` is a super-string, as it contains every element of S as a sub-string. It is also a solution of shortest length, although it is not the unique solution (`CAAGATTACAT` is another solution). Given that genome sequencing returns (presumably) all possible genome sub-strings, it is tempting to think of the genome as a shortest common super-string. And indeed, early genome assembly relied on solving this problem.

$\boxed{\text{Genome assembly was first modeled as an instance of the shortest common super-string problem.}}$

However, the inherent parsimony of solutions to the shortest super-string problem does not fit well the structures of genomes. Genomes often contain many repetitions of a sub-sequence. Casting assembly as a shortest string essentially discards the possibility of repetitions. In the previous example, sequencing the genome $g =$ `GATTACATTACAA` also can also produce the set S (in fact, S contains exactly all the sub-strings of g of length 3). However, as $|g| > |s|$, the genome cannot be recovered as a shortest super-string of S.

Hence nowadays, the vast majority of approaches found in the literature rely on other models: sequence graphs [33]. There are two main sequence graph models: the de Bruijn graph and the string graph. Both aim to transform read sequences into a global representation, based on observed overlaps between the reads. More importantly, sequence graphs permit a representation of the genome repeats structure. Thus, the repeat collapsing problem in shortest common super-string formulations can be avoided. Elaborated traversals of those graphs allow to construct an assembly of the reads.

$\boxed{\text{Sequence graphs provide a better assembly model than the shortest common super-string.}}$

To give an intuition of this, consider a graph where the set S from the previous example is the set of nodes. Sequences which overlap by exactly two nucleotides are linked together by an edge (yielding a de Bruijn graph, Figure 1-4). For a moment, assume that the genome g is known (this is not the case in actual assembly). Then, one can order the reads by their position on the genome. The path in the graph which traverses all the nodes in that order spells g (modulo overlaps of length 2 that are repeated twice). In this example, this path happens to be the shortest path that goes through each node at least once. In general, paths having this property are called minimum-length Hamiltonian paths. The good news is that Hamiltonian paths can be found without prior knowledge of the genome, as they only depend on the graph topology. Finding minimum-length Hamiltonian paths is the theoretical foundation behind graph-based genome assembly.

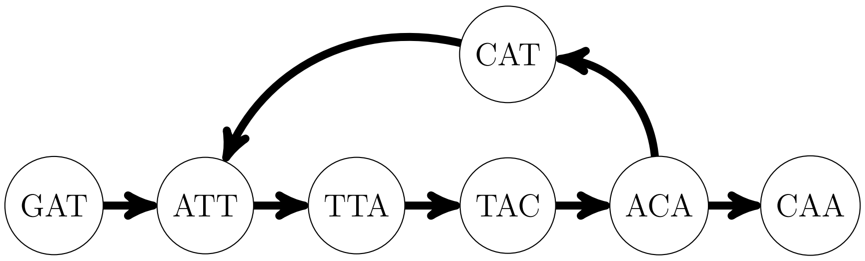

Figure 1-4: Example of a de Bruijn graph for the set of strings S.

1.2.2 Contribution

Sequence graphs have been widely used to generate high-quality genome assemblies. However, constructing a sequence graph requires hundreds of gigabytes of memory for vertebrate genomes. This prohibits assembly of species with larger genomes. This thesis work, independently of several other groups [4, 1, 8], demonstrated that high-quality assemblies consisting of contiguous, gap-less sequences (called *contigs*) can be constructed without building a complete sequence graph. This opens the possibility of assembling large genomes within reasonable computational resources.

Whole-genome sequence graphs are not required to assemble a genome into contigs.

Other groups cast the graph-less assembly in terms of a greedy algorithm. Roughly speaking, a typical greedy assembly starts from an un-assembled short region, extends it by one nucleotide repeatedly, and stops when there are 2 or more possibilities. There are several issues with greedy assembly. Perhaps the most prominent one is that small repeats are not elucidated, yielding more fragmented assemblies. Instead, we introduced the concept of *localized assembly*, which consists in partial graph construction and traversal. Localized assembly combines the memory benefits of greedy algorithms with the locally complete information of graphs.

Localized graph assembly can be used to assemble a genome into contigs.

We extended the localized assembly approach to fully take into account the information provided by paired reads. Our construction is based on the result that paired reads can be grouped together to robustly detect the absence of repetitions. Given this information, contigs can be extended to form gapped sequences (chains of contigs called *scaffolds*). We developed the first assembler (Monument) that constructs scaffolds directly from reads.

Localized graph assembly can be used to assemble a genome into scaffolds.

In the near future, assembly will have to tackle much larger genome instances (e.g., species with a genome larger than that of human, and meta-genomes). The combination of higher throughput from sequencing platforms and meta-genomes is expected to overwhelm current assembly models. We investigated the possibility of assembling genomes without any indexing structure. Mapsembler, a prototype implementation performing index-free assembly, was developed.

Contigs can be assembled without any indexing structure.

1.3 Thesis outline

Chapter 2 We study how paired reads enable better re-sequencing, in terms of genome coverage. We analyze several model genomes, ranging from bacteria to human, using a novel algorithm. Our results indicate that paired reads enable to reach coverage threshold that would have required unfeasibly longer single reads. Also, this work provided evidence that insert size is more important than read length for assembly.

Chapter 3 This chapter is a theoretical study of *de novo* assembly using pairing information. We show that paired reads can be incorporated into classical assembly models. We study the complexity of related graph theoretical problems. A paired assembly problem is formulated, which enables to derive new results on the parameterized complexity of assembly.

Chapter 4 The methodology presented in the previous chapter needs refinements in order to permit the assembly of actual genomes. In that chapter, a localized graph traversal scheme is introduced, taking into account pairing information. Also, indexing sequencing data into the assembly structure is typically the most memory-intensive step. We propose the first parallel and memory-efficient approach to index paired reads, using succinct hash tables.

Chapter 5 The algorithms presented in the previous chapter are combined in the implementation of a new assembly software, the Monument assembler. Some implementation choices are presented and Monument is benchmarked against state-of-the-art assembly software.

Chapter 6 As an effort to go beyond classical, memory-heavy assemblers, an index-free targeted assembler, Mapsembler, is presented. It can perform versatile targeted assemblies on a simple desktop computer. Also, several other applications for the succinct hash tables used in Chapter 4 are presented.

Chapter 2

Analysis of paired genomic re-sequencing

In the Introduction, genome sequencing, re-sequencing and de novo *assembly were introduced. In this second chapter, the gap between single and paired data in genome re-sequencing is investigated using novel methods.*

2.1 Motivation

Next-generation sequencing technology is enabling massive production of high-quality paired reads. These short read lengths were initially dedicated to re-sequencing applications. Re-sequencing consists in aligning reads to a reference sequence to improve it and/or to detect variations (e.g. SNPs, indels). Whiteford *et al.* [56] outline that the *feasibility of re-sequencing* is determined by the percentage of single (unpaired) reads that are present at a unique location in the reference sequence. In other words, read-length regions of the genome that are unique can be probed unambiguously by the reads. They performed simulations showing that short reads of length 50-100 nt (nucleotides) are long enough to re-sequence large (human chromosome sized) genomes.

An immediate extension the feasibility of re-sequencing is the feasibility of assembly. Whiteford *et al.* established the *feasibility of de novo assembly* can be checked with the following measure: the percentage of the genome covered by simulated contigs greater than a given length, in function of the length of the reads. Simulated contigs are constructed by iteratively extending a read, as long as a single overlap with another read is found. The simulated contigs are interrupted whenever a read overlaps with two or more reads. Another way to see the feasibility of assembly is to remark that contigs are interrupted by repeated regions, of length greater or equal to the read length. With this method, they demonstrate that longer reads (over 100 nt) should not significantly improve assembly contiguity.

There is a natural link between the feasibility of re-sequencing, which is given by the percentage of unique fixed-length regions, and the feasibility of assembly, which is a consequence of the repartition of non-unique regions of length greater than the reads. For this reason, we argue that analyzing the feasibility of re-sequencing with paired reads is an important step towards the feasibility of assembly with paired reads.

Note that the previous analyses of feasibility of re-sequencing and assembly only considered single reads. In terms of paired reads, Chaisson, Brinza and Pevzner [7] determined experimentally that the paired read length threshold for *de novo* assembly of the *E. coli* genome is ≈ 35 nt, and ≈ 60 nt for the *S. cerevisiae* genome.

By conducting an analysis extending Whiteford *et al.* results, we investigate to what extent genome re-sequencing is feasible with ultra-short paired reads. This extends the previous feasibility of re-sequencing studies to paired reads. We obtain theoretical read length lower bounds for coverage in re-sequencing. We also show that insert size, i.e. the distance between two reads in a pair, plays a crucial role in re-sequencing. This analysis provides evidence that insert size plays an important role in *de novo* assembly.

2.2 Reads uniqueness

2.2.1 Single reads uniqueness

This chapter adopts a perfect abstraction of the genome sequencing process. A genome of length n is given as an input; there is no assumption made on its sequence. A *read* is an exact substring the genome of fixed length l. The read length l is considered to be a constant, i.e. all the reads have the same length. An *occurrence* of a read is a position in the genome where the read appears. By definition, each read has one or more occurrences. A read r is said to be *unique* if it has only one exact occurrence in the genome. Here, uniqueness is perfect: no differences (substitutions, insertions or deletions) are allowed between occurrences. Perfect uniqueness is a lower bound for imperfect uniqueness measures, which is easier to compute. The single reads uniqueness ratio can be simply formulated as the number of unique reads divided by the total number of possible reads, i.e. $n - l + 1$:

$$SU = \frac{|\{\text{unique reads}\}|}{n - l + 1}$$

2.2.2 Paired reads uniqueness

We now turn to the perfect uniqueness of paired reads. We define a (σ, δ)-*pair* (r_1, r_2) as two sequences r_1 and r_2 that are separated by $\sigma \pm \delta$ nucleotides in the genome. The distance will never be negative, i.e. $\sigma \geq \delta$. The sequences r_1 and r_2 are said to be respectively the left and right *mates*. A (σ, δ)-pair can represent indifferently a mate-paired or a paired-end read (as defined in the Introduction). Typical values for σ are 300 for Illumina paired-end reads and 2000 for Illumina mate-paired reads. In practice, the insert size roughly follows a normal distribution centered at σ. A more crude approximation is often made: $\delta \approx 0.1\sigma$.

A (σ, δ)-pair (r_1, r_2) is (σ, δ)-*unique* if there is only one occurrence of (r_1, r_2) distant of $\sigma \pm \delta$ in the genome. More generally, a single read r is said to be (σ, δ)-*unique* if, for any read r' distant of $\sigma \pm \delta$ from r, there is only one occurrence of (r, r') distant of $\sigma \pm \delta$ in the genome (in other words, the (σ, δ)-pair (r, r') is unique).

The following relations are clear:

$$r \text{ is unique} \Rightarrow r \text{ is } (\sigma, \delta)\text{-unique}$$

$$r_1 \text{ is } (\sigma, \delta)\text{-unique} \iff \forall r_2, (r_1, r_2) \text{ is } (\sigma, \delta)\text{-unique} \tag{2.1}$$

2.2.3 Two definitions of paired uniqueness

As a preliminary step, the total number of (σ, δ)-pairs in a genome is:

$$(n - \sigma - \delta - 2l + 1) \cdot (2\delta + 1) + \delta^2 - \delta,$$

where the first term is the number of (σ, δ)-pairs having $(2\delta + 1)$ possible right mates, and the last two terms $(+\delta^2 - \delta)$ correspond to $\sum_{k=0}^{2\delta} 2\delta - k$, i.e. the left mate is at a position $n - k - \sigma - 2l + 1$ in the genome, $0 \leq k \leq 2\delta$, and there are only $2\delta - k$ possibilities for the right mate.

Given the definitions in the previous section, there are two ways to formulate a perfect paired uniqueness ratio for a genome. Consider these quantities:

$$U_1 = \frac{|\{(\sigma, \delta)\text{-unique pairs}\}|}{(n - \sigma - \delta - 2l + 1) \cdot (2\delta + 1) + \delta^2 - \delta}$$

$$U_2 = \frac{|\{(\sigma, \delta)\text{-unique reads}\}|}{n - l + 1}$$

The first quantity, U_1, is the ratio of the number of pairs (r_1, r_2) that are (σ, δ)-unique, over the total number of (σ, δ)-pairs (r_1, r_2). The second quantity, U_2, is the ratio of the number of reads r that are (σ, δ)-unique, over the total number of reads r. Given the implication 2.1, it is clear that $U_2 \leq U_1$. Figure 2-1 shows the actual computations of these ratios for the *Lambda phage* genome ($48.5 \cdot 10^3$ nt).

In the context of re-sequencing, U_1-uniqueness is more relevant than U_2-uniqueness. U_1-uniqueness gives the ratio of (σ, δ)-paired reads that map uniquely to the genome.

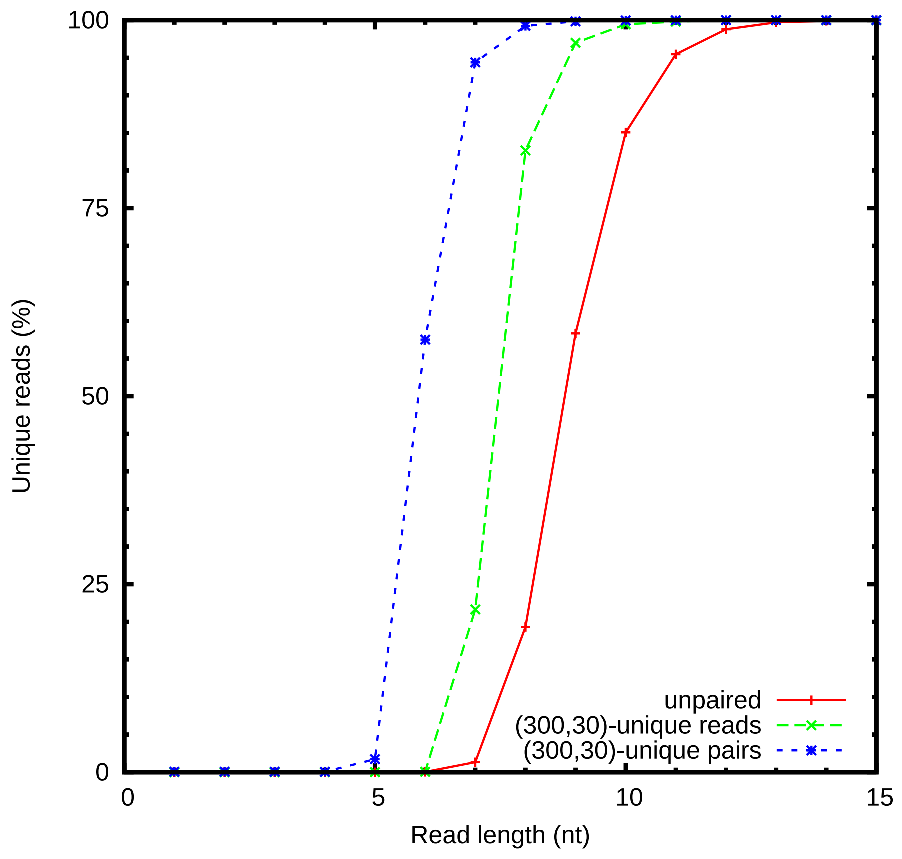

Figure 2-1: Comparison of single (in red), paired U_1 (in blue) and paired U_2 uniqueness (in green) of reads in the lambda-phage genome.

However, in the context of *de novo* assembly, U_2-uniqueness is directly related to single reads uniqueness. The gap between single reads uniqueness and U_2-uniqueness can be interpreted as repeated reads that become unambiguous as soon as paired information is available. Since we are interested in paired read length lower bounds, the lowest uniqueness ratio, i.e. U_2-uniqueness, will be considered in the remaining.

2.3 Algorithms

2.3.1 Suffix arrays

Two useful algorithmic objects for computing repetitions in a genome are the suffix array and the longest common prefix array. Given a string s, the *suffix array* is a lexicographically sorted sequence $(SA_i)_{i\geq 0}$ of all the suffixes $s[j..]$ of s, for $j \geq 0$. The *longest common prefix* (LCP) array $(LCP_i)_{i>0}$ is a sequence of integers, the value at position i is the length of the longest prefix common to SA_i and SA_{i-1}. An inverted index of LCP_i is introduced: for a given read length l, $H_l = \{i \mid LCP_i = l\}$. Note that H_l needs not be computed for values of l larger than the length of the longest repetition in the genome. Hence, in the following, we assume that l is trivially upper-bounded by $(n/2)$. The suffix array and the LCP can be constructed in linear time [26]. To illustrate these definitions, the suffix and LCP arrays of the word `babar` are shown in Table 2.1.

i	SA_i	LCP_i
0	abar	
1	ar	1
3	babar	0
2	bar	2
4	r	0

Table 2.1: Example of suffix array and LCP array for the word 'babar'.

2.3.2 Uniqueness ratio using a suffix array

We recall a method for computing single reads uniqueness in genomes [56]. The uniqueness ratio of reads in a genome can be computed with the suffix and LCP arrays of the genome. Note that for each read r, there exists an index i such that the read is a prefix of SA_i. If the read r is repeated elsewhere in the genome, then by the lexical ordering of the suffix array, r will also be a prefix or either SA_{i-1} or SA_{i+1}. Hence, in terms of longest common prefixes (recall that l is the read length):

$$r \text{ is a repeat } \Rightarrow \text{ either } LCP_i \geq l \text{ or } LCP_{i+1} \geq l \tag{2.2}$$

For all suffixes longer than l, the converse is true. Then, by negation, unique reads exactly correspond to indexes i of suffixes longer than l, such that $LCP_i < l$ and $LCP_{i+1} < l$. There are $(l-1)$ suffixes shorter than l, and they necessarily yield

LCP values strictly under l.

2.3.3 Single uniqueness algorithm

Whiteford [56] presented an algorithm for counting unique reads, based on the following incremental relation: the number of unique reads of length $l + 1$ is equal to the number of unique reads of length l plus the number of "new" entries. The new entries are those such that $(LCP_i = l, LCP_{i+1} \leq l)$, and those such that $(LCP_i < l, LCP_{i+1} = l)$. In the latter case, the strict inequality is introduced to avoid counting $(LCP_i = l, LCP_{i+1} = l)$ twice. The new entries are efficiently enumerated using the array H_l, which by definition gives all the indices i such that $LCP_i = l$.

The algorithm for counting unique reads is presented in Algorithm 1, and a new memory-efficient variation is presented in Algorithm 2.

Algorithm 1 Whiteford algorithm for reads uniqueness (`UniquesWhiteford`)

```
Require: $LCP_i$, $H_l$
1: procedure UniquesWhiteford($LCP_i$,$H_l$)
2:     $cnt \leftarrow 0$
3:     for increasing read length $l$ do
4:         for all $i \in H_l$ do
5:             if $LCP_{i+1} \leq l$ then
6:                 $cnt \leftarrow cnt + 1$
7:             if $LCP_{i-1} \leq l - 1$ then
8:                 $cnt \leftarrow cnt + 1$
9:         $unique_{l+1} \leftarrow cnt - l$
```

The variation, `UniquesVariation`, uses the fact that H_l and LCP_i are different representations of the same information. From the implication 2.2, the number of repeats of length greater than l is equal to the number of indices i such that either $LCP_i \geq l$ or $LCP_{i+1} \geq l$. The number of repeats of length exactly $l+1$ is the number of repeats added since length l. The algorithm requires a bit array to mark each suffix array index i and prevent it from being counted twice. The bit array requires less memory (n bits) than the LCP array ($n \log_2(max(LCP_i))$ bits).

Algorithm 2 Variation of Whiteford algorithm for reads uniqueness (`UniquesVariation`)

```
Require: $H_l$
1: procedure UniquesVariation($H_l$)
2:     $repeats \leftarrow 0$
3:     for decreasing read length $l$ (starting from $l = n/2$) do
4:         $MarkDuplicates(l)$
5:         $unique_l \leftarrow \frac{n - l + 1 - repeats}{n - l + 1}$
6: procedure MarkDuplicates($l$)
7:     for all $i \in H_l$ do
8:         if $i$ is not marked then
9:             $repeats \leftarrow repeats + 1$
10:            $mark(i)$
11:        if $i + 1$ is not marked then
12:            $repeats \leftarrow repeats + 1$
13:            $mark(i + 1)$
```

Both algorithms are linear in the length of the input string (since $(H_l)_{l \geq 0}$ contains exactly n elements).

2.3.4 Paired uniqueness algorithm

We present a novel algorithm that computes the U_2-uniqueness of paired reads for all read lengths, given fixed values for (σ, δ). Our algorithm is a non-trivial extension of Algorithm 1 to paired reads. For convenience, we define $Pairs_{(\sigma,\delta)}(i) = \{j \mid (s[i..i + l], s[j..j + l])$ is a (σ, δ)-pair$\}$. The principle of this algorithm is to first mark which single reads are duplicated (using the procedure `MarkDuplicates` of Algorithm 2). Then, non-unique (σ, δ)-pairs are found by considering two cases. If the left mate is unique, search for duplicate right mates. If the left mate is repeated, right mates are sieved.

The time complexity of Algorithm 3 is $O(Rn(1+\delta))$, where R is the length of the longest exact repetition and n is the size of the text. In the worst case, $R = n/2$; in an actual genome, $R << n$ is a constant.

Simple modifications to Algorithm 3 enable the computation of U_1-uniqueness: increment `paired_repeats` by the sum of occurrences of reads having ≥ 2 occurrences

Algorithm 3 Paired uniqueness of reads (U_2)

```
Require: $H_l$
1:  procedure PairedUniqueness($H_l$)
2:      for all decreasing read lengths $l$ (starting from $l = n/2$) do
3:          $MarkDuplicates(l)$
4:          $paired\_repeats \leftarrow 0$
5:          for all unmarked indices $i$ do
6:              Initialize a read counter
7:              for all $j \in Pairs_{(\sigma,\delta)}(i)$ do
8:                  increment count of $s[j..j+l]$
9:              if $\exists\ j \in Pairs_{(\sigma,\delta)}(i)$, $s[j..j+l]$ has $count \geq 2$ then
10:                 $paired\_repeats \leftarrow paired\_repeats + 1$
11:         Initialize a paired read counter
12:         for all marked indices $i$ do
13:             for all $j \in Pairs_{(\sigma,\delta)}(i)$ do
14:                 increment count of $(s[i..i+l], s[j..j+l])$
15:         for all marked indices $i$ do
16:             for all $j \in Pairs_{(\sigma,\delta)}(i)$ do
17:                 if $(s[i..i+l], s[j..j+l])$ has $count \geq 2$ then
18:                     $paired\_repeats \leftarrow paired\_repeats + 1$
19:                     break
20:         $paired\_unique_l \leftarrow \frac{n-\sigma+\delta-2l+1-paired\_repeats}{n-\sigma+\delta-2l+1}$
```

at line 10, and remove the `break` (line 19) to count the occurrences of each repeated pair. Finally, line 20, the number of reads $(n - \sigma + \delta - 2l + 1)$ should be replaced by the number of pairs, $(n - \sigma + \delta - 2l + 1)(2\delta + 1) + \delta^2 - \delta$.

2.4 Results

Actual genomes are analyzed, in contrast with a purely theoretical approach which would consider the genome as a random string of length n. Such theoretical model does not take into account that genomes contain significantly more repetitions than random strings.

Using Algorithm 3, we analyze viral, bacterial and eukaryotic genomes to determine the uniqueness of paired reads given a fixed pair distance of 300 nt. Then we study the impact of longer pair distances with high variability on the uniqueness of paired reads in the *E. coli* genome.

2.4.1 Paired vs. unpaired uniqueness

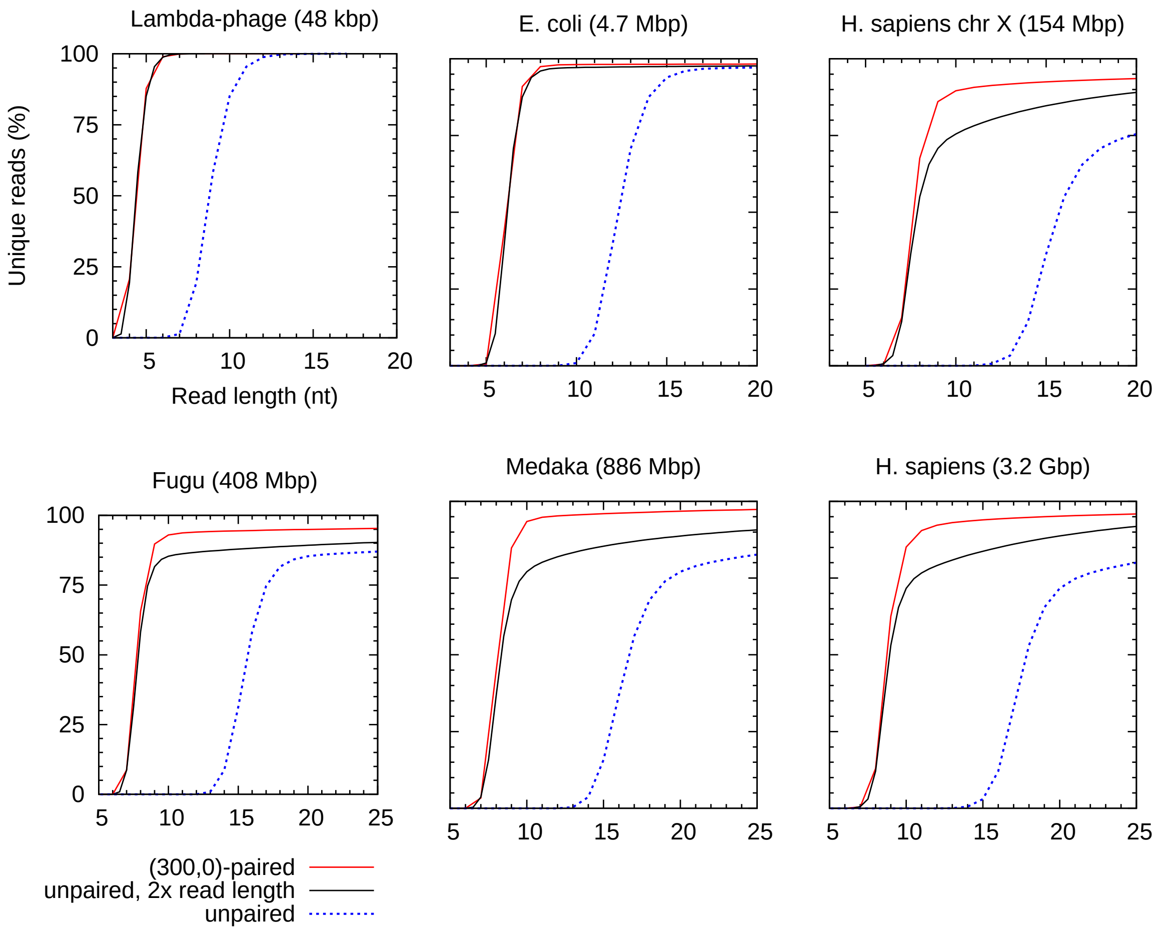

Figure 2-2: Percentage of unique $(300, 0)$-paired and unpaired reads as a function of read length for six genomes.

Each genome consists of both the forward and the reverse strands. A fair comparison between paired and unpaired uniqueness can be achieved by considering unpaired uniqueness of reads twice longer (referred as "unpaired, 2x read length" in Figure 2-2). Our hypothesis is that comparing the uniqueness of paired reads of length l and unpaired reads of length $2l$ essentially assesses the influence of the insert size.

We find that 97.4% of the *E. coli* genome is covered with unique paired reads of read length 8 nt, and 90% of the *H. sapiens* genome with unique paired reads of read length 11 nt and gap 300 nt (see Figure 2-2). These results suggest that for large genomes, re-sequencing requires significantly shorter (for *H. sapiens*, around 70% shorter) paired reads to achieve a coverage comparable to unpaired reads.

2.4.2 Influence of insert size

Moreover, there exists a trade-off between read length and pair distance. Figure 2-3 represents the paired U_2-uniqueness of the *E. coli* genome (in shades of yellow) as a function of read length and σ, δ parameters. Under the unrealistic $\delta = 0$ condition, for each pair distance σ, paired U_2-uniqueness greatly increases between read lengths 5 to 9 up to a plateau uniqueness value inside $[0.97; 1]$. Note that a read lengths ≤ 10 are extremely small for sequencing standards. However, the whole genome cannot be uniquely probed by these short paired reads with $\delta = 0$ until larger pair distances ($\sigma \geq 5000$).

Under the more realistic $\delta = 0.1\sigma$ case, it is observed that for small read lengths, U_2-uniqueness is degraded by longer pair distances. This behavior is a consequence of short random strings that are likely to appear as duplicate right mates inside any long enough region of size δ. However, longer read lengths enable to uniquely probe a larger portion of the genome even for large values of δ.

The following additional results are not shown in the Figure. Given a fixed ($\delta = 0$) pair distance of 5,000 nt (resp. 2,000 nt), the whole *E. coli* genome can be unambiguously probed by paired reads of length 18 nt (resp. 700 nt) or greater. When the uncertainty in pair distance is $\pm 10\%$, only a small part of the genome cannot be uniquely probed (resp. 0.3% and 0.1% in the previous cases).

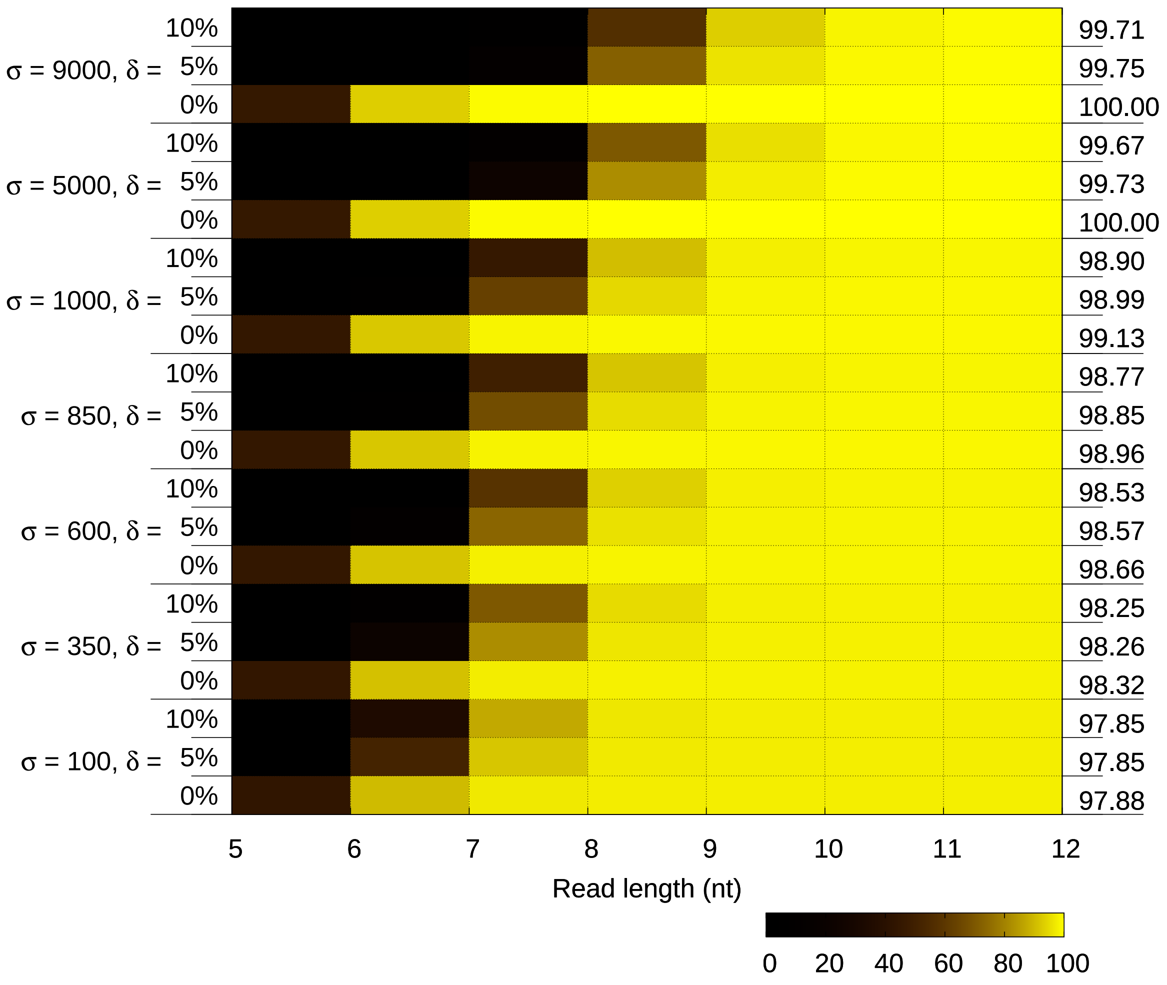

Figure 2-3: Relationship between read length, pair separation (σ), distance uncertainty (δ), and paired read U_2-uniqueness ratio (in shades of yellow) in the *E. coli* genome. Paired uniqueness ratio for the read length $l = 12$ are reported on the right column.

2.5 Discussion

This work provided rigorous evidence that paired reads enable more sensitive genome probing. Also, as it is technologically easier to sequence using long insert sizes than long read lengths, the good news is that a longer insert size compensates for a shorter read length. Genomic repetitions and the difficulty of assembly are strongly connected [56]. It is known that proper use of paired reads can significantly improve contiguity in genome assembly.

In terms of computation resources usage, our algorithm processes the human genome with $(\sigma, \delta) = (300, 0)$ in 2 days using 64 GB of memory and a single CPU core at 2.6 GHz. Similar resources usage was reported by RepAnalyse to compute single reads uniqueness [55]. Since this is a one-time computation, optimizing running time is not a critical aspect of the method.

Several directions were not explored. Inexact repeats were not considered, because suffix arrays record exact sequence information. Considering that a paired reads is repeated if it appears at a different genome location, within a fixed (typically low) edit distance threshold, would be a more realistic assumption. However, since this present study is a special case of inexact paired repeats, it is already useful to derive lower bounds on the read length. Moreover, recent techniques enable the compression of suffix arrays very efficiently (e.g. the FM-index v2 [18]). Such techniques would enable paired uniqueness analysis of larger genomes and meta-genomes on modest hardware, with only little modifications of the algorithms presented here. Finally, we did not explore the connection between paired uniqueness and paired assembly contiguity. A possible analysis would be to predict an ideal unpaired assembly (following [55]) then simulate scaffolding using the unique paired reads which link contigs unambiguously.

Chapter 3

Paired *de novo* assembly theory

In the previous chapter, the difference between single and paired reads was investigated in the context of genome re-sequencing. Now, we focus on the problem of de novo *genome assembly using pairing information. This chapter begins with an exposition of classical assembly computational models. Then, new computational models are designed which include pairing information. Finally, we investigate the parametric complexity of paired assembly.*

3.1 Introduction

de novo assembly consists in recovering a genome sequence given only a set of substrings (*reads*) obtained by DNA sequencing. This problem is challenging, as any DNA sequencing method only yields reads much shorter than the original genome. For the human genome, reads are millions of time shorter. Hence, to assemble a genome in a reasonable amount of time, it is necessary to efficiently process millions or billions of reads. Before 2007, assembling genomes was computationally easier, because DNA sequencing methods were able to produce relatively long reads (at a high cost) [46]. Several algorithmic formulations for the assembly problem were pioneered: Shortest Common Superstring, Hamiltonian Path, de Bruijn Superwalk (all covered later in this Chapter). These will be reviewed in this chapter. However, the performance of these algorithms severely degraded when dealing with the huge number of shorter reads produced by next-generation sequencers. As of today, there is still a need for better algorithmic models for *de novo* assembly.

Furthermore, every known assembly model corresponds to a NP-hard computational problem (references provided in this chapter). In other words, there is no known polynomial time algorithm that can solve the assembly problem. The reason behind this high complexity is that genomes can contain repetitive sub-sequences longer than the length of the reads. Hence, recovering a most likely assembly (typically, a minimum-length sequence) requires to consider up to an exponential number of possible assemblies. Hence there is little hope of ever designing a fast algorithm that solves the assembly problem under one of the known models. Whether there exists a satisfactory assembly model with lower complexity remains an open question.

We study the computational complexity of new assembly models related to the context of next-generation sequencing: the assembly of paired reads. How does assembling paired reads compare with the single reads, in terms of computational complexity? Paired reads assembly is not a generalization of single reads assembly: in contrary, the former can be seen as a constrained version of the latter. Can we hope to use paired constraints to design an assembly model with lower complexity than its classical counterparts? We formulate a negative answer to this question for some classical models: Shortest Common Superstring, Hamilton Path and de Bruijn Superwalk. We define a natural extension of classical models to paired reads and prove that the complexity of assembly under these models remains unchanged, even when given plenty of pairing information. We study the problem of jigsaw puzzle assembly and conclude that assembling a jigsaw puzzle with paired pieces is as hard as assembling a classical jigsaw puzzle. However, this does not rule out efficient paired assemblers based on heuristics. Eventually, we study the parametric complexity of paired assembly: assuming a genome contains only a certain class of repeats, we show that the assembly problem can be solved in polynomial time. This result provides evidence that pairing information contributes to making the assembly problem feasible in certain cases.

Our results rely on reductions from known problems. We recall the definition of a reduction.

Definition 1 (Reduction)**.** A reduction is a polynomial-time transformation of a problem A to a problem B.

Thus, if A is NP-hard and there exists a polynomial-time computable function f mapping instances I of A to instances $f(I)$ of B, then B is also NP-hard.

3.2 Classical assembly models

3.2.1 Genome assembly is not a Shortest Common Superstring

We say that s is a **superstring** of a set of strings $S = \{s_1, .., s_n\}$ if every s_i is a substring of s, and s satisfies the "no extra character" condition, that is, for any $i \in [1, |s|]$, at least one element of S is aligned with s at an interval containing i.

Problem 1 (Shortest Common Superstring, SCS)**.** For a set $S = \{s_1, .., s_n\}$ of strings over an alphabet Σ, find the smallest string s containing every string of S as a substring.

Note that, because of this definition of superstring, a solution of SCS is always a superstring. SCS is known to be NP-hard for strings of size greater or equal to 3 over a binary alphabet [24]. It is also Max-SNP hard (hard to approximate), and the greedy algorithm achieves an upper bound of $4OPT$ (always retrieves a string at most 4 times larger than the smallest string). It is conjectured that its actual upper bound is $2OPT$. In the early history of sequencing, several assemblers implemented variants of the SCS greedy algorithm [28].

The SCS problem, however, is not an ideal formulation of the assembly of sequencing reads. The following are two situations where the shortest common superstring has no biological meaning:

1. Tandem repeats collapsing: `ARRRRRB`$\rightarrow$`ARRB`

   Consider a genome of the form `ARRRRRB`, where each letter is a region of the size of the read length. Then, reads will only indicate overlaps inside `AR`, `RR` and `RB`. The SCS does not take into account the number of times the region `R` is seen in the reads. Hence, the SCS of this genome is `ARRB`, which is a biologically invalid approximation of the original genome.

2. Over-collapsing: `ARBRCRD`$\rightarrow$`ARBSCSD`, where `S` $=$`R`$[1..r]+$`R`$[|$`R`$|-r..]$ and r is the read length.

   Another example is given by a genome of the form `ARBRCRD`. This time, each letter is a region much larger than the read length. Notice that the sequence `ARBSCSD` (with `S` defined as above) is also a common super-string of the reads. And it is even shorter than the genome. Each instance of `R`, except one, has been replaced by a compressed instance (`S`) to minimize redundant information. Since the repeated region `R` is reconstructed fully only once in the SCS, this solution is not a biologically sound one.

To avoid these problems, assembly models which explicitly account for such repeat cases were designed: string graphs and de Bruijn graphs.

3.2.2 String graphs

String graphs are a natural representation of overlaps of a set of strings. Here, we consider a very simple definition of an overlap between strings. Two strings (r, r') are said to *k-overlap* if a suffix of r matches exactly a prefix of r' over a fixed length of k characters. For a set of strings S, the **overlap graph** $OG^k(S)$ is a directed graph. Its set of vertices is S, and an edge $s_1 \rightarrow s_2$ is present if there is a k-overlap between s_1 and s_2. The **string graph** $SG^k(S)$ is a directed graph whose set of vertices and edges are included in those of the overlap graph. Contained reads (reads which are included in other reads) and transitive edges (edges of the form $s_1 \rightarrow s_3$, whenever there exists s_2 such that $s_1 \rightarrow s_2$ and $s_2 \rightarrow s_3$) are removed [37]. Figure 3-1 shows an example string graph for the set of strings $\{abcd, bcde, cdef, defi, efic, ficd, icde, defg, efgh\}$.

In practice, string graphs are constructed with overlaps based on inexact matching, where a percentage of identity is defined between strings. The string graph model is not tied to a specific overlap definition.

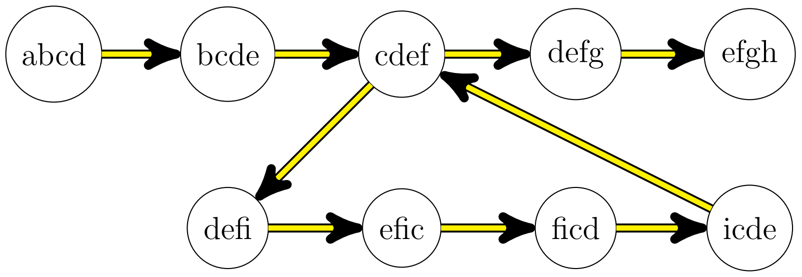

Figure 3-1: Example of a string graph with 3-overlaps

Genome assembly using string graphs has been as a computational problem, referred here as the Assembly Problem [38].

Problem 2 (Assembly problem, AP)**.** Find a path which visits each node of $SG^k(S)$ at least once (generalized Hamiltonian Path), minimizing path-string length.

This problem is NP-hard, by a reduction [33] from the Shortest Common Superstring problem. An interesting development showed that the Assembly Problem is hard only because of repeats [38]. Its parameterized complexity is polynomial in specific cases where repeats lengths are bounded. For instance, and perhaps counter-intuitively, assembly becomes easy if all repeats are longer than $2r - k + 1$, where r is the read length. The intuition behind this surprising result is that the absence of shorter repeats removes the possibility of false overlaps between reads. Given only true overlaps, the assembly problem can be solved in polynomial time as an instance of the Chinese Postman Problem [17].

3.2.3 de Bruijn graphs

For a set of strings S, the **de Bruijn graph** (S) is a directed graph whose vertices are all the k-length substrings of each string in S, and an edge $s_1 \rightarrow s_2$ is present if there is a $(k-1)$-overlap between s_1 and s_2.

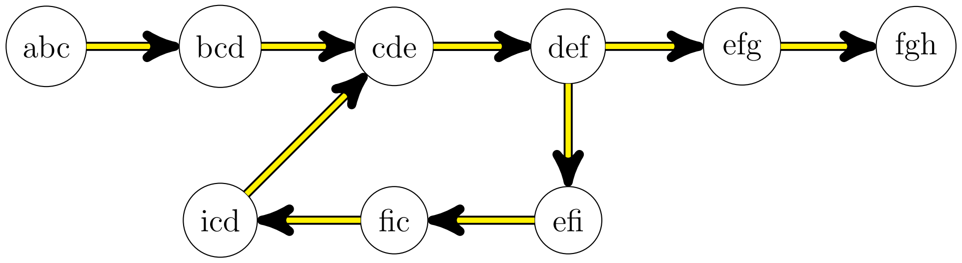

Figure 3-2: The de Bruijn graph of the same set of string as Figure 3-1 for $k = 3$

Genome assembly using de Bruijn graphs has been formulated as a computational problem known as the de Bruijn Superwalk Problem [33]. If s is a string, a **walk** $w(s)$ is a path in $dBG^k(S)$ of the form $s[1..k] \rightarrow s[2..k+1] \rightarrow ... \rightarrow s[|s|-k+1..|s|]$. For a path w', a subpath w of w' is a path over a subset of the nodes and edges of w. We say that w is a **subwalk** of w' if w is a subpath of w'. Observe that any path w in $dBG^k(S)$ is a walk $w(s)$ for some string s. A **superwalk** for a set of strings S is a walk w' such that for every $s \in S$, $w(s)$ is a subwalk of w'.

Problem 3 (de Bruijn Superwalk Problem, BSP)**.** Given a set of strings $S = \{s_1, .., s_n\}$ over an alphabet Σ and an integer k, does a minimum length superwalk for S in $dBG^k(S)$ exist?

This problem is NP-hard for $|\Sigma| \geq 3$ and any $k \geq 1$ [33].

3.2.4 Scaffolding a sequence graph

Scaffolding consists in ordering DNA sequences given pairing information. It is never applied to order a set of reads, but rather to a set of longer sequences constructed from the reads. This problem finds its roots in the heuristics of assembly. In the next chapter, it will be seen that reconstructing the genome in one single sequence is not always feasible due to repetitions. Instead, a set of sub-sequences (called *contigs*) is constructed from the single reads. Pairing information is then used to order the contigs. This is formulated as the following computational problem. The contig graph is defined as follows: $V = \{contigs\}$, $E = \{(c_1, c_2) \text{ s.t. } |\{(r_1, r_2), r_1 \in c_1, r_2 \in c_2\}| \geq t\}$ where t is an arbitrary threshold, indicating that contigs are linked if they are supported by at least t paired reads.

Problem 4 (Scaffolding problem [19])**.** Find an ordering of contigs in the bi-directed contig graph that is supported by a maximal number of paired reads.

Scaffolding requires a complete set of contigs. In most assembly implementations, pairing information is exclusively used during the scaffolding step. The question we seek to answer in the following is: can paired information be used at read-level assembly, and is it an improvement?

3.3 Shortest Common Superstring of paired strings

The classical Shortest Common Superstring problem can be extended to deal with paired reads. First, some definitions are given. A string s_2 is **aligned** with s_1 at some interval $[i, j]$ if $s_1[i, j] = s_2$. A **σ-gapped string** s over an alphabet Σ (such that $\{-\} \notin \Sigma$) is a string of the form $s = x -^{\sigma} y$, where x and y are two single reads in Σ^*. A string $s \in (\Sigma \cup \{-\})^*$ is a **substring** of a string $s' \in \Sigma^*$ (which we denote by $s \in s'$ for brevity) if there exists an index i such that, for any j such that $s[j] \in \Sigma$, $s'[i + j] = s[j]$. In other words, this is a classical substring definition where the symbol '$-$' is treated as a wildcard character. This allows us to define a variant of SCS.

Problem 5 (Shortest Common Superstring of σ-gapped strings, $PSCS$)**.** For a set $S = \{s_1, .., s_n\}$ of σ-gapped strings over an alphabet Σ, find the shortest gap-less string $s \in \Sigma^*$ containing every string of S as a substring.

Theorem 1. *For $\sigma \geq 1$ and $|\Sigma| \geq 3$, $PSCS$ is NP-hard.*

Proof. We reduce an instance $S = \{s_1, .., s_n\}$ of SCS to an instance S' of $PSCS$, where a new symbol ($\sharp$) is added to the alphabet. We first define a function $f_\sigma(s)$ that transforms s such that a string $\sharp^\sigma$ is inserted before each letter of s, with one extra $\sharp$ at the end, e.g. $f_1(abc) = \sharp a\sharp b\sharp c\sharp$. Observe that the length of $f_\sigma(s)$ is $(|s|+1)(\sigma+1)-1$. Let $g_\sigma(s)$ be a function that, given an input of the form $s = s_1\sharp^\sigma s_2$ with s_1, s_2 non-empty strings and the length $|s_1|$ of s_1 is minimal among all possible choices of s_1, replaces $\sharp^\sigma$ by a gap $-^\sigma$, e.g. $g_1(\sharp a\sharp b\sharp c\sharp) = \sharp a - b\sharp c\sharp$. Let $S' = \{s'_1, .., s'_n\}$, where

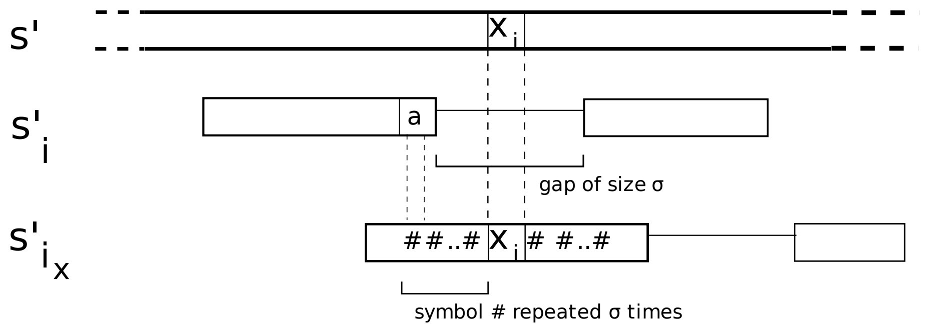

Figure 3-3: Illustration for the proof of Theorem 1

$s'_i = g_\sigma(f_\sigma(s_i))$. We will now show that the existence of a superstring of S of size n is equivalent to the existence of a superstring of S' of size $(n+1)(\sigma+1)-1$.

Let s be a superstring of S of size n. Since $s_i \in s$, $f_\sigma(s_i) \in f_\sigma(s)$ by the definition of f_σ, and $g_\sigma(f_\sigma(s_i)) \in f_\sigma(s)$ by the definition of g_σ. Hence, $f_\sigma(s)$ is a superstring of S' of size $(n+1)(\sigma+1)-1$.

Conversely, let s' be a superstring of S' of size n' that does not contain $\sharp^{\sigma+1}$ as a substring. Note that the shortest superstring of S' verifies this condition. For each occurrence of s'_i in s', let us show that such occurrence is actually $f_\sigma(s_i)$, that is, that the gap $-^\sigma$ in s'_i is filled by the string $\sharp^\sigma$ in s'.

For the sake of contradiction, let us assume that one of the characters that fills this gap in s' is $s'[x_i] \neq \sharp$. Since s' is a superstring, there exists i_x such that s'_{i_x} is aligned with s' at an interval containing $s'[x_i]$. By the definition of s'_{i_x}, the character $s'[x_i]$ is either followed by a gap and preceded by $\sharp^\sigma$, or followed by $\sharp^\sigma$ and preceded by a gap, or both followed and preceded by $\sharp^\sigma$. Either way, a string $\sharp^\sigma$ follows or precedes $s'[x_i]$ in s'. However, in the string s'_i, the gap is immediately followed and preceded by a non-$\sharp$ character, which leads to a contradiction (see Figure 3-3 for an illustration).

Therefore s' is a superstring of $f_\sigma(S)$, and recall that it does not contain $\sharp^{\sigma+1}$. Then s' is of the form

$$\sharp^\sigma s'[\sigma+1]\sharp^\sigma s'[2(\sigma+1)]\sharp^\sigma..\sharp^\sigma s'[|s'|-\sigma]\sharp^\sigma,$$

and $f_\sigma^{-1}(s)$ is well-defined. By removing the $\sharp$ symbols, we obtain that $s_i \in f_\sigma^{-1}(s')$.

Hence, $f^{-1}(s')$ is a superstring of S of size $\frac{n'-\sigma}{\sigma+1}$.

We have shown that there exists a superstring of S of size n if and only if there exists a superstring of S' of size $(n+1)(\sigma+1)-1$ that does not contain $\sharp^{\sigma+1}$. Any solution to SCS for S can be mapped polynomially from a solution to $PSCS$ for S', which concludes the reduction. $\square$

Since the reduction above increases the length of superstrings by only a constant factor $(\sigma+1)$, it is said to be a L-reduction [25], therefore $PSCS$ is Max-SNP hard. That is, no polynomial algorithm that can approximate $PSCS$ with polynomially small error exists. Whether an algorithm can approximate $PSCS$ with worst-case constant approximation ratio is an open question. The greedy algorithm for SCS that consists of choosing a seed and repeatedly merge it with the longest overlapping string may not be easily extended to $PSCS$, since gaps are unlikely to be successfully filled using the greedy approach.

Note that a slightly modified problem, where the gap size σ is allowed to vary between $[\sigma-\delta,\sigma+\delta]$, is also NP-Hard and Max-SNP hard, since it is a generalization of $PSCS$.

3.4 Two paired variants of graph problems

3.4.1 Hamiltonian Path with paired vertices

Problem 6 (Hamiltonian Path, HP)**.** Given a directed graph $G=(V,E)$, does G have a Hamiltonian path, i.e. a path which visits each vertex $v \in V$ exactly once?

The HP problem is classically known to be NP-complete. A **paired graph** $G=(V,E,L)$ is defined as a graph (V,E) with additional directed edges $L \subset V^2 \backslash E$ representing a special pairing between two vertices.

Problem 7 (Hamiltonian Path with paired vertices, PHP)**.** Given a paired graph $G=(V,E,L)$ where $|V|=n$ and an integer σ, does G have a Hamiltonian path $w=w_1 \rightarrow .. \rightarrow w_n$ satisfying the following constraint: for each $(v_i,v_j) \in L$, there exists an index l such that $v_i = w_l$ and $v_j = w_{l+\sigma}$. That is, each pair of vertices in L are exactly $\sigma - 1$ vertices away from each other in the Hamiltonian path.

When $L = \emptyset$, the problem is trivially HP. A slightly more constrained flavor of the PHP problem is when $|L| = \Theta(|V|))$. We denote this flavor as PHP-largeL. It is no longer a generalization of HP, because solving HP without the constraints does not guarantee that a solution satisfies the constraints. The following proof show that PHP-largeL still difficult (NP-complete).

Theorem 2. *For any integer $\sigma \geq 1$, PHP-largeL is NP-complete.*

Proof. We reduce HP to PHP-largeL. For a graph $G = (V, E)$, construct $G' = (V', E', L)$ as follows: for each $v \in V$, define σ new vertices $v^{(1)}, .., v^{(\sigma)}$ in V' such that E' contains the path $v^{(1)} \rightarrow v^{(2)} \rightarrow .. \rightarrow v^{(\sigma)}$. Then, for each edge $v \rightarrow w$ in E, add the edge $v^{(\sigma)} \rightarrow w^{(1)}$ in E'. Let L be the set of edges $v^{(1)} \rightarrow v^{(\sigma)}$ for each $v \in V$.

We now show that any Hamiltonian path of G' corresponds to a Hamiltonian path of G. Let w' be a Hamiltonian path of G'; for each $v \in V$, the vertices $v^{(1)}, .., v^{(\sigma)}$ are visited exactly once in w', therefore w' contains $v^{(1)} \rightarrow .. \rightarrow v^{(\sigma)}$. Transform the path w' into w such that, for every $v \in V$, the subpath $v^{(1)} \rightarrow .. \rightarrow v^{(\sigma)}$ of w' is replaced by the single vertex v in w. Then w is a path visiting every $v \in V$ exactly once, therefore it is a Hamiltonian path of G. □

3.4.2 de Bruijn Superwalk Problem with σ-gapped strings

We define a variant of the de Bruijn Superwalk Problem which explicitly deals with the paired nature of reads. de Bruijn graphs over σ-gapped strings are defined as a natural extension of de Bruijn graphs. Intuitively, the gap symbol '$-$' is treated as a wildcard in walks.

Formally, let S be a set of σ-gapped strings. Let $G = dBG^k(S)$ be the classical de Bruijn graph of S, with no special semantic associated with the gap symbol. Let us define a relation $\prec$ between two strings s_1 and s_2 as follows: $s_1 \prec s_2$ if s_1 can be obtained by replacing some characters in s_2 by gaps, i.e. $a - -c \prec ab - c$. We extend the definition of classical de Bruijn graphs to associate to each vertex v a set of labels $\{s \in (\Sigma \cup \{-\})^k | v \prec s\}$. These labels are all possible replacements of gaps in v by non-gap characters. We also extend the definition of overlap to allow gaps: s_1 and s_2 overlap if there exists a string s such that a suffix s_1' of s_1 and a prefix s_2' of s_2 are both substrings of s, and $|s_1'| = |s_2'| = |s|$. Consequently, there is an edge $v_1 \rightarrow v_2$ if one of the labels of v_1 overlaps with one of the labels of v_2 by exactly $k-1$ characters. In this new definition of de Bruijn graphs, walks are well-defined for σ-gapped strings. Let s' be a σ-gapped string, and assume that it is a substring of a gap-less string s. Recall that the notation $w(s')$ indicates that $w(s')$ is a walk in the de Bruijn graph following the consecutive k-length substrings of s'. If $w(s)$ is a walk in $dBG^k(S)$, then $w(s')$ is also a walk in $dBG^k(S)$ and $w(s')$ is a subwalk of $w(s)$.

Problem 8 (de Bruijn Superwalk Problem with σ-gapped strings, $PBSP$)**.** Given a set of σ-gapped strings $S = \{s_1, .., s_n\}$ over an alphabet Σ and an integer $k \geq 1$, decide whether a minimum length superwalk w of $dBG^k(S)$ exists, such that $w = w(s)$ and s contains no gap symbol.

We now show that reads pairing does not affect the complexity of DNA assembly with de Bruijn graphs.

Theorem 3. *For $k, \sigma \geq 1$ and $|\Sigma| \geq 4$, $PBSP$ is NP-hard.*

Proof. We reduce $PSCS$ to $PBSP$ by following the reduction of SCS to BSP [33]. Recall that SCS is NP-hard even for an alphabet of size 2 [24]. Two new symbols will be added to the alphabet in this reduction: $\{-, \sharp\}$. For an instance $S = \{s_1, .., s_n\}$ of σ-gapped strings (recall that each s_i is of the form $x_i -^\sigma y_i$), create the de Bruijn graph of σ-gapped strings $G = dBG^k(f_k(S))$, where f_k is the function defined in Theorem 1 (also the same function f as in [33]). In Theorem 1, this function was used to inflate the size of the SCS by a factor of $\sigma + 1$. In this reduction, the size of the walk will be inflated by a factor of $k+1$. When applying f_k to a σ-gapped string, a string of $\sharp^k$ is inserted between each symbol, including between each gap symbol. We shall show that the length of the shortest superwalk of G is $k+1$ times the length of the shortest superstring of S.

Given a (gap-less) superstring s of S, let w be the walk $w(f_k(s))$. For any i, the string $f_k(s_i)$ is a substring of $f_k(s)$, so $w(f_k(s_i))$ is subwalk of w, therefore w is a superwalk of G. Furthermore, it is a superwalk of length $|w| = (k+1)|s|$.

Conversely, suppose that w is a superwalk of G of the form $w = w(s')$, where s' does not contain any gap symbol. We recall the following observations, originally made in [33]. Each walk $w(f_k(s_i))$ is a sequence of cycles $C_{s_i[1]} \to .. \to C_{s_i[|s_i|]}$, where $C_c = w(f_k(c))$ for $c \in \Sigma \cup \{-\}$ (each cycle starts and ends with the node $\sharp^k$). Also, w can be uniquely expressed as a longer sequence of cycles $C_{j_1} \to .. \to C_{j_{\frac{|w|}{k+1}}}$. Let s be the string $j_1..j_{\frac{|w|}{k+1}} \in \Sigma^*$. Since $w(f_k(s_i))$ is a subwalk of w, there exists m such that $C_{s_i[1]} \to .. \to C_{s_i[|s_i|]}$ is a subwalk of $C_{j_m} \to .. \to C_{j_{m+|s_i|-1}}$. Since the decomposition of walks into cycles is unique, instead of an equality as in [33], we obtain the following inclusion: $s_i \prec j_m..j_{m+|s_i|-1}$. Therefore, $s_i \in s$. Hence, the string s is a superstring of S of length $\frac{|w|}{k+1}$.

We have shown that there exists a (gap-less) superstring of S of size n if and only if a gap-less superwalk of G of size $\frac{n}{k+1}$ exists, therefore the reduction is proved. □

3.5 Paired-pieces jigsaw puzzle

There is a strong analogy between our previous results and the assembly of jigsaw puzzles. Consider a simple jigsaw puzzle where each piece is a square whose edges can either be straight, or augmented with a tab or a pocket of arbitrary shapes. Deciding whether n jigsaw pieces exactly fit into a $\sqrt{n} \times \sqrt{n}$ square box is NP-complete [14]. We define a variant of such puzzle in the spirit of the difference between single reads assembly and paired assembly.

Problem 9 (Paired-pieces jigsaw puzzle)**.** Given xy rectangular pieces with tab, pocket or straight edges, each piece being linked to another piece by a string of finite length, decide whether these xy pieces exactly fit into a $x \times y$ box with the additional constraint[1] that every string must be tightened.

[1]one may also see it as a hint

In other words, one knows how far apart two pieces must be in a solution. How hard is this variant? Merely discarding links and solving the classical puzzle does not guarantee to obtain a solution that satisfies all the links constraints. We show that this variant belongs to the same complexity class.

Theorem 4. *The paired-pieces jigsaw puzzle is NP-complete.*

Proof. We reduce the jigsaw puzzle to this problem. For a classical jigsaw puzzle instance, cut each piece into 12 paired sub-pieces. The middle sub-piece at each edge should have the same outer edge attribute (tab, pocket or straight) as the original piece. Set every other sub-piece outer edge as a straight line. Inner edges of every sub-pieces are given an unique tab/pocket attribute. Figure 3-4 describes the construction,

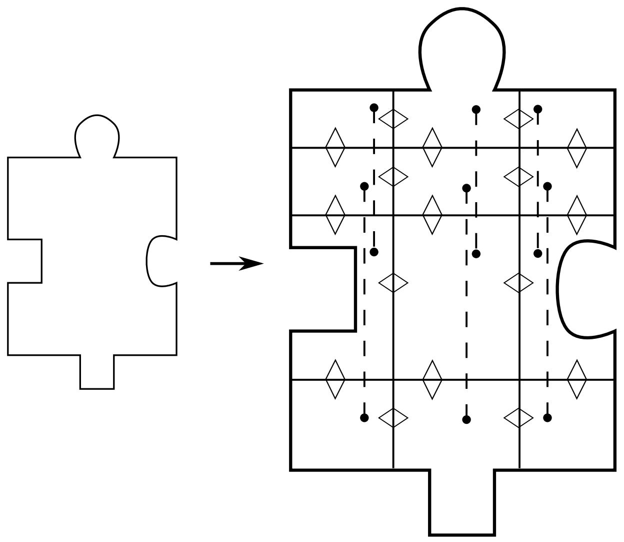

Figure 3-4: Reduction of a jigsaw puzzle to a paired-pieces jigsaw puzzle (proof of Theorem 4)

where each $\diamond$ symbol must be replaced by an unique tab/pocket attribute, and dashed lines show a possible pairing between pieces. Deciding whether the resulting paired-pieces jigsaw puzzle fits in a $\sqrt{n} \times \sqrt{n}$ square is equivalent to deciding whether the original puzzle also fits. Therefore, we have reduced the jigsaw puzzle to the paired-pieces jigsaw puzzle. $\square$

The instances of paired-pieces jigsaw puzzle related to DNA assembly correspond to puzzle pieces where strings have (roughly) the same length. This hardness result also generalizes to puzzles where each piece has paired links to more than one other piece. This is related to sequencing reads where the same mate of a paired read is sequenced again, this time with a different mate.

3.6 Paired assembly problem

This section deals with paired reads. To introduce some notations, the set R_1 is always the set of left mates (the left read in a paired read), and similarly, R_2 is the set of right mates. A set S of paired reads is thus noted $S \subset R_1 \times R_2$. When discarding pairing information, we refer to the set of reads as $R_1 \cup R_2$.

The paired string graph is defined as an extension of the classical string graph over a set of paired reads $S \subset R_1 \times R_2$. Recall (from the introduction) that the **insert size** of a paired read is the sum of both read lengths and the length of the gap between them. Two reads $(r, r') \in R_1 \cup R_2$ are said to *k-overlap* if a suffix of r matches a prefix of r' exactly over k characters. For a set of paired reads $S \subset R_1 \times R_2$, the *paired string graph* $PG^k(S)$ is defined as a directed graph by assigning a vertex to each read in $R_1 \cup R_2$. An edge $r \rightarrow r'$ is created between two reads if r k-overlaps r' (*overlap edge*). A special type of edge $r \dashrightarrow r'$ is created if (r, r') is a paired read (*paired edge*). Classical string graph transformations are applied: reads that are substrings of other reads, and transitively redundant overlap edges are discarded (paired edges are ignored during this step). No transitive reduction is performed for paired edges. For instance, consider the sequence $S = ab\mathbf{cd}ef\mathbf{cd}gh$ and perfect sequencing with insert of length 6 and paired reads of length 2. The paired string graph of these reads is drawn in Figure 3-5.

A *mixed path* in the paired string graph is a succession of vertices linked by either overlap edges or paired edges, e.g. $r_1 \rightarrow r_2 \dashrightarrow r_3 \rightarrow r_4$. *A path-string* is a string corresponding to the concatenation of nodes strings along a mixed path. The path-string is formed by the following rules: after an overlap edge, the string is appended with the concatenation of both nodes strings with their overlap repeated only once; after a paired edge, the string is appended with a gap corresponding to the paired

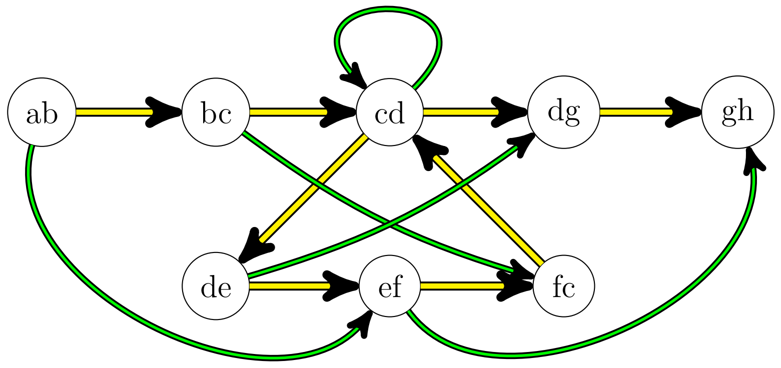

Figure 3-5: Example of a paired string graph from paired reads (insert size of 6) covering the sequence $S = ab\mathbf{cde}f\mathbf{cd}gh$. Green edges represent paired links and yellow edges represent 1-overlaps between reads.

insert size. In Figure 3-5, the path-string of $p = ab \rightarrow bc \dashrightarrow fc$ is $abc -^2 fc$, where $-$ still denotes a single-character gap.

Similarly to the Assembly Problem, the *Paired Assembly Problem* can be defined as a constrained flavor of AP.

Problem 10 (Paired Assembly Problem)**.** For a set of paired reads $S \subset R_1 \times R_2$, Finding a path that visits each node at least once (generalized Hamiltonian path) in $PG^k(S)$, and corresponds to a path-string s such that:

- the length of s is minimized, and
- for every pair (r, r') in S, the distance between r and r' in s matches the paired insert size.

Similarly to AP, this problem can also be shown to be NP-hard. Note that this problem is not an immediate generalization of AP: an instance of the Paired Assembly Problem cannot be straightforwardly constructed from an instance of AP, because $|S|$ (possibly non-trivial) pairing constraints need to be constructed.

In Chapter 4, we will focus on constructing a collection of subpaths that belong to all possible solutions of the Paired Assembly Problem.

3.7 Parametric complexity of paired assembly

This section extends a parametric complexity result by Nagarajan *et. al.* [38] to paired assembly. The result will be recalled after the following definitions. A repeat is defined as a string of length $\geq k$ (the overlap parameter in the Assembly Problem formulation) that occurs more than once in the genome. Let $[r_{min}, r_{max}]$ be respectively the shortest and largest read lengths. An overlap edge in the (paired or classical) string graph is **required** if it is the unique in-edge or the unique out-edge of a node. All other overlap edges are said to be **optional**. It is easy to see that required edges are necessarily part of a solution of the (Paired) Assembly Problem, while optional edges may or may not.

Theorem 5 ([38]). *The Assembly Problem can be solved in polynomial time if each repeat instance shorter than r_{min} is included in a single node having only required edges into and out of it, and each repeat instance of length in the range $[r_{min}, 2r_{max} - k - 2]$ has a read contained within it.*

We now consider a special class of repeats: those which are shorter than the insert size, and each instance of these repeats is separated by an insert size distance from any instance of any repeat. As a reminder, we consider that the insert size of a paired read is the sum of both read lengths and the length of the gap between them.

Definition 2 (**Short i-interspersed repeats**). A repeat is short and i-interspersed if it is shorter than $i - r_{max} + k$, and each instance of this repeat is flanked to the left and to the right by a repeat-free region of length greater than $i - r_{min} - k$.

We introduce the notion of semi-complete pairing, as a way to ensure that, at least, repeat-elucidating pairs are sequenced. In the remaining, we use the letters x, y, z to denote reads (which are nodes in the paired string graph), and the symbol $\dashrightarrow$ still indicates read pairing.

Definition 3 (**Semi-complete pairing**). For any read x, $\exists$ reads (x', x'') s.t. $x \dashrightarrow x'$ and $x'' \dashrightarrow x$. In other words, each sequenced read is the right mate of a pair and the left mate of another pair.

This is a weaker assumption than a complete set of pairs. For instance, consider an instance of a short i-interspersed repeat in the genome. Semi-complete pairing only requires at least two paired links for each read inside this instance. Whereas with actual complete pairing, i.e. if the sequencer returned all possible pairs, there would be significantly more links (twice as many paired links as the number of instances of this repeat).

Theorem 6. *Assuming semi-complete pairing with constant insert size $i > 2r_{max}$, if all repeats in the genome are short i-interspersed repeats, then the Paired Assembly Problem can be solved in polynomial time.*

The proof will consists in simplifying the graph, such that any remaining edge $x \rightarrow y$ need to be included in any solution (denoted by $x \xrightarrow{\text{in genome}} y$). This first lemma establishes how pairs with exact insert size propagate the necessity of including an edge in a solution.

Lemma 7. *Assuming constant insert size, if $x' \xrightarrow{\textit{in genome}} y'$, $x \dashrightarrow x'$ and $y \dashrightarrow y'$, then $x \xrightarrow{\textit{in genome}} y$.*

Lemma 8. *Under the hypotheses of Theorem 6, let $e = x \rightarrow y$ be an optional edge. For any x' s.t. $x \dashrightarrow x'$, for any y' such that $x' \rightarrow y'$, then $x' \xrightarrow{\textit{in genome}} y'$.*

Proof. Since e is optional, by definition there exists $z \neq y$ such that $x \rightarrow z$, and by transitive reduction y and z cannot have an edge between them. Hence, the k-suffix of x is contained in a repeat instance R. By semi-complete pairing, there exists x' s.t. $x \dashrightarrow x'$. Since R is a short i-interspersed repeat, it extends by at most $i - r_{max}$ to the right. By pairing, since the k-suffix of x' is at a genome distance $d > i - r_{max}$ to the k-suffix of x, it is not part of a repeat. Since repeat-free regions consist of only required edges, any out-edge of edge x' is required. □

We can now prove Theorem 6.

Theorem 6. The proof strategy is to show that any optional edge that is not part of the genome can be detected and removed from the graph, producing a simplified graph. Then, since the simplified graph only consists of edges that need to be included in any solution, the problem can be solved as an instance of a variation of the Chinese Postman problem [17] in polynomial time. The variation enforces pairing constraints, leaving the complexity unchanged. Let $e = x \rightarrow y$ be an optional edge. Let $\{x'_i\}$ be the set of right mates of x, and $\{y'_j\}$ be the set of right mates of y. By semi-complete pairing, these sets are non-empty. The first case we consider is the existence of a pair (i, j) s.t. $e' = x'_i \rightarrow y'_j$. By Lemma 8, e' is part of the genome. By Lemma 7, that makes e part of the genome.

We now turn to the case where there is no pair (i, j) s.t. $x'_i \rightarrow y'_j$. Assume, for the sake of obtaining a contradiction, that e is part of the genome. Then, at a distance i in the genome from an instance of the sequence represented by e, there exists a pair of reads (x', y') such that $x' \rightarrow y'$, we refer to this edge as e'. Since e is part of a repeat, a similar argument to that of Lemma 8 yields that e' is repeat-free. As x' and y' are present in only one copy in the genome, by semi-complete pairing, the paired links $x \dashrightarrow x'$ and $y \dashrightarrow y'$ must exist. Then, there exists (i, j) such that $x' = x'_i$ and $y' = y'_j$ and $x'_i \rightarrow y'_j$, yielding a contradiction. Hence, e can be removed from the graph as it is not part of the genome. □

3.8 Discussion

In this chapter, we reviewed the classical mathematical objects and problems in *de novo* assembly: strings and the Shortest Common Superstring problem, string graphs and the generalized Hamilton Path problem, de Bruijn graphs and the de Bruijn Superwalk problem. We defined a natural extension of these objects and problems to paired reads, introducing σ-gapped strings and paired graphs. We proved that the complexity of these problems is unchanged by introducing pairing constraints. We introduced a parallel between assembly and jigsaw puzzles, as evidenced with the result that jigsaw puzzle with paired pieces is as hard to solve as a classical jigsaw puzzle. This provides intuition that, no matter how paired information is used in an assembly algorithm, it cannot help escaping the NP-hardness within current models.

However, this does not rule out efficient paired assemblers based on heuristics (this will be covered in the next section). Eventually, we studied the parametric complexity of paired assembly. We showed that the paired assembly problem can be solved in polynomial time when repeats are shorter than the insert size and interspersed. In fact, the construction used to prove this result provides a formal mechanism to solve a certain class of repeats using paired reads..

Chapter 4

Practical assembly methods

The previous chapter covered the theoretical aspects of de novo *genome assembly using pairing information. Several practical aspects of assembly merit specific attention. For instance, actual genome assemblies cannot be obtained using paths proposed in the theoretical model (explained in 4.2.1). In this chapter, we propose a graph traversal scheme that takes into account pairing information. Also, indexing sequencing data into the assembly structure is the most memory-intensive step. We propose the first parallel and memory-efficient approach that creates a data structure to reference paired reads.*

4.1 Introduction

Even producing an approximation of the genome is a computationally difficult task. For assembly of human-sized genomes using short reads ($<$ 100 bp), current state of the art implementations (using de Bruijn graphs) require hundreds of gigabases of memory and several CPU weeks of computation [31]. Surveying current assembly implementations is a vast task, as each assembler implements its own set of heuristics [35]. However, from a high level perspective, the vast majority of assemblers rely on the same mathematical objects: the string graph, the de Bruijn graph, or a k-mer based index used for greedy assembly. Hence, some of the most popular assemblers can be classified as such:

de Bruijn graph Allpaths [20], SOAPdenovo [31], Velvet [57], ABySS [51].

string graph Newbler (unpublished), CABOG [52], SGA [50].

***k*-mer index** PE-Assembler [1], Ray [4], Meraculous [8].

These assemblers implement data structures that were designed to assemble single reads. However, most of these are in fact capable of assembling paired reads as well. A natural question is: when, and how is pairing information taken into account? The general answer is that pairing information is used to improve the quality of an assembly done without pairing constraints. The typical pipeline of an assembler is as follows:

1. Treat all paired reads as single reads, by essentially discarding pairing information.
2. Assemble these single reads, either as a set of contigs or as a simplified graph
3. The initial assembly is improved by using paired reads, either by scaffolding contigs or by performing further graph simplifications

Note that independent software can also be used to perform scaffolding [42, 22]. It may appear unsatisfactory to perform paired reads assembly using graph simplification or scaffolding, as such approach requires to solve unpaired assembly beforehand, which deliberately ignores pairing information.

Previous research has explored the benefits of using paired-end reads during contigs construction. The Arachne assembler searches for pairs of paired Sanger reads where both mates overlap to construct contigs [2]. The Shorty assembler uses pairing information to greedily construct contigs from paired reads anchored to long reads [21]. PE-Assembler extends contigs greedily and attempts to resolve ambiguous extensions using paired reads anchored nearby [1]. Medvedev et al. recently introduced the paired de Bruijn graph formalism, which incorporates pairing information in the de Bruijn graph [34]. Donmez et al. also recently proposed an approach to transform a string graph into a mate-pairs graph [15]. Each of these approaches aim to resolve repeats when constructing contigs. They will be reviewed in more details in the next section.

In the next section, assembly of paired reads is formalized using the paired string graph representation. It is shown that scaffolds correspond to paths in the graph under ideal sequencing conditions. The definition of these paths is then refined to account for sequencing errors and biological variants.

4.2 Issues with existing models

4.2.1 Limitations of theoretical assembly

Practically, assembly cannot be solved as an instance of the Assembly Problem or the de Bruijn Superwalk Problem. The following two practical issues arise:

Repetitions Many minimal-cost solutions are possible.

For instance, in Figure 4-1, there are two possible reconstructions: $abcdefcdijcdgh$ and $abcdijcdefcdgh$.

Imperfect coverage As edges or nodes are missing, a generalized Hamiltonian (resp. Eulerian) path does not exist.

Classical heuristics consists in outputting a set of linear paths (contigs) from the string graph or the de Bruijn graph. Until recently, pairing information was not used in contigs construction. The paired assembly problem, formulated in the previous section, provides a natural framework to include pairing information in contigs construction using a string graph. We now survey other existing approaches to construct contigs using pairing information.

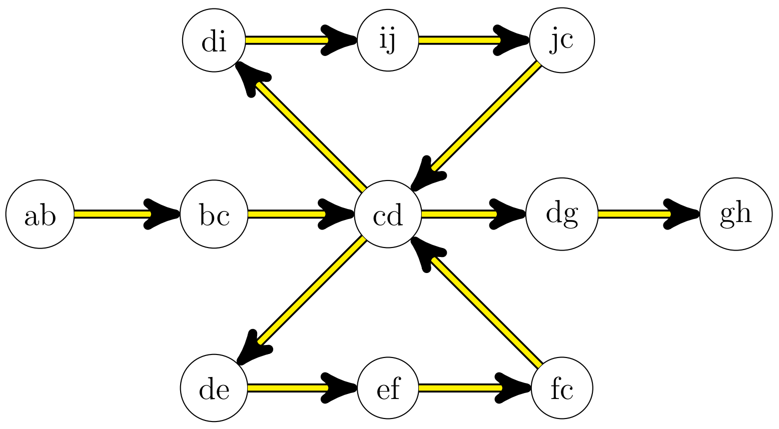

Figure 4-1: Problem with theoretical assembly (where the solution is a connected path): many ambiguous reconstructions.

4.2.2 Including pairs in contigs assembly

Incorporating paired reads in the first stage of assembly is a novel idea. As shown in the previous chapter, it requires a significant shift from classical assembly models. Three different approaches have been very recently published.

1. The paired de Bruijn graph [34]

   The problem of assembling paired reads is formulated using a graph where vertices are paired k-length substrings $(k_1|k_2)$ of paired reads (r_1, r_2). Edges of a single type are created by linking component-wise prefix and suffix of $((k+1)\text{-mer}|(k+1)\text{-mer})$. For example, an edge linking two nodes is of the form: $(AG|TG) \overset{(AGC|TGT)}{\rightarrow} (GC|GT)$.

2. The mate-pair graph [15]

   This formulation is based on the string graph. A preliminary step is carried to find paths between paired reads in the string graph. A new graph is constructed where each vertex is a paired read, and an edge links two paired read paths which overlap.

3. Greedy with paired consistency [1]

   This approach consists in greedily extending a starting sequencing with reads that satisfy a pairing consistency criterion.

All these approaches aim to produce contigs, i.e. gap-less sequences. In the following, we consider a natural extension: is it possible to use paired information to generate scaffolds (gapped sequences) directly from the reads? This approach is essentially different as the three reviewed methods, because it uses paired reads for direct scaffold construction. One main advantage is that missing read overlaps (possibly due to sequencing artifacts, such as coverage gaps or localized errors) can be represented by gaps in scaffolds, whereas they would necessarily interrupt contigs. Note that all methods, including ours, do not implement mechanisms to resolve repetitions longer than the insert size.

4.3 Non-branching paths

We describe the heuristics used to practically approximate the paired assembly problem. This construction enables to directly construct scaffolds from the paired reads, without performing a complete single reads assembly beforehand. This work has appeared in the proceedings of WABI 2011 [11]. Note that this section introduces the fundamental result on which the implementation in the next chapter is based.

4.3.1 Non-branching paths in the ideal case

Scaffolds can be directly constructed from the graph by following special types of mixed paths. To illustrate this, we first assume unrealistic sequencing conditions: error-free reads, perfect coverage (all the possible paired reads exist) and exact insert size. These conditions will be relaxed in the next section.

A mixed path p of length $|p|$ is *non-branching* if, for each pair of consecutive nodes $(n_i, n_{i+1}), 0 < i < |p|$ in this path, the following two conditions are met:

1. if the edge between n_i and n_{i+1} in the path p is a paired edge, then n_{i+1} must have an in-degree of 1 in the graph with respect to paired edges, and n_i must have an out-degree of 1 in the graph with respect to paired edges.

2. if the edge between n_i and n_{i+1} in the path p is an overlap edge, then n_{i+1} must

have an in-degree of 1 in the graph with respect to overlap edges, and n_i must have an out-degree of 1 in the graph with respect to overlap edges.

In other words, non-branching paths are mixed paths that traverse portions of the graph where no branching occur, with respect to the edge type. In traditional assembly heuristics, a contig can be represented as a NBP where each edge is an overlap edge (*simple path*). For example, maximal-length contigs from the paired string graph in Figure 3-5 on page 45 are the following simple paths:

$$\{ab \rightarrow bc \rightarrow cd,$$

$$cd \rightarrow de \rightarrow ef \rightarrow fc \rightarrow cd,$$

$$cd \rightarrow dg \rightarrow gh\}.$$

In contrast, a non-branching path that involves paired links is

$$\{ab \dashrightarrow ef \dashrightarrow gh\},$$

where the path-string $(ab -^2 ef -^2 gh)$ is a scaffold which covers the whole string. Under ideal sequencing conditions, non-branching paths immediately correspond to valid scaffolds. One can also consider *in- (resp. out-) branching paths*, for which only out- (resp. in-) degree of nodes in the path with respect to the corresponding edge type is 1. It can be shown that such paths are also valid scaffolds (proof omitted).

4.3.2 Practical non-branching paths

In actual sequencing, we distinguish two situations: undetected paired branching and additional overlap branching. These situations are illustrated in Figures 4-2 and 4-3. Previously, paired branching was always detected because of perfect coverage and exact insert size.

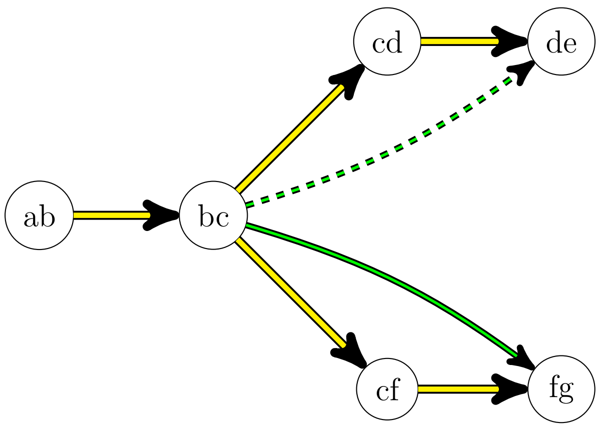

Figure 4-2: Issues in practical assembly: undetected paired branching. The paired link $bc \dashrightarrow de$ was not sequenced, hence the link $bc \dashrightarrow fg$ incorrectly represents the only paired link from bc. This example indicates that non-branching paths should not rely on single paired links.

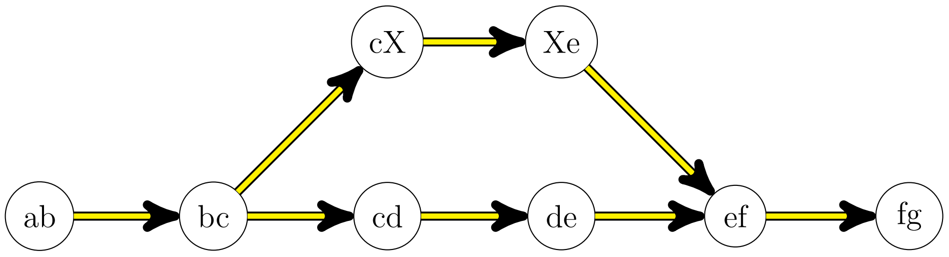

Figure 4-3: Issues in practical assembly: additional overlap edges due to errors. The letter d was incorrectly sequenced as X in the reads cd and de. This error introduces a so-called bubble structure between bc and ef.

Undetected paired branching

Now, it is no longer sufficient for a node to have an unique paired edge in order to unambiguously extend a scaffold. Weaker conditions can be formulated to detect the absence of paired branching, given imperfect coverage and variable insert size. First, assume that the insert size deviation is bounded by a constant i. Second, consider a simple path p of length $2i + 1$, and let n be the central node (p_{i+1}).

Postulate 1. A paired edge $n \dashrightarrow n'$ is considered to satisfy the non-branching condition if the sub-graph induced by the opposite mates of nodes in p is a simple path p' containing n'.

In other words, it is possible to detect that p' is the only genomic region which

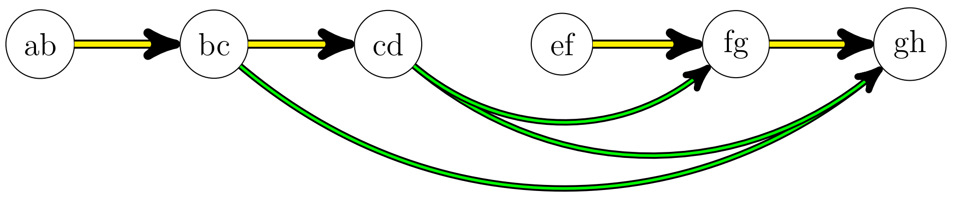

Figure 4-4: Illustration of Postulate 1: paired non-branching condition. Several paired links are used to ascertain that a paired edge satisfies the non-branching condition. Here, the sub-graph induced by opposites mates of the path $p = ab \rightarrow bc \rightarrow cd$ of central node bc is a simple path $p' = fg \rightarrow gh$. Hence, the paired edge $bc \dashrightarrow gh$ satisfies the non-branching condition.

appears at approximately an insert distance further than p. See Figure 4-4 for an example. The original definition of non-branching paths can then be extended to include this condition in place of the paired degree condition.

Additional overlap branching

Furthermore, sequencing errors and biological variants introduce additional branching in the graph. The branching structures are referred as *bubbles* (multiple paths that starts and ends at the same nodes) and *tips* (short interrupted paths) [57]. See Figure 4-3 for an example.

Graph-based assembly algorithms remove bubbles and tips after the whole graph is constructed [35]. Here, the bubble detection technique presented in [57] is adapted to also detect tips. As these structures are short, one can set a maximal length $d > 0$ for paths within them. A general characterization of these structures can be made in terms of sub-graph traversal. Observe that both structures form sub-graphs which start at a single node, and paths which are not interrupted converge after a certain length. A general definition of these events can be formulated. For a fixed integer $d > 0$, a *variant sub-graph* with starting node n of a paired string graph G, is a sub-graph of G such that the breadth-first search of G from n yields a single node of depth d.

Postulate 2. An edge $n \rightarrow n'$ is considered to satisfy the non-branching condition if it belongs to a variant sub-graph.

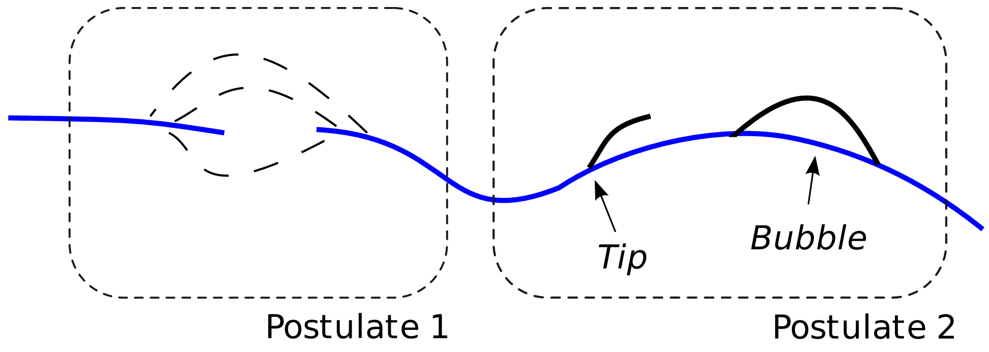

Figure 4-5: Practical non-branching path traversal (blue line) of a paired string sub-graph. Thick lines represent paths of overlap edges. Dashed lines represent paired edges between reads. Postulate 1 is used to traverse a gap, as paired reads link together two simple paths. Postulate 2 is used to traverse small branching regions (a tip and a bubble).

In other words, non-branching paths are extended to permit traversal in short branching sub-graphs through overlap edges. Figure 4-5 illustrates both postulates. In summary, we define a *practical non-branching path* as follows:

- for each path paired edge $n \dashrightarrow m$, both n and m correspond to middle nodes of simple paths of length $2i+1$ for which Postulate 1 is verified.
- for each path overlap edge $n \rightarrow m$, either the overlap out-degree of n and the overlap in-degree of m are both 1, or $n \rightarrow m$ is part of a variant sub-graph.

Note that setting $i = 0$ and $d = 1$ corresponds to the original definition of non-branching paths. Practical in-branching (resp. out-branching) paths are defined similarly, except that Postulate 1 only needs to be verified for n (resp. m).

4.4 Parallel and memory-efficient indexing

This section deals with time and memory limitations of current assemblers. As we will see, these limitations mainly come from the data structure used by the assemblers. Until the emergence of next-generation sequencing (NGS) technologies, software for assembling genomes could process up to millions of long ($\sim 10^4$ bp) reads. Now, a typical genome assembly instance for a vertebrate genome consists of billions of short

(100 bp) reads. Despite this technological shift, computational models and data structures for assembly remained essentially based on constructing and simplifying a monolithic structure (graph or ad-hoc index).

Graph-based assembly models are the most popular ones [35]. They require the construction of a large graph: either a string graph containing all the reads (not contained in other reads), or a de Bruijn graph containing all the k-length substrings (k-mers) of the reads. Graph-based NGS assembly tools rely on optimized implementations of these graph models. For human-sized genomes, an optimized de Bruijn graph assembler requires hundreds of gigabytes of memory [31]. For more details concerning graph-based assembly implementations, refer to a recent survey [35]. In a near future, larger eukaryotic genomes and meta-genomes will be sequenced at a faster pace than computational resources growth. Hence, graph-based models need to be further optimized to sustain the increasing rate of NGS technologies.

Several recent theoretical advances have been proposed to reduce the memory usage of graph-based assemblers. Simpson et al. implemented compression techniques (the FM-index [18]) to construct the string graph [50], saving memory at the expense of moderately higher running times. Conway et al. used succinct bit array structures [39, 44] to construct an immutable (that cannot be modified) de Bruijn graph efficiently [13]. Distributed de Bruijn graph construction using a message passing interface have been implemented in several assemblers (notably ABySS) [23, 51, 29]. Still, the memory usage of graph-based assembly methods remains very large (30-150 Gb for a human genome).

Greedy assemblers use a different assembly strategy. Instead of constructing and simplifying a graph, they rely on a static index, with a typically smaller memory footprint. This index is queried to recursively extend a starting sequence, until branching is detected. Previous implementations of greedy assemblers used a prefix tree to store reads [53], which consumes significantly more memory than a de Bruijn graph. Recent optimized implementations use custom k-mer indexing structures for memory efficiency [1, 4, 8]. In particular, these implementation have been applied to complex mammalian genomes, demonstrating that greedy assemblers are no longer limited to bacterial genomes. Unlike de Bruijn graph assemblers, data structures used in greedy assemblers typically contain references to read sequences. Hence, efficient read indexing is necessary to keep memory usage low.

The greedy/graph dichotomy has been persistent in assembly techniques. In other words, each assembler falls into one of the two following categories. (i) Either it is based on a greedy index and will never consider using graphs during assembly, (ii) or it constructs a large graph as its primary data structure. We seek to overcome this dichotomy by constructing a lightweight greedy index, which will later support the construction of small sub-graphs during assembly. The assembly part will be the topic of the next chapter.

In the next section, we propose a parallel reads indexing procedure designed specifically for greedy assembly. Two novel filtering methods are introduced to reduce memory usage: a procedure to remove erroneous k-mers on the fly, and a procedure to avoid referencing redundant reads. Finally, a prototype implementation is applied to real Illumina data to validate the method.

This work appeared in the proceedings of the Workshop on Parallel Computational Biology 2011, and was made in collaboration with G. Chapuis [9].

4.4.1 Distributed and multi-threaded indexing

A multi-threaded, multi-node procedure for reads indexing is proposed. A hash table is constructed, where the entries are k-mers, and the values are references to reads (i.e. identifiers, without sequence or position information). Taking advantage of shared memory between threads, reads sequences are stored separately in memory, without redundancy within a node. Index construction is distributed among N nodes, and each node performs independent computations in parallel. Specifically, each node n is running T_n threads, each thread t_n constructs a separate sub-index $I(n, t_n)$. A binning method adapted from [51] assigns each k-mer to a unique sub-index. Let h be a k-mer hash value obtained by classical collision-free hashing of a k-mer [51]: each nucleotide is mapped to a 2 bits number, yielding a $2k$ bits hash. The corresponding

k-mer belongs to the sub-index $I(n, t_n)$ if:

$$\begin{cases} h \ mod \ N = n \\ h \ mod \ T_n = t_n \end{cases}$$

which ensures that each sub-index contains distinct k-mers. Each thread reads the entirety of the input data to construct its sub-index. When all the sub-indexes are constructed, a single-pass, external-memory scan of all the sub-indexes merges them into a complete index. Hence, the indexing procedure always constructs the same complete index on different architectures. In the following, two algorithmic ingredients are described for parallel sub-index construction: k-mers filtering and reads indexing.

4.4.2 On-line k-mers filtering

Memory efficiency is crucial when assembling NGS data. In many approaches, including the one proposed here, memory consumption is proportional to the number of indexed k-mers. It is therefore important to *filter* erroneous k-mers (by removing them from the index) as early in the indexing process as possible. Erroneous k-mers are produced whenever the sequencing process makes a mistake during base calling. The abundance distribution $K_t^n(m)$ is defined as the number of k-mers seen exactly m times at indexing time t by node n. A key fact is that the hash function used above evenly distributes k-mers among sub-indexes. Hence, each $K_t^n(m)$ is identically distributed as the entire distribution $\sum_n K_t^n(m)$. This observation enables independent, parallel filtering for each sub-index. The superscript n is then omitted in the following.

The distribution of $K_t(m)$ at final time t is multi-modal. A large number of k-mers occur significantly less than the expected sequencing coverage: these are mostly sequencing errors. Assuming uniform sequencing coverage, the distribution of correct k-mers is a Gaussian mixture. The most abundant component is centered at the expected coverage of the target genome. Less abundant components are centered at multiples of the coverage, due to repeats in the genome. The proposed method consists in (i) detecting two components, one corresponding to erroneous k-mers and the other corresponding to correct ones, as soon as they separate sufficiently from each other, and (ii) finding an appropriate erroneous threshold (cut-off value). Every k-mer that has appeared fewer times than the erroneous threshold so far is then considered as an error and removed. In the future, this procedure could possibly be extended to correct errors in reads, but it is outside the scope of the current indexing scheme.

Error detection The following inequalities (4.1 and 4.2) must be satisfied to trigger filtering. First, erroneous k-mers are identified by their abundance. A classical hypothesis made in next-generation sequencing is that the multiplicities of erroneous k-mers follow a Poisson distribution with $\lambda < 1$ [7]. Given a random genome and uniform coverage with fixed-length reads, the probability that a k-mer is erroneous, given that is has been seen m times, remains not trivial to derive [12]. Thus, we will simply consider that an erroneous k-mer is strictly less likely to appear $m + 1$ times than m times. Thus, in our model, the abundance of erroneous k-mers peaks at $m = 1$ and has a strictly decreasing slope. The low end $m_{low}(t)$ is computed as the largest m that satisfies $K_t(m-1) > K_t(m)$ for $m \geq 1$. Then, the peak abundance $m_{high}(t)$ of correct k-mers is computed as the parameter at which the maximum value of $K_t(m)$ is attained for $m > m_{low}(t)$. Erroneous and correct k-mers are considered to be separated when:

$$m_{high}(t) - m_{low}(t) > r \tag{4.1}$$

where r is a user-defined resolution parameter. Second, to avoid the computational cost (linear traversal of the index) of filtering too soon or too often, a constraint is imposed on the amount of erroneous k-mers. Let S_{min} be a minimum amount (user-defined) of erroneous k-mers before the filtering process can be triggered:

$$\sum_{m=1}^{m_{low}} K_t(m) > S_{min} \tag{4.2}$$

Practically, every m-th read seen ($m = 10^6$ in our implementation), a filtering pass is triggered (i.e. putative erroneous k-mers are removed from the index) if 4.1 and 4.2 are satisfied. The complexity of one filtering pass is $O(n)$, where n is the number of distinct k-mers in the dataset.

Calculating the cut-off value During early filtering passes, a small fraction of correct k-mers is likely to contributes to the erroneous component. Hence, removing the entire component at each filtering pass is not a sensible choice. An incrementing value, defined as the *cut-off value*, is introduced to overcome this problem. Every k-mer of abundance lower than the cut-off value is removed. Formally, a threshold m_{solid} is defined as the number of occurrences below which a k-mer is considered a potential error. All k-mers over this threshold at the end of the indexing phase are said to be *solid* k-mers. Let $tReads$ and $nReads(t)$ be the total number of reads in the input file and the number of reads processed at time t respectively. The cut-off value $F(t)$ is calculated according to the following formula:

$$F(t) = \left\lfloor \frac{m_{solid} \cdot nReads(t)}{tReads} \right\rfloor$$

4.4.3 Paired reads indexing structure

Each sub-index is populated independently with a filtered set of references to reads, given a filtering function designed for *de novo* assembly. The *extension* of a k-mer in a read is defined as the suffix immediately following the k-mer (e.g. for a read $r = uwv$ where w is a k-mer and u, v are arbitrary strings, v is the extension of w in r). We introduce a notion of redundancy between extensions. Let (v_1, v_2) be two extensions of the same k-mer, without loss of generality assume that the length $|v_1|$ is shorter than $|v_2|$. Two extensions $v_{1,}v_2$ are said to be u-redundant if the Hamming distance between their prefixes of length $|v_1|$ is lower than u. Figure 4-6 illustrates redundancy between reads with respect to extensions of the same k-mer.

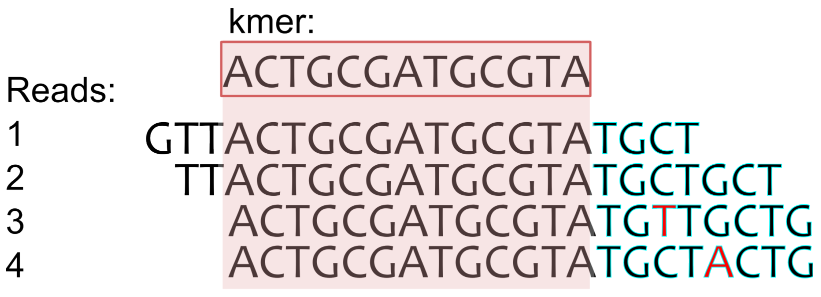

Figure 4-6: Set of pair-wise 1-redundant reads with respect to a k-mer (ACT-GCG(..)GTA). Red letters indicate sequencing errors.

The *representative read spectrum* with similarity threshold u, noted $RRS(k, u)$, is defined for a set of input reads as follows:

(i) For each distinct solid k-mer w present in the reads, associate to w a set S_w of reads containing the k-mer w.

(ii) in S_w, if there exists a group of reads for which the extensions of w are u-redundant, remove all but one read of the group from S_w. Specifically, a read in the group with the longest extension is kept, and ties are broken arbitrarily. Eventually, no pair of reads in S_w have u-redundant extensions of w.

Figure 4-7 shows an example of a representative reads spectrum. The reads sequences referenced by the RRS are stored separately from the RRS. Practically, both a read and its reverse-complement are indexed (doubling the memory usage).

In essence, this structure records a representative set of reads for each solid k-mer. Note that this indexing does not correct errors in read, but merely ignores errors in reads suffixes. Erroneous prefixes yield un-solid k-mers, hence these reads are not indexed in the structure. This property is well suited with Illumina reads as sequencing errors are known to mostly occur at read suffixes [49]. Provided the sequencing coverage is high, errors in suffixes can be corrected at a later stage during a consensus phase. This justifies the arbitrary removal of other reads having equally long u-redundant extensions. To maximize the effectiveness of the structure for assembly, sequencing reads should contain solid k-mers corresponding to every position in the genome. Hence, either a high sequencing coverage or a low error-rate is required. Both criteria are typically met with contemporary *de novo* Illumina sequencing experiments [49].

Since this structure stores references to read sequences, it can be immediately extended to index paired reads. Both mates of the pair are considered separately for insertion inside the structure. When any of the mates is inserted, the complete paired read is stored separately, and is referenced by the structure. References are explicitly made to the left or the right mate of the read.

The following paragraphs explains how traversal of the paired string graph is performed with this structure. The RRS acts as an "incomplete inverted index" for the reads, in the sense that given a k-mer, it returns a set of reads containing this k-mer. Specifically, in the paired string graph, out-neighbors of a read (i.e., other reads that overlap that read to the right) are retrieved by querying the RRS with each of the read k-mers, yielding candidate out-neighbors. Actual out-neighbors are retrieved by filtering candidates through exact overlap computation. In-neighbors (left overlap) are equivalent to out-neighbors of the read reverse-complement. Paired links are immediately retrieved by retrieving opposite mates from the paired read sequences.

4.4.4 Indexing results

We developed an implementation of the on-line k-mers filtering and the reads indexing algorithms. The implementation has been tested on two actual sequence datasets from *R. sphaeroides* (SRA reference SRR034530) and *N.crassa* (all libraries from [48]) sequenced using the Illumina technology. The *R. sphaeroides* dataset (dataset 1) contains 46 million reads of length 36 bp. The *N.crassa* dataset (dataset 2) contains 320 million reads of average length 32 bp. Benchmarks were run on a 64-bit 8-cores machine with 66 GB of memory. In this implementation, read sequences are stored in memory on each node as an array of 2-bit encoded sequences. In the case of multi-nodes computation, $\frac{n}{4}$ bytes are redundantly stored per node, where n is the number

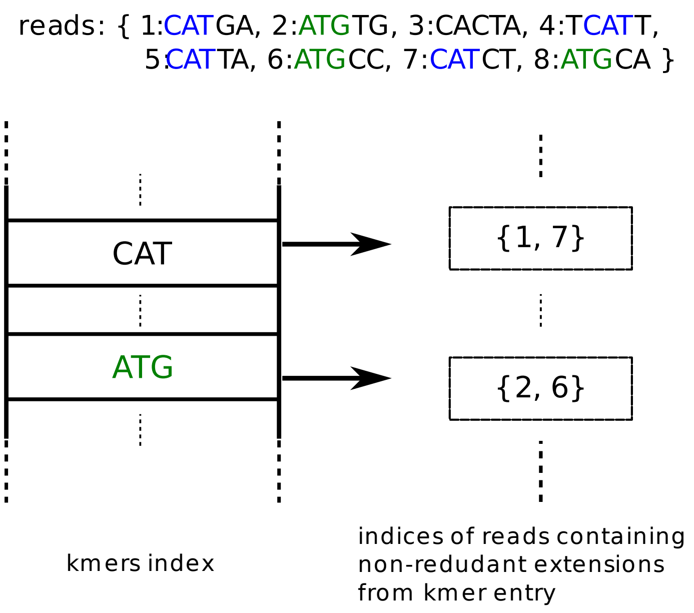
reads: { 1:CATGA, 2:ATGTG, 3:CACTA, 4:TCATT,
5:CATTA, 6:ATGCC, 7:CATCT, 8:ATGCA }
CAT
ATG
{1, 7}
{2, 6}
kmers index
indices of reads containing non-redudant extensions from kmer entry

Figure 4-7: The representative reads spectrum for a set of 8 reads with parameters $k = 3, u = 1$. Entries are solid k-mers from reads. Each k-mer is associated with a list of reads which extend the k-mer to the right. The extensions are filtered for u-redundancy. For instance, read 4 is not indexed in the CAT entry because the extension T is 1-redundant with respect to extension GA from read 1. In the CAT entry, read 5 could have been inserted instead of read 1 (arbitrary tie removal). Reverse complements of reads are also indexed, but are omitted in this figure.

of nucleotides in the reads. For the *R. sphaeroides* reads set, this amounts to 0.462 GB.

Online k-mer filtering results

We first examined the effect of on-line k-mers filtering on the first dataset. To this end, only the abundance count is retained for each k-mer. A comparison against the same k-mer counting procedure without filtering is made in Figure 4-8. It is important to note that, when entries corresponding to erroneous k-mers are removed from the hash table, the allocated memory is not freed but is instead made available for new entries. There are 144 M distinct k-mers in the dataset, only 4.5 M (3.1%) of which are correct (each sequencing error potentially induces up to $k-1$ new erroneous k-mers). On-line filtering enabled to keep the number of k-mers in the hash table under 23 M at any time. We verified that 4,544,973 solid k-mers are retrieved without filtering, compared to 4,464,256 (98.2%) solid k-mers with filtering (solid threshold $m_{solid} = 10$). The difference of 80,717 k-mers corresponds to premature filtering of k-mers that would be solid if given enough time before filtering. Hence, this procedure enabled to detect a very large amount of solid k-mers, yet it missed a small percentage of them.

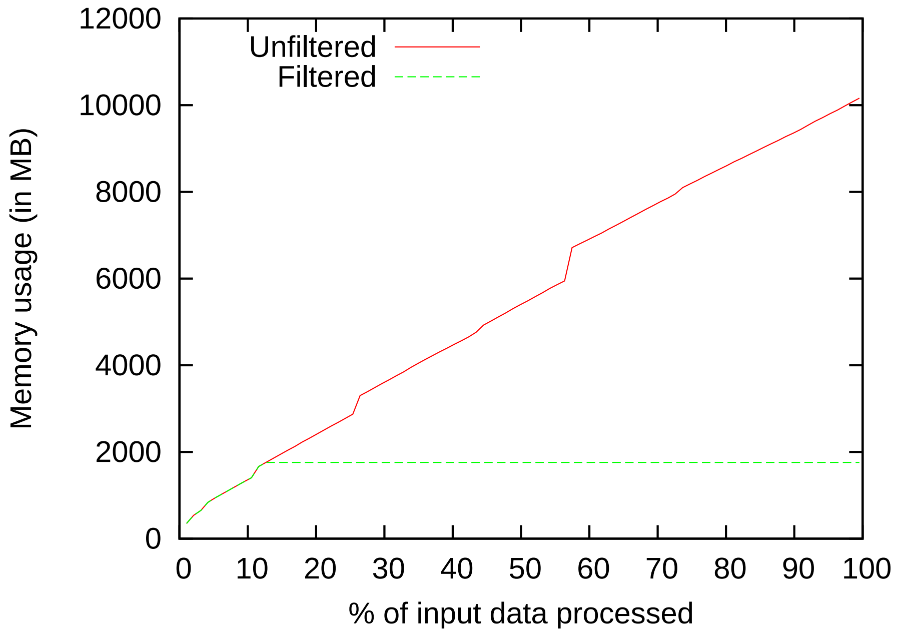

Figure 4-8: Memory usage during the k-mers counting procedure with on-line filtering, compared with the same procedure without filtering. Dataset 1 is processed with solidity threshold $m_{solid} = 10$, error-detection resolution $r = 10$, minimum amount of erroneous k-mers $S_{min} = 10^7$ and using 1 thread. The first filtering pass is triggered at 11.6% of the dataset. Sporadic jumps in memory consumption correspond to resize operations of the hash table. Figure taken from [9]

Parallel indexing speed-up

Then, we computed the full indexing time for an increasing number of cores (Figure 4-9). Note that a significant overhead occurs, which ultimately limits the benefits of parallel processing. This overhead is due to two factors: first, the complete set of reads is pre-loaded on each node, and this step cannot be parallelized. Second, the complete set of k-mers is processed by each thread. Specifically, for each k-mer, each thread computes a hash to determine whether it is designated to process the

k-mer or to discard it. As seen in Section 4.4.1, this enables each thread to insert only a fraction of k-mers in the table, therefore consuming a fraction of the total memory. The overhead is significant, partly because our implementation does not take advantage of the fact that k-mers are consecutive inside a read. We expect that a more optimized implementation would greatly reduce the overhead, as in principle the operations are not intensive. However, in the next chapter, we demonstrate that the assembly step does not suffer from such overhead. A near-linear speed-up will be achieved for the assembly phase.

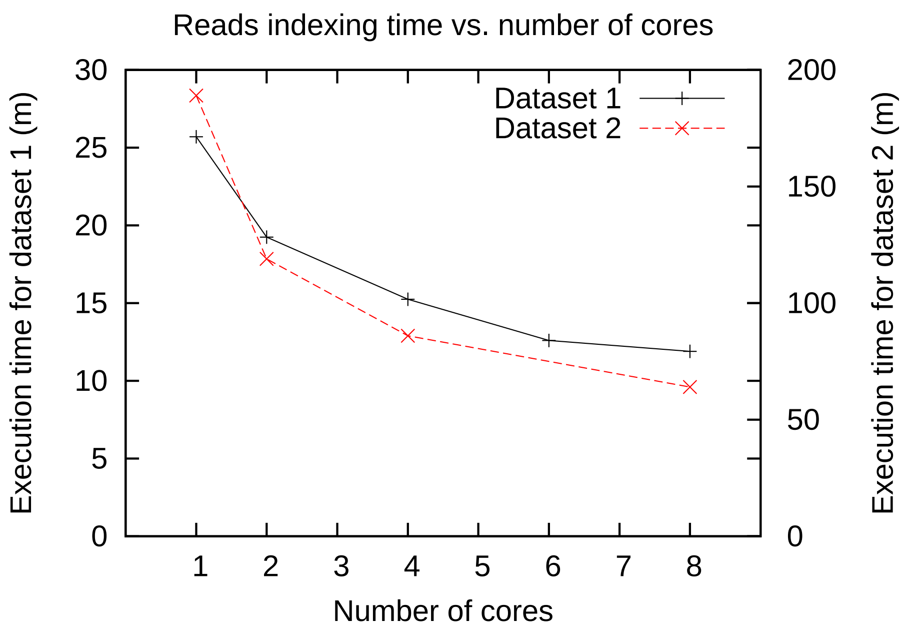

Figure 4-9: Execution time of our indexing implementation on datasets 1 and 2 using 1 node and 1 to 8 threads.

Memory usage

We compared memory usage of indexing procedures from other popular ultra-short reads assemblers with our implementation. The Velvet assembler [57] (version 1.1.03) and the SOAPdenovo assembler [31] (version 1.05) are based on de Bruijn graphs and use graph simplification heuristics. SOAPdenovo is specifically optimized for memory efficiency, it discards reads and pairing information in the initial graph structure. Our implementation uses spectrum parameter $s = 4$, $S_{min} = 10^6$, $m_{solid} = 10$ and $r = 0$

for both datasets. All the assemblers are executed with k-mer size of 21. Only the indexing phase of assemblers were run (`pregraph` for SOAPdenovo, `velveth` for Velvet). Our index stores essentially the same type of information (references to reads for each k-mer) than SOAPdenovo and Velvet, with some implementation differences.

Results are summarized in Table 4.1. The k-mers filtering step is essential in our method: complete indexing of Dataset 1 without k-mers filtering required 20.1 GB of memory. In terms of wall-clock time, these methods are comparable: for the largest dataset, SOAPdenovo and our prototype completed indexing in respectively 41 and 64 minutes using 6 threads (Figure 4-9 for our prototype, data not shown for SOAPdenovo, Velvet time was not measured but is presumably longer due to single threading). In summary, this indexing scheme significantly reduces the memory bottleneck for assembly, with minor impact on parallel indexing time.

	Dataset	Our prototype	Velvet	SOAPdenovo
Peak memory (GB)	1	2.7	7.7	3.9
	2	15.3	-	31.4

Table 4.1: Practical memory usage of indexing 46 M reads *from R. sphaeroides* (dataset 1) and 320 M reads from *N.crassa* (dataset 2) using Velvet, SOAPdenovo and our prototype. Velvet exceeded the memory limit (66 GB) on the second dataset.

4.4.5 Static k-mer index

The indexing techniques presented in this chapter, while being intrinsically parallel, can also be used to reduce memory usage of single-threaded greedy assemblers. Usually, the same data structure is used to construct the final reads index and then access it during assembly. This structure is typically a (dynamic, i.e. which supports insertions and deletion of elements) hash table. However, the greedy assembly phase does not require a dynamic structure. Hence, taking advantage of immutability, one can focus on designing a more compact representation of the reads index once it is fully constructed. For memory efficiency, sub-indexes can be simply constructed one at a time.

We realized that the idea of succinctly representing de Bruijn graphs [13], can be applied to more complex indexes, as long as keys are k-mers, such as ours. The hash table can be replaced by the union of an entropy-compressed rank/select dictionary of keys, and a rank-indexed array of values. Succinct rank/select data structures such as the sdarray [39] represent sparse sets in near-optimal memory usage. A bit array of length n with m entries set to 1 is represented by the sdarray using

$$m\lg(\frac{n}{m}) + 1.92m + o(m)$$

bits of memory, allowing $o(n)$ *rank* and *select* queries. The $rank(i)$ query returns the number of 1 in the array before position i. The $select(i)$ query returns the position of the i-th 1 in the array. A classical 2-bit transformation is used to convert a k-mer, where $k \leq 32$, into a 64 bits integer. The rank/select dictionary is used to represent the set of k-mers as a set of integers. A simple array containing fixed-length arrays stores representative reads.

To query this structure for a given k-mer corresponding to integer i, the membership of i in the dictionary is first tested by checking that

$$select(rank(i)) = i.$$

Then, the value at position $rank(i)$ in the array of values is the value associated to the query k-mer.

There is no pointer overhead in such read indexing structure. An actual benchmark indicates that the entire index of dataset 2 is represented in only 4.2 GB of memory with this method. This represents only 27% of the original hash table index size, without loss of information.

4.5 Discussion

In this chapter, we introduced two novel ingredients for efficiently assembling paired reads. First, we studied how to perform traversal of paired string graphs in the ideal case. The notion of non-branching path was introduced, which extends classical simple path to pairing information. Non-branching paths take full advantage of the information contained in paired string graphs. Such paths permits direct construction of scaffolds from reads. Then, the practical case of traversing a graph constructed from error-prone reads was examined. Two difficulties were identified: dealing with additional branching, and dealing with incomplete pairing information. To solve both problems, practical non-branching paths were defined, by incorporating classical assembly heuristics (for additional branching), and novel heuristics (for incomplete pairing). Practical non-branching paths provide a greedy scheme for traversing the paired string graph (this will be used in the next section). This contrasts with approaches which rely on graph simplifications to transform the string graph or the de Bruijn graph, to then output simple paths. A main practical advantage of our method is that the graph needs only to be stored statically. Also, a practical novelty is that constructing non-branching paths can be done in parallel.

The second ingredient is a parallel and memory-efficient indexing scheme. This scheme relies on two novel concepts: (i) early detection and filtering of erroneous k-mers, and (ii) associating reads to k-mers and filtering redundant reads. Multi-core and multi-nodes parallelism is achieved by observing that the set of k-mers can be partitioned. Hence, each thread constructs an independent part of the index. To detect and filter erroneous k-mers, the abundance histogram of k-mers is dynamically computed. Filtering is triggered whenever the histogram shows a sufficient separation between erroneous and correct k-mers. Redundant reads were identified by clustering reads containing the same k-mer, and removing those with nearly-identical extensions. This indexing scheme was implemented and compared with that of two popular assemblers. In one benchmark, the index required 30-50% less memory than the pregraph of SOAPdenovo, with comparable multi-threaded indexing time. By replacing the classical hash table with an optimized static index, the index uses 86% less memory.

Chapter 5

Monument assembler

The previous chapter covered two practical aspects of de novo *genome assembly using pairing information: graph traversal and memory-efficient indexing structure. These two ingredients are combined in the implementation of a new assembly software, the Monument assembler. We describe the structure of this implementation, including the details of some implementation choices and heuristics. Benchmark results are given for bacterial and fungus genomes, and Monument is compared to other assemblers in the context of a large-scale benchmark.*

Current genomic assemblers for high-throughput data suffer from two notable limitations: high memory footprint and lack of easily parallelizable algorithms. This is because modern assemblers rely on sequential algorithms that simplify a monolithic graph structure. These limitations can be traced back to the original assembly models. These models were designed to assemble a much lower number of reads, hence computational resources were not an issue.

For large genomes, constructing the classical string graph is a memory-intensive task. This issue also applies to paired string graphs, as they contain strictly more information. One solution to alleviate memory consumption would be to construct the graph using a compressed reads index. The FM-index [18] of the reads has been shown to significantly reduce construction memory usage of compressed string graphs [50].

This approach could be extended to include paired edges. To avoid memory overhead, paired edges could be computed dynamically from indexed paired reads. However, the construction run-time of the FM-index is high, and difficult to parallelize.

Thus, we propose a new strategy based on the indexing scheme presented in the previous Section. The key concept is: instead of constructing the whole graph, construct only a fraction of the graph at a time. Indeed, any part of the graph can be computed efficiently from a starting region: overlaps between sequences and pairing information are retrieved by querying the paired reads index. As described in the following, we take advantage of the localized graph traversal method to assemble the whole genome by constructing many local assemblies.

In this chapter, we bring together the concepts introduced in the previous chapters to implement this strategy. The Monument assembler is a new assembly software based on local construction of practical non-branching paths (Section 4.3.2), and paired reads indexing (Section 4.4). The software aims to be much more memory-efficient, and embarrassingly parallelized. Simultaneously, we target a quality of results comparable to that of other graph-based assemblers.

5.1 Pipeline

The pipeline of the assembler is given in Figure 5-1. Compared to a classical assembly pipeline (error correction, indexing, assembly into contigs, scaffolding, gap-filling), two main differences are noted.

No prior error-correction phase We claim that reads correction is not necessary, as the reads indexing module discards erroneous k-mers. A consensus phase later in assembly performs a majority vote which uses solid k-mers information to correct errors in reads.

No intermediate contigs Scaffolds are directly constructed from the reads, following a strategy described in the Section 4.3. We demonstrate that it is possible to construct scaffolds locally, without requiring a complete contigs assembly to be completed beforehand. This effectively allows targeted scaffolds assembly.

5.1.1 Indexing module

The indexing module of Monument follows the model developed in Section 4.4 on page 58. Two variants have been implemented:

1. k-mers pre-filtering then reads indexing: first, k-mers are inserted into the hash table with no references to reads. Second, erroneous k-mers are removed on the fly using the mechanism described in Section 4.4.2 on page 61. Third, references to paired reads are added.

2. Direct indexing: paired reads are directly inserted into the hash table (indexed by k-mer keys). Erroneous key/value pairs, i.e. low abundance k-mers with references to reads, are removed on the fly.

The first mode enables memory-efficient indexing at the cost of reading the input reads twice. The second mode processes the reads only once, at the cost of temporarily inserting erroneous entries. In both modes, the read indexing structure is constructed using the parallel partitioning strategy from Section 4.4 on page 58. The resulting index will be used in the assembly module.

5.1.2 Assembly module

The assembly module constructs all possible, (almost) non-overlapping, practical non-branching paths (Section 4.3.2 on page 55). To this end, it constructs a paired string graph using the index created in the previous module. Note that only a small subgraph of the complete paired string graph of the reads needs to be constructed for each path. Using this property, the module performs assembly locally, greedily, and in parallel.

Concretely, we propose a greedy, localized assembly procedure, that can be executed on an arbitrary number of threads. The procedure starts from an un-assembled

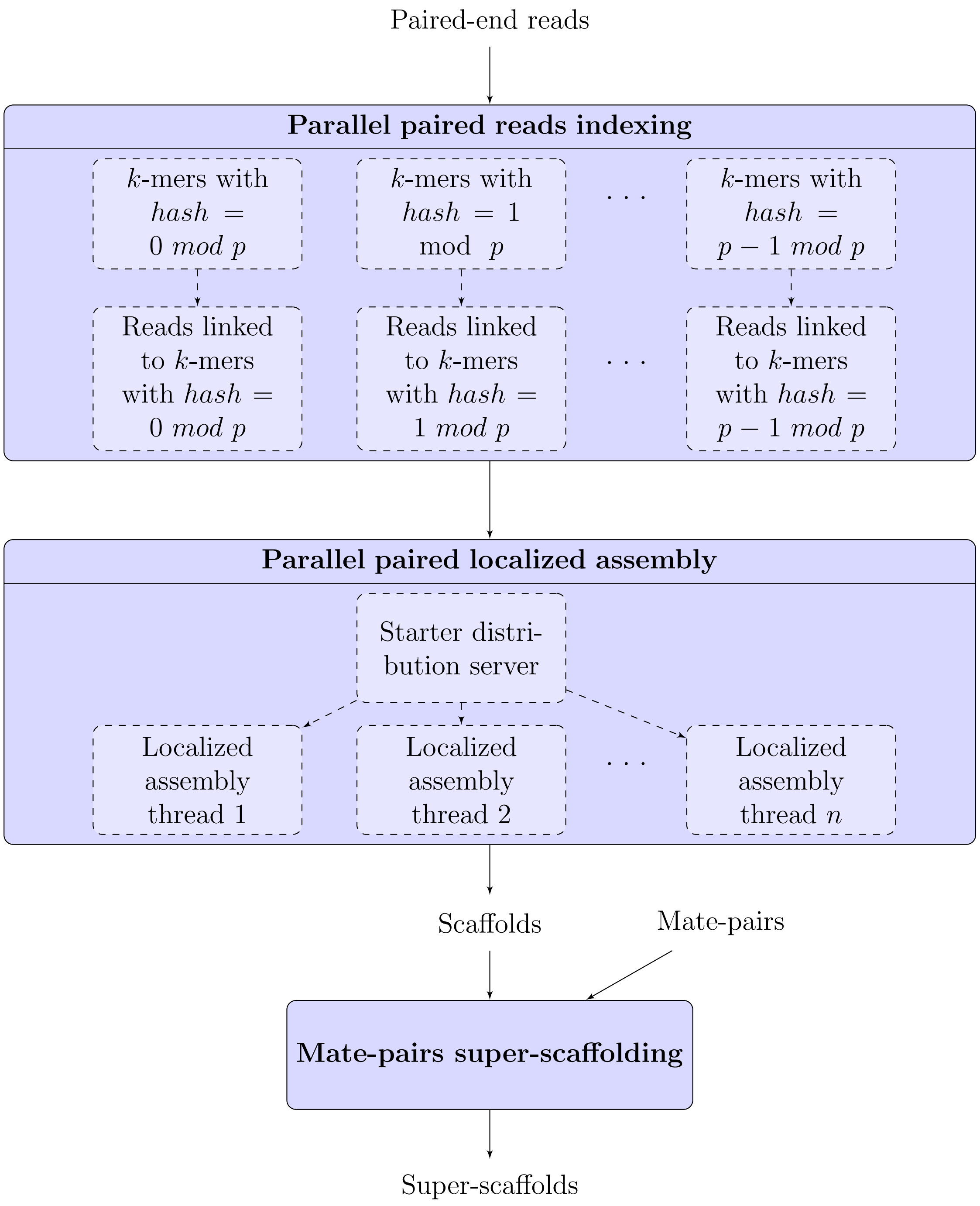
Paired-end reads
Parallel paired reads indexing
k-mers with $hash = 0 \ mod \ p$
k-mers with $hash = 1 \mod p$
. . .
k-mers with $hash = p-1 \ mod \ p$
Reads linked to k-mers with $hash = 0 \ mod \ p$
Reads linked to k-mers with $hash = 1 \ mod \ p$
. . .
Reads linked to k-mers with $hash = p-1 \ mod \ p$
Parallel paired localized assembly
Starter distribution server
Localized assembly thread 1
Localized assembly thread 2
. . .
Localized assembly thread n
Scaffolds
Mate-pairs
Mate-pairs super-scaffolding
Super-scaffolds

Figure 5-1: Modules of the Monument assembler

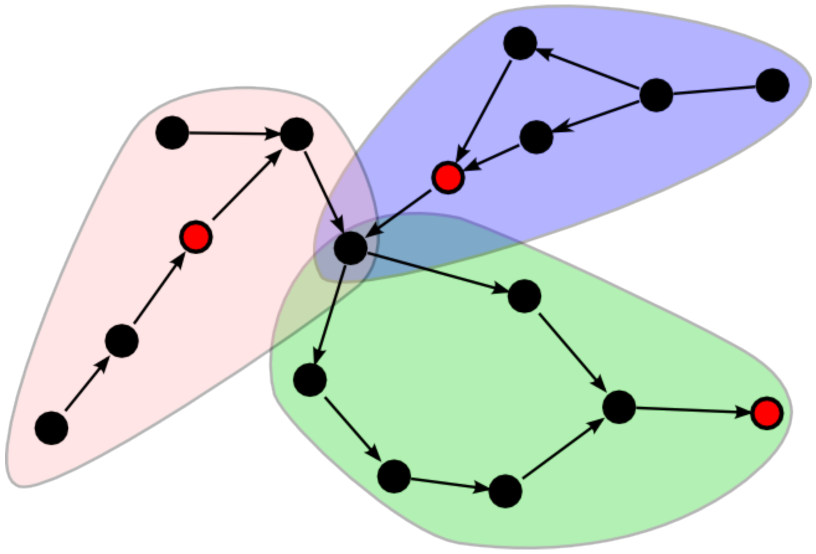

Figure 5-2: Sketch of assembly using localized sub-graph construction. Red nodes are starting regions. Each starting region initiates the construction of a sub-graph (clusters of different colors). This graph does not depict an actual assembly.

region (defined formally in Section 5.2.3). Instead of extending sequentially and stopping at the first ambiguity, it constructs a sub-graph of the paired string graph on the fly, in both directions, by following a practical non-branching path. Here, assembly is said to be localized, because the procedure only explores a sub-graph of the paired string graph. It is greedy, because the sub-graph is constructed by successive extensions.

This corresponds to splitting the complete paired string graph into almost-disjoint sub-graphs, each sub-graph corresponds to exactly one scaffold. Figure 5-2 shows an example of such split. This approach induces a memory overhead due to the parallel construction of sub-graphs. However, only a constant number of scaffolds are assembled in parallel at any given time. The assembly procedure marks k-mers to make sure that each k-mer appears only once in the global assembly. This condition can be violated when the same k-mer is assembled by two or more sub-graphs in parallel. Also, as shown in Figure 5-2, sub-graphs overlap at branching nodes. Such assembly redundancies can be filtered out in the resulting scaffolds, by recording redundant k-mers. For instance, a redundancy-filtering procedure is given in Chapter 6, Section 6.2.2.

To construct each sub-graph, overlaps between paired reads and pairing information need to be accessed efficiently. Hence, it is required that a complete index of the reads resides in memory. However, such index occupies less memory than a string graph. Provided that each worker can access the full index, embarrassingly parallel (i.e. independent) construction of scaffolds can be achieved.

While this is technically a greedy assembly algorithm, such approach overcomes the shortcomings of classical greedy algorithms. Except for Taipan [47], which does not support paired reads nor parallel assembly, most greedy assemblers do not construct a graph to solve local extension ambiguities [5, 1, 4]. As a consequence, greedy assemblers stop contigs extension at small biological variations or sequencing artifacts. Localized graph-based assembly using Postulate 2 (in Chapter 4) overcomes this problem by explicitly performing traversal of such structures.

5.2 Implementation of the assembly procedure

Algorithm 4 Monument assembly procedure

```
1: Start from an initial sequence $s^{(0)}$
2: t ← 0
3: **repeat**
4:     $s^{(t)'} \leftarrow$ extension of $s^{(t)}$ using an extension graph
5:     $s^{(t+1)} \leftarrow$ extension of $s^{(t)'}$ using a paired extension
6:     t $\leftarrow t + 1$
7: **until** the previous paired extension is successful
8: $s_g^{(t)} \leftarrow$ attempt to fill the gaps in $s^{(t)}$
9: **return** an assembled, gap-filled scaffold $s_g^{(t)}$
```

Algorithm 4 describes how the assembly procedure is implemented in Monument. In Step 4, a more efficient implementation of paired string graphs is used for regions where pairing information is not necessary. For instance, genomic regions where repeats are shorter than the reads can be assembled without pairing information. In such cases, we take advantage of the in-memory full reads index to construct a compact sub-graph: the *extension graph*. The extension graph yields a contig which cannot be extended using only overlap information.

Then, in Step 5, using pairing information, we perform a *paired extension* following the practical non-branching paths strategy (Section 4.3.2 on page 56). Implementation details of paired extensions are described later in this section. An overview of the method is given here: a paired string graph G is constructed from the contig extremity of length equal to the insert size. This contig extremity is a simple path p, with a central node n. If there exists a simple path p' of central node n', such that the paired link $n \dashrightarrow n'$ satisfies the non-branching condition, the contig can be extended as follows: the consensus sequence of p' and appended, along with a gap of suitable length, to the contig. This completes the paired extension.

To continue assembling, another extension graph is then constructed by taking this consensus as a starting region, and the whole process is repeated as long as paired extensions are possible.

5.2.1 Extension graphs

The extension graph can be seen as a compressed de Bruijn graph, where non-branching regions are compressed into a single node at construction time to save space. It is characterized as follows: nodes are arbitrarily long sequences (reflecting unambiguous contigs), and edges indicate exact $(k-1)$-overlaps between sequences. Extension graphs were first used in the targeted assembly approach of Mapsembler [40].

The construction of an extension graph always starts from a single node (a starting region). Each node is extended by the set of k-mers that $(k-1)$-overlap with it. Extensions stop when the breadth of the tree exceeds a maximal breadth b (experimentally set to 20), or when branching (tree breadth ≥ 2) is present during more than m consecutive depths (experimentally set to 1000). In other words, the graph is extended as long as Postulate 1 (Section 4.3.2 on page 57) is satisfied. In essence, a depth-first search is performed from the starting region, yielding a tree. Leaf nodes are extended as long as the graph is acyclic and of bounded breadth.

The procedure to construct the extension graph from a starting region is given in Algorithm 5. Several classical consensus steps are implemented to transform the extension graph into a contig.

Algorithm 5 Construction of an extension graph from a starting region

```
Require: Starting region $r$, maximal branching length $m$, maximal branching breadth $b$
Ensure: Extension graph $G$.
 1: procedure EXTENSION($G, E$)
 2:     $E' = \emptyset$
 3:     for node $e$ in $E$ do
 4:         $P \leftarrow$ { set of $k$-mers that perfectly $(k-1)$-overlap with $e$ }
 5:         if $P$ is a single sequence $p$ then
 6:             replace $e$ by the node labeled by $(e + p[k-1])$ in $G$
 7:             add $(e + p[k-1])$ to $E'$
 8:         else
 9:             for each sequence $p$ in $P$ do
10:                 add a node labeled by $p$ and an edge $(e, p)$ to $G$
11:                 add $p$ to $E'$
    return $G, E', l_b$

12: $G = (\{\text{r}\}, \emptyset)$                          ▷ Extension graph
13: $E = \{\text{r}\}$                                 ▷ Nodes to be extended
14: $l_b = 0$                                       ▷ Branching length
15: while $E \neq \emptyset, |E| \leq b, l_b \leq m$ do
16:     $G, E, l_b =$ EXTENSION($G, E$)
17:     if $|E| > 1$ then
18:         $l_b \leftarrow l_b + 1$
19:     else
20:         $l_b \leftarrow 0$
    return $G$.
```

5.2.2 Paired extensions

Without loss of generality, we consider the paired string graph of all the reads where one mate align to the right extremity of the contig. The sub-graph induced by the mates which align to this extremity is a simple path. Consider G, the sub-graph induced by the opposite mates. The paired branching condition (Section 4.3.2 on page 56) requires that G is also a simple path. In practice, G might have a slightly more complex structure than a simple path (see Figure 5-3 for an example). Heuristics had to be implemented to cope with additional overlap branching possibly present in G. For the sake of completeness, we present them here.

Instead of collapsing/removing branching, we adopt a more conservative approach which detects long enough simple paths (longer than $2i + 1$, where i is the maximal insert length deviation). Also, if G contains a cycle, paired extension is stopped. If G is cycle-free, a topological sort of G is performed to compute a partial ordering of the

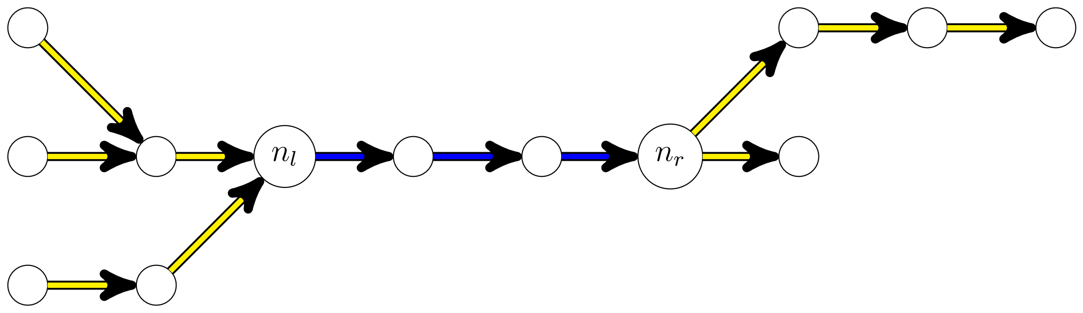

Figure 5-3: Illustration of simple path detection for paired extensions. The initial graph (taking the union of yellow and blue edges) is constructed from opposite mates of paired reads. A procedure analyzes the graph to detect where in-branching and out-branching end, i.e. at nodes n_l and n_r. If the simple path between n_l and n_r is long enough (blue edges), a paired extension is performed.

nodes. Then, a reverse breadth-first search is performed from one of the last ordered nodes of the graph. This permits to detect the last node where out-branching occurs (n_r in Figure 5-3). If no such node exists, one of the last ordered nodes is selected as n_r. From n_r, another breadth-first search enables to detect the first node where in-branching occurs n_l. If none exists, one of the first ordered nodes is selected as n_l. Eventually, the path p between nodes n_l and n_r is guaranteed to be a simple path, with nodes appearing consecutively in the topological sort (Figure 5-3). Hence, the paired extension is validated if p is long enough. This indicates that G contains a simple path that is a unique possible paired extension.

A paired extension is performed if p is long enough, i.e. longer than $2i + 1$, where i is the paired reads insert size deviation. The contig is extended with the consensus sequence of p. Finally, the gap size between the contig and the extension is estimated using the mean value of insert sizes from paired reads which align on both sides of the gap.

Paired-end data typically permits reliable paired extensions, because of the high sequencing depth. However, the pairing extension fails with mate-pair data, because such data has larger insert deviation and the lower coverage. To make use of mate-pairs, a subsequent scaffolding phase is necessary.

5.2.3 Starting region distribution and assembly termination

We describe the mechanisms used to distribute starting regions across threads to initiate a local assembly. To select an un-assembled region, the indexing structure is augmented with flags which indicate which k-mers have already been used in the assembly. Recall that the indexing structure is a key/value table where k-mers are keys, and values are filtered references to reads. An additional marker bit is appended to each value, to indicate whether a k-mer has already been assembled. Whenever a thread requests a new starting region, the following steps are performed on the server:

1. The first un-marked k-mer of the indexing structure is considered as a potential starting region.

2. A short extension graph is constructed from this k-mer. The graph should not contain any branching node. The motivation is to construct a short, simple path, yielding a short consensus sequence s.

3. If the length of s is below a minimum length (arbitrarily set to $2k + 1$), all the k-mers from this sequence are marked and the procedure restarts from Step 1.

4. s is returned as a new starting region, i.e. s is assembled by the thread.

5. When the thread completes the local assembly around s, all assembled k-mers are marked.

When should this mechanism stop, i.e. decide that no more regions need to be assembled? Typically, in string graph methods, all nodes of the graph (i.e., reads) need to be marked as explored. This is similar with de Bruijn graphs approaches, where k-mers are marked. Here, k-mers in the indexing structure are used to detect which regions are already assembled. The starting region distribution module decides that the assembly is completed when all the k-mer keys have been marked as explored.

5.2.4 Gap filling algorithm

To increase the size of contigs, a simple heuristic procedure can be used to fill scaffolds gaps. By construction, the length of scaffold gaps are shorter than the insert size. Inspired by related techniques [6, 7], the proposed procedure does not require the complete graph (as in Euler-USR [7]). Furthermore, reads localization (as in Allpaths [6]) is not a pre-requisite. To fill the gap between two reads l and r, the procedure starts from the string $t = l$ and repeatedly extends t to the right in a depth-first fashion, until r is reached. Right-most extensions of t are computed as follows. All the overlaps between the read-length suffix of t and the input reads are retrieved. Possible extensions are computed from the set of matches according to a voting mechanism to cope with sequencing errors. The search tree for t can possibly be large due to repeats in the original sequence. When an unambiguous sequence f to the left or to the right of the gap is long enough, reads localization is performed. In this context, reads localization selects paired reads where the other mate aligns to the sequence f, and extends only with these reads.

The following conditions are introduced to force early failure rather than long search time. During search, the string t cannot exceeds $(1 + a)$ times the expected size of the gap, where a is a constant that estimates the maximum uncertainty of actual gap size. A valid t that reaches r cannot be shorter than $(1 - a)$ times this size. Furthermore, the procedure fails after a constant number of iterations to prevent stalling on highly-branching regions.

Gap-filling is an important aspect of assembly, and the procedure above is only a naive solution. As it only performs a depth-first search, the procedure may fail to detect that two or more solutions exist for a given gap to fill. A breadth-first search would not suffer from this problem, however the search tree would become impractically large in complex cases. We wish to emphasis that gap-filling surely deserves a more in-depth treatment than what is presented here.

5.2.5 Dealing with sequencing errors

NGS assemblers adopt one or both of the following methods to cope with sequencing errors: either error-correction during a pre-assembly phase (e.g. Euler-SR assembler), or in-assembly graph simplification to remove vertices corresponding to erroneous reads (e.g. Velvet assembler). Both techniques are implemented in the SOAPdenovo assembler. Pre-assembly correction is computationally expensive as its running time is comparable to the whole assembly. In-assembly correction requires the construction of a large initial indexing structure containing extra erroneous entries. For large genomes, pre-assembly error correction using an external program becomes inevitable for classical assemblers because of memory constraints [31]. Short-read pre-assembly error correction algorithms typically construct a k-mer abundance histogram. This histogram is used to detect and correct low-frequency k-mers in reads.

The Monument assembler implements pre- and in- assembly error discarding, not correction. Low-abundance k-mers were discarded during indexing. This acts as a first filter towards discarding erroneous k-mers. Note that this step does not correct errors in reads, but merely discards references to reads. During assembly, remaining errors in reads are discarded by two mechanisms. The first mechanism is the voting procedure used in several assembly stages (e.g. paired extension and gap filling), which computes consensus sequences of reads sharing a common k-mer. The second mechanism is Postulate 2 of practical non-branching paths (Chapter 4). Even if an error is seen more times than the low-abundance threshold, it creates an erroneous graph portion, that can be discarded in a practical non-branching path.

5.3 Results

5.3.1 Assembly metrics

The quality of an assembly cannot be measured by a single metric. Many metrics and variations thereof have been introduced. We give a canonical sample of metrics here.

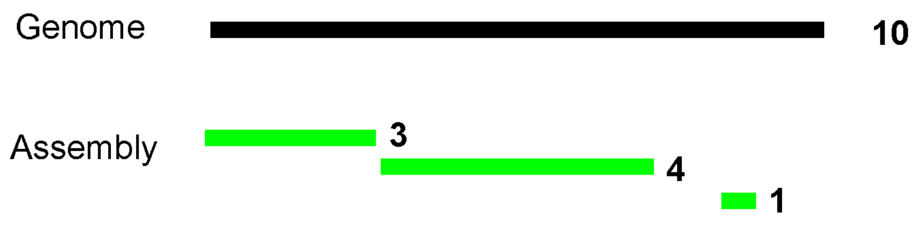

Figure 5-4: Toy example for computing the N50 and NG50 metrics. A genome (black line) is represented along with an assembly (green lines, representing contigs or scaffolds). The number next to the each line specifies a fictional sequence size. The N50 of this assembly is 4, and NG50 is 3.

The N50 metric is the scaffold/contig length at which you have covered 50% of the total assembly length. In other words, the N50 metric measures the length of the smallest element of the set of largest scaffolds (resp. contigs) which cover at least 50% of the assembly. Alternatively, the NG50 metric is the scaffold/contig length at which you have covered 50% of the total genome length (assuming it is known). Figure 5-4 is a fictional assembly of contigs lengths $(3, 4, 1)$ with respect to a genome of length 10. In this case, the N50 is 4 and NG50 is 3.

We derive formulations of the coverage and accuracy metrics from methods proposed by the authors of Allpaths [6]. Assembled sequences (contigs or scaffolds) are divided into chunks of size less than 10kb. Each chunk is considered to be valid if it aligns with more than 99% identity to the reference genome (alignment with undetermined nucleotides are considered valid). The global accuracy of the assembly is the ratio of valid chunks over total chunks. Similarly, the coverage is deduced as the ratio of reference genome bases covered by at least one aligned chunk. In Figure 5-4, the coverage of the assembly is 0.8.

5.3.2 Bacterial assembly results with simulated variants

We evaluate localized paired assembly using practical non-branching paths on a small (bacterial) dataset. Two short reads assemblers based on de Bruijn graphs are compared with Monument. The Velvet assembler (version 1.1.03) uses graph simplification heuristics [57]. The Ray assembler (version 1.3.0) implements a greedy traversal strategy [4]. The assemblers were run with default parameters and $k = 23$. By setting a similar k-mer size, all assemblers, including ours, virtually explore the same de Bruijn graph.

For Monument, the maximal graph depth d for genomic variants is set such that any path has genomic length less than $10 + k$. The insert size deviation i is set to half the value of the insert size, which is a very conservative deviation with respect to actual paired-end data.

We first compared assemblers on experimental Illumina short paired reads from *E. coli* (SRA SRX000429). This dataset (Dataset 1) contains 10 million paired reads of length 36 bp and insert size 200 bp. We then investigated the ability of our method to assemble diploid genomes. To this end, we simulated 3 million paired reads of a diploid genome based on the *E. coli* sequence (Dataset 2). The `wgsim` paired reads simulator was used with default parameters [30], producing 75 bp reads (500 bp inserts) with simulated sequencing errors.

Assembly results for the two datasets are shown in Table 5.1. For the empirical dataset, Monument obtains the best scaffold N50 value and the second best contig N50 value (second to Ray). For the simulated dataset with variants, Monument outperforms both methods in terms of N50 values. The genome coverages of the three assemblies are almost equivalent, for both the empirical (96.4%-97.4%) and the simulated datasets (87.9%-91.0%). In terms of accuracy, Monument produced the most inaccuracies in the experimental dataset. We noted that a significant portion of inaccurate chunks from the Monument assembly are caused by a minor effect: gap lengths between scaffolds were mis-estimated. In general, all the assemblies are of high quality: no mis-joins between two distant regions were detected in any assembly.

To understand why Ray has difficulties assembling the second dataset, we simulated a third dataset of reads, similar to Dataset 2 but without variants. This time, Ray obtains a scaffold N50 of 89.4 Kbp and largest scaffold of length 268.5 Kbp. This experiment confirms that mechanisms for variants traversal, such as practical non-branching paths, are a key requirement for greedy assemblers.

We recorded execution time and memory usage during indexing and assembly of

Dataset	Software	Contig N50 (Kbp)	Scaffold N50 (Kbp)	Longest scaffold (Kbp)	Coverage (%)	Accuracy (%)
Experimental (1)	**Monument**	38.0	101.8	236.0	96.4	96.7
	Velvet	26.3	95.3	267.9	96.9	99.1
	Ray	69.5	87.3	174.4	97.4	98.4
Simulated with variants (2)	**Monument**	113.3	134.1	340.5	91.0	95.0
	Velvet	30.8	132.6	327.2	87.9	92.3
	Ray	10.2	10.2	41.2	89.2	100.0

Table 5.1: Quality of the assemblies of simulated and experimental paired-end reads from *E.coli* using Velvet, Ray and Monument.

the experimental dataset. The size of the paired reads index is 0.4 GB and peak memory usage during assembly is 0.6 GB. Velvet and Ray have peak memory usage of 2.4 GB and 3.2 GB respectively. However, to reduce memory usage, Ray is the only assembler in this benchmark that has the possibility to distribute its index structure on a cluster. Using 6 threads, our implementation completed the assembly in 7 minutes, Velvet in 8 minutes and Ray in 16 minutes.

Our implementation can also assemble a scaffold around a specified genomic region, i.e. perform targeted assembly. This is of particular interest as targeted assembly methods (such as Mapsembler [40]) only produce contigs. Targeted assembly with Monument is also very fast: one scaffold is assembled in a few seconds. However, contrary to targeted assemblers, Monument requires the complete reads index to reside in memory.

5.3.3 Fungus assembly results, parallel speed-up measurements

We now assemble a genome that is one order of magnitude larger than the previous bacteria. The main goal of this experiment is to demonstrate how Monument performs localized paired assembly in a distributed fashion.

The size of the *N. crassa* genome is 39 Mbp. Two Illumina sequence libraries are publicly available [48], of respective insert lengths 200 bp (paired-end) and 3600 bp (mate-pairs), representing a 123x coverage. The paired-end library was error-corrected using SOAPdenovo corrector. The parallel indexing performance for this dataset was measured in the previous chapter (Section 4.4.4 on page 65). We performed localized paired assembly with Monument using the paired-end library only; the mate-pairs library is only used during a final scaffolding phase.

Figure 5-5 shows the time taken by the assembly phase of our implementation, with respect to the number of cores. An initial, one-time overhead (green area) is due to an imperfect index serialization procedure. Specifically, instead of fully serializing the static hash after indexing, the hash table is reconstructed from scratch (using a single thread) at each execution of the software. Using lower-level serialization could essentially remove this overhead. Hence, the serialization overhead is disregarded in the following analysis. Figure 5-5 shows that the speed-up of the assembly phase is near-linear up until a large number of distributed cores. Our implementation, using the largest number of cores (34), which were distributed on 3 nodes, achieves a 23x speedup (34x is the theoretical maximum). On this setup, after indexing, the *N. crassa* genome is assembled in 3.3 minutes. This represents more than 10 Mbp assembled per minute. In contrast, the reported assembly time for Allpaths 2 is 86.6 hours (using 16 cores). SOAPdenovo performs the assembly steps in 4.5 minutes (using 16 cores). Interpolating Figure 5-5 to 16 cores, Monument would have performed the assembly in 5 minutes. Allpaths and SOAPdenovo could not be evaluated on 34 cores, because they do not support distributed assembly.

To achieve this level of parallelism, we reduced the communication overhead between threads and the starting region distribution mechanism to a strict minimum. Specifically, the distribution server pre-loads several starting regions, to avoid latency when a thread requests a new region. Also, the redundancy of assembled sequences is filtered by the starting region server, upon receiving a fully assembled scaffold. This could have been done during assembly on each thread. However, doing so required threads to communicate with the server, to know whether a region was already

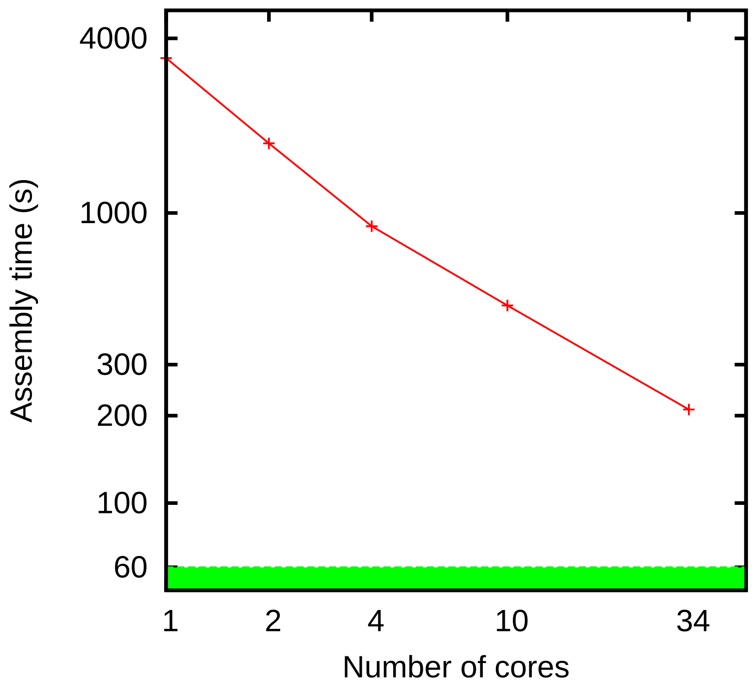

Figure 5-5: Parallel speed-up measurement, assembly of the *N. crassa* genome. The red curve indicates the wall-clock time taken by the assembly phase of Monument (localized paired assembly). Note that the assembly on last measurement (34 cores) was distributed on three nodes. The green area indicates the one-time, single-threaded overhead to load the index on each node.

assembled. In practice, this degraded parallel performance.

The assembly produced by Monument is compared with two other assemblies (from Allpaths 2 and SOAPdenovo) in terms of quality metrics. Table 5.2 reports the scaffold N50, coverage and accuracy of the assemblies. SOAPdenovo had problems using the mate-pairs library: including mate-pairs resulted in a lower N50, hence we report an assembly using paired-end reads only. Our resulting assembly, after incorporating mate-pairs, obtains the highest N50 metric by a wide margin. Monument dedicated scaffolder (manuscript in preparation) enabled a significant N50 improvement. All assemblies obtain comparable coverages (89-92 %), SOAPdenovo has 2.7 points more coverage than Allpaths and Monument. Allpaths obtains the highest accuracy of the three methods. Pre/post-scaffolding results suggest that 3.6 % of inaccuracies in Monument are caused by the scaffolding step. We verified that the origin of these inaccuracies is a mis-estimation of the number of N's inside gaps, due to the large deviation of mate-pairs. These inaccuracies could be easily fixed in SOAPdenovo and our scaffolder, by designing a more precise gap estimation formula.

Software	Scaffold N50 (kbp)	Coverage (%)	Accuracy (%)
Allpaths 2 (reported in [48])	58	89.5	97.8
SOAPdenovo	71	92.2	93.2
Monument, pre-scaffolding	28	89.5	96.5
Monument, post-scaffolding	292	89.4	92.9

Table 5.2: Quality results for the assemblies of the *N. crassa* genome using SOAPdenovo, Allpaths 2, and Monument. SOAPdenovo was only able to assemble the paired-end library. Monument results are reported before and after the scaffolding phase, which used the mate-pairs library. Note that since the reference genome is incomplete, the accuracy reported by Allpaths is possibly close to the (unknown) upper bound.

5.3.4 Assembly benchmarks

Assemblathon 1 & 2, dnGASP

The Symbiose team (R. Chikhi, G. Chapuis, D. Naquin, N. Maillet and D. Lavenier) participated to several assembly benchmarks using the Monument software.

The Assemblathon 1 was an international collaborative effort to evaluate state-of-the-art methods of *de novo* assembly as of early 2011. Simulated Illumina data from an completely artificial genome ($\approx$ 100 Mbp) was made publicly available. The artificial genome was not known to the participants. A distant genome from the artificial genome was produced, however it has not been used by any participant. The benchmark received submissions from 17 different groups which used either their own software, and/or publicly available tools. The metrics used to measure the "best" assembly were not known in advance. However, it was implicit that the classical metrics such as N50, genome coverage, short-range (indels) and long-range accuracy (misassemblies) would be measured.

dnGASP (http://cnag.bsc.es/) was a similar effort which took place simultaneously to the Assemblathon 1. These two events had almost identical requirements: perform *de novo* assembly of an artificial genome using simulated data. The main difference with the Assemblathon 1 is that the artificial genome was one order of magnitude larger ($\approx$ 1.8 Gbp). Due to the difficulty of assembling such a large genome, dnGASP received less submissions (8 groups participated).

The Assemblathon 2 took place six months after Assemblathon 1. The goal was to assemble three mammalian-scale genomes from actual sequencing data. The species are: *Maylandia zebra* ($\sim$ 1 Gbp genome), *Boa constrictor constrictor* ($\sim$ 1.4 Gbp genome), *Melopsittacus undulatus* ($\sim$ 1.2 Gbp genome). The benchmark received 43 entries (16 for the fish genome, 15 for the parrot, and 12 for the snake) from 21 groups.

Results for the dnGASP and Assemblathon 2 benchmarks were unpublished at the time of writing. The Assemblathon 1 consortium has published an extensive analysis of the quality of assemblies [16]. Perhaps less attention has been given to the computational resources needed to run the assemblers. In the next section, we describe the pipeline used with the Monument assembler to participate in these benchmarks, and present a selection of Assemblathon 1 results, including an unpublished computational resources summary (compiled here from informal reports of each participant).

Pipeline and results

Pipeline The assembly pipeline used for these benchmarks is described. The indexing module of Monument was used to perform k-mers pre-filtering then indexing using only the paired reads. For Assemblathon 1 and dnGASP, an earlier assembly module, without extension graphs, was implemented. The SSPACE [3] scaffolder was used to re-scaffold the results of Monument paired-end assembly, using mate-pairs. For Assemblathon 2, extension graphs were implemented and a new scaffolder (SuperScaffolder, unpublished) was developed. The SOAPdenovo GapCloser program was used to close gaps in our scaffolds [31].

Assemblathon 1 Figure 5-6 shows an overview of the NG50 quality metrics computed for assemblies scoring the best scaffold NG50 for each assembler. The purpose of this /figure is to show the absolute best scaffold NG50 that can be produced by each assembler. Note that in the Assemblathon 1 article, different assemblies were selected for specific evaluation, according to more criteria than just scaffold NG50. Figure 5-7 reports the typical running time and memory usage for each assembly pipeline, from indications given by each participant. This information is inaccurate, as hardware resources can vary greatly between groups, this is not a fair computational resources benchmark. However, it gives an indication of the computational resources needed to run each assembler.

Figure 5-6 shows that the compared methods produced assemblies that are very different. Several factors contribute to these differences. First, some groups performed undisclosed pre-assembly data processing, e.g. read correction and trimming. Read correction is still an active research area, and no method produces optimal results. Second, the contig N50 differences can be explained by whether a group performed gap-filling on the scaffolds or not. Third, the scaffold N50 can be explained by (i) the quality of initial contigs and (ii) differences between scaffolding heuristics. Fourth, groups probably optimized their assembly according to different metrics, i.e. choosing an accurate assembly over a contiguous one.

Several other quality metrics were computed in the Assemblathon 1 article. Assembly errors and coverage are reported here succinctly, using data from Tables 4 and 5 of the Assemblathon 1 article. Note that in the previous paragraph one assembly was selected per software, based on the best scaffold NG50 length. Here, accuracy and coverage results are reported for one assembly per group, selected by best Assemblathon 1 metrics overall score. However in practice, accuracy and coverage do not vary significantly between assemblies produced by the same assembler. Table 5.3 shows the number of inter- and intra- chromosomal mis-joins in selected assemblies, as well as insertion-deletions errors. Figure 5-8 shows the haplotype and genic coverages of selected assemblies.

In summary, our pipeline performed well in terms of scaffold NG50, structural errors, wall-clock time and memory usage, and had major weaknesses in terms of contig NG50 and coverage. A possible explanation for this is the choice of relatively low k-mer size (32) compared to other groups (other k-mer sizes were not always disclosed, but some participants reported using values of k ranging from 64 to 100). Another point is the lack of advanced gap-filling (which is the step aimed at improving contig N50 by filling gaps in scaffolds).

Preliminary results for dnGASP A similar pipeline to that of Assemblathon 1 was used to participate in the dnGASP benchmark. Preliminary results for our pipeline, from dnGASP organizers, are of similar nature to that of Assemblathon 1. Specifically, our assembly has one of the lowest scores in terms of the contig N50 metric, whereas it has the second best score in terms of the scaffold N50 metric. In terms of computational requirements, our pipeline completed the assembly using 78 Gb of memory, within a day, on a 6 nodes cluster (70 cores).

Preliminary results for Assemblathon 2 Preliminary results from the Assemblathon 2 indicate that our improved pipeline produced assemblies that are competitive with top assemblers in terms of results quality. Specifically, our assemblies of the *M. zebra* and *B. constrictor* genomes consistently rank 2^{nd} in terms of contig N50 and 6^{th} in terms of scaffold N50. Our improved pipeline shows a significant increase on the contig N50 metric compared to Assemblathon 1, possibly due to the combination of extension graphs and usage of the GapCloser program, which were not present in our Assemblathon 1 entry. With respect to the scaffold N50 metric, the gap between top assemblies and our assemblies is no longer as large ($\approx$ 0.6 order of magnitude, compared to $\approx$ 0.8 order of magnitude in Assemblathon 1). A weakness of our pipeline is that heterozygous repeated regions were assembled into distinct contigs, instead of classically collapsing them into a single contig. This was an error in setting assembly parameters. At the time of writing, the accuracy of each assembly has not yet been evaluated.

	Intra-chr mis-joins	Inter-chr mis-joins	**sum of mis-joins**	insertions	deletions	indels	insertions at ends
DOEJGI (Meraculous)	21	160	**181**	55	108	40	72
WTSI-S (SGA)	6	191	**197**	56	76	19	127
Broad (ALLPATHS)	75	161	**236**	524	379	9	96
IRISA (Monument)	147	203	**350**	925	1593	116	3375
BCCGSC (ABySS)	351	285	**636**	255	233	102	1641
BGI (SOAPdenovo)	368	288	**656**	355	639	98	130
CSHL (Celera)	396	337	**733**	417	3287	223	486
CRACS (ABySS)	687	303	**990**	198	121	51	306
EBI (SGA)	458	563	**1021**	127	547	53	307
RHUL (OligoZip)	691	349	**1040**	172	264	26	1049
IoBUGA (SOAPdenovo)	919	330	**1249**	1663	2933	356	109
GACWT (Cortex)	757	730	**1487**	905	1292	216	4722
CIUoC (Kiki)	1205	684	**1889**	1189	2026	65	6113
ASTR (PE-Assembler)	2065	200	**2265**	109	227	73	144
WTSI-P (Phusion2)	1940	449	**2389**	1851	289	87	279
UCSF (Price)	2731	2396	**5127**	5908	6223	1018	6711

Table 5.3: Assembly errors for Assemblathon 1 groups (taken from [16]). For each group, the assembly which maximizes an overall score of quality metrics is selected. Groups are sorted according to mis-joins errors, instead of the total sum of errors, as mis-joins errors are arguably the most detrimental type of errors in an assembly.

5.3.5 Discussion

In summary, a new *de novo* assembly software, Monument, is developed using paired string graphs. The novelty of this software is threefold: (i) it is able to perform localized assembly of scaffolds, (ii) its memory footprint is small and (iii) the assembly phase is embarrassingly parallel.

Localized scaffolds assembly Prior to this work, scaffolds were constructed from an ordering of contigs, requiring a complete assembly of contigs to be known beforehand. We show that it is possible to assemble scaffolds locally around a genomic region by following non-branching paths greedily. This approach allows to design the first localized assembly algorithm which directly constructs scaffolds from reads. Benchmark results on a bacterial dataset indicate that localized scaffolds assembly yields longer scaffolds than two popular short reads assemblers. On this dataset, the greedy scaffold traversal appears to obtain comparable results to scaffolding algorithms based on a complete contigs graph. We conjecture that scaffolders implemented in these assemblers do not take full advantage of the whole contigs graph.

Lower memory with similar quality Our method does not require a large graph to be stored in memory. A small graph is constructed for each scaffold. An indexing structure similar to that of greedy assemblers is used to construct each graph. However, compared to other greedy approaches, the graph approach takes into account biological variants. Hence, it does not suffer from degraded contiguity due to genomic variants.

Parallelism Additionally, this assembly algorithm is embarrassingly parallel in nature. A starting region distribution server enables scaffolds to be constructed independently by many threads. By assembling a 40 Mbp genome, we observed near-linear parallel speed-up to up to 34 cores (3 nodes). This is the first implementation which can assemble a genome in parallel with very little synchronization overhead.

Perspectives Two lines of improvement should be considered: (1) gap-closing in scaffolds is a key step for obtaining long contigs. Most complex repeats were not resolved by our simple path-finding procedure, hence a more elaborate algorithm is needed. (2) Incorporating mate-pairs with long inserts in genomic graphs is still an unaddressed challenge in the literature. These reads are produced with higher insert size variability and lower coverage than paired-end reads. Mate-pairs cannot be used in our current framework, because Postulate 1 (absence paired branching) almost never holds for such data. An immediate solution would be to perform re-scaffolding of scaffolds using mate-pairs links.

As short read sequencing is progressively shifting towards longer reads (over 100 bp), the landscape of assembly software has to adapt to high-coverage, longer reads. Specifically, de Bruijn graph implementations appear to be unable to assemble long reads (i.e. from the 454 technology) with quality comparable to string graph implementations. In contrast, string graph-based methods are limited to assembling low-volume datasets because of memory constraints. We believe that our methodology of local string graph construction will lead to software able to assemble both short and long reads at any coverage without sacrificing running time or results quality.

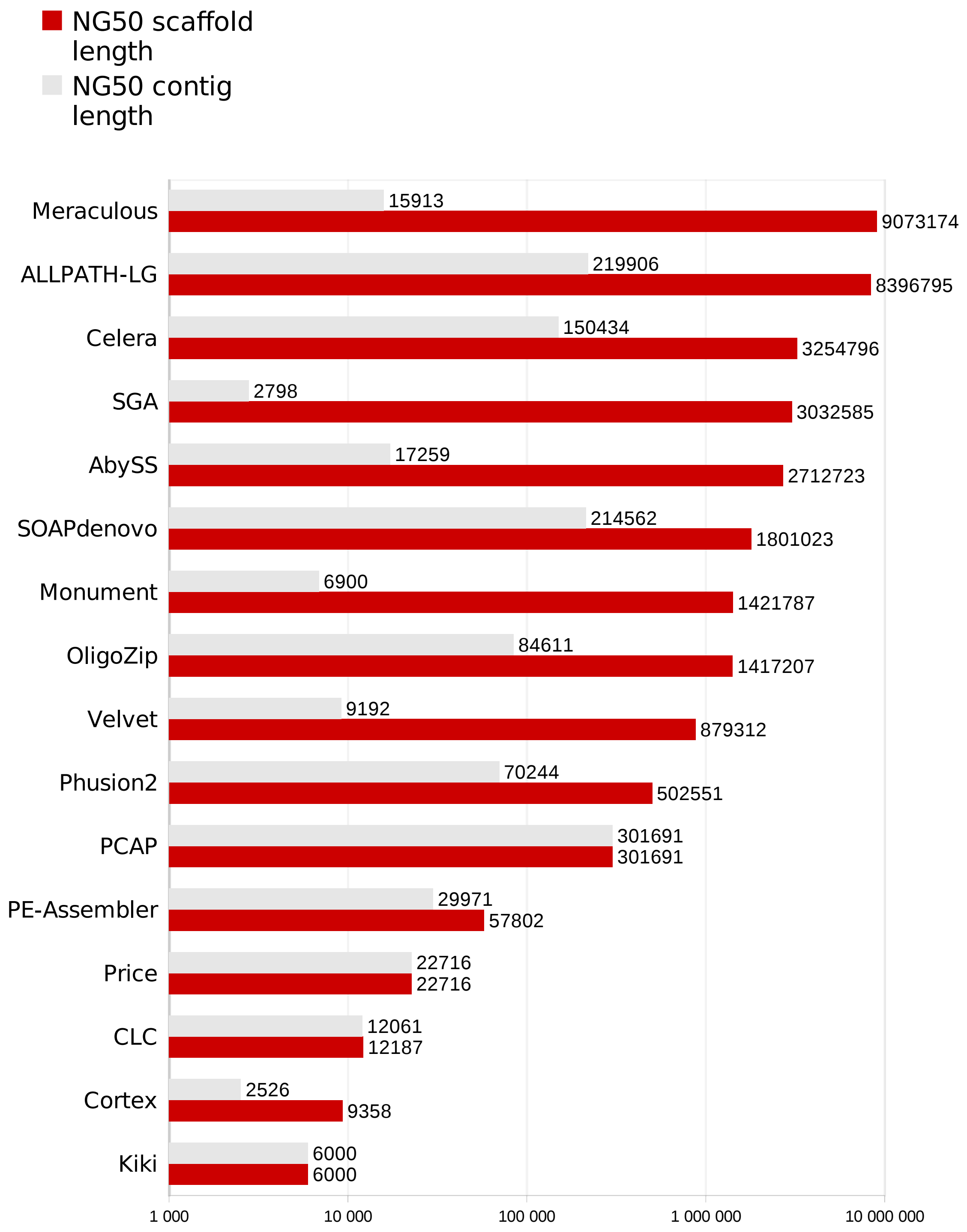

Figure 5-6: Assemblathon 1 quality results. For each assembly software, the best assembly with scaffold NG50 is selected. Assemblies are sorted by longest scaffold NG50 length. The corresponding contig NG50 length is shown. Assemblies computed by Assemblathon organizers are not shown.

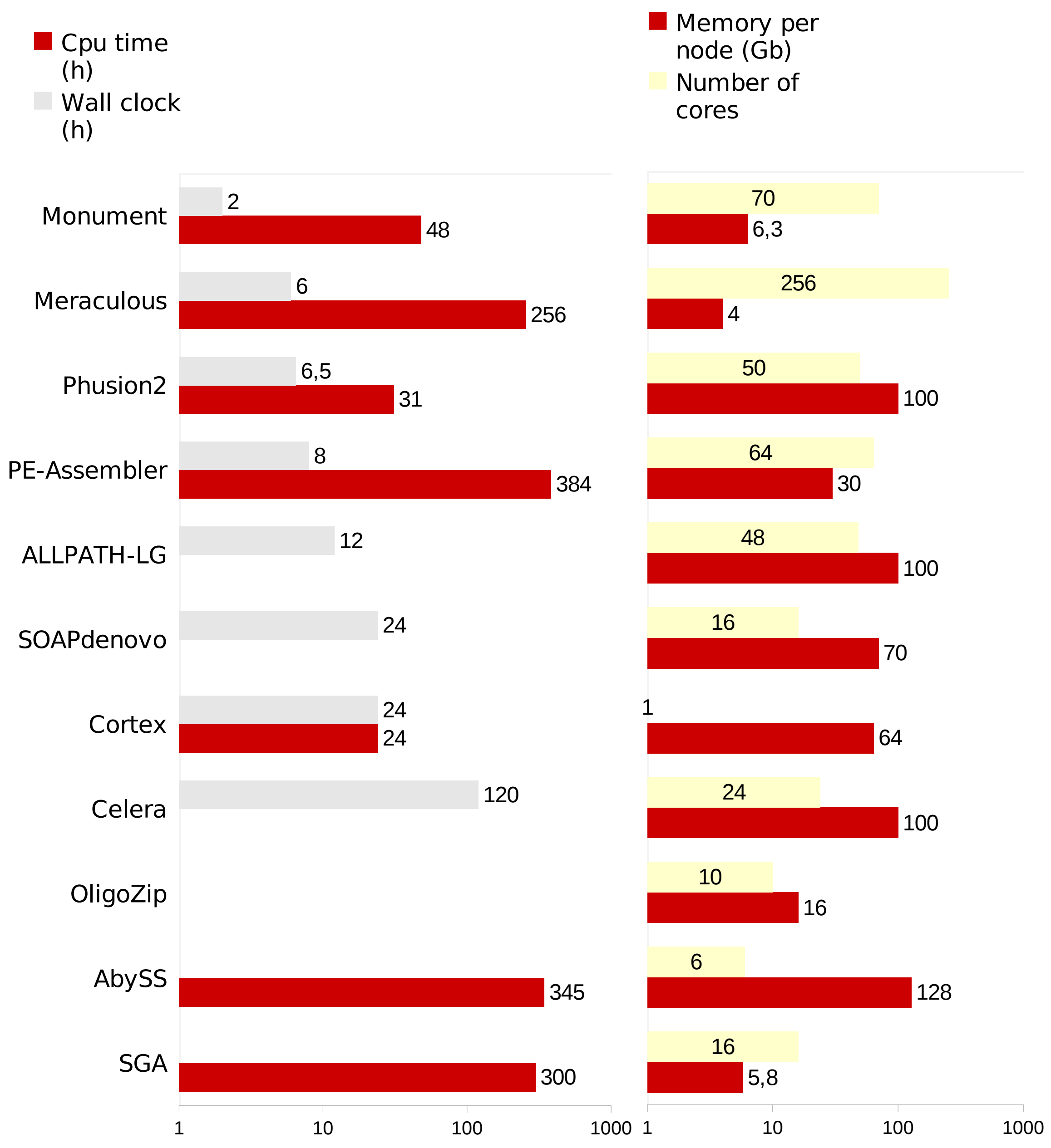

Figure 5-7: Assemblathon 1 resource usage, one per assembly software. This figure was compiled using information given by participants, and may contain inaccurate information.

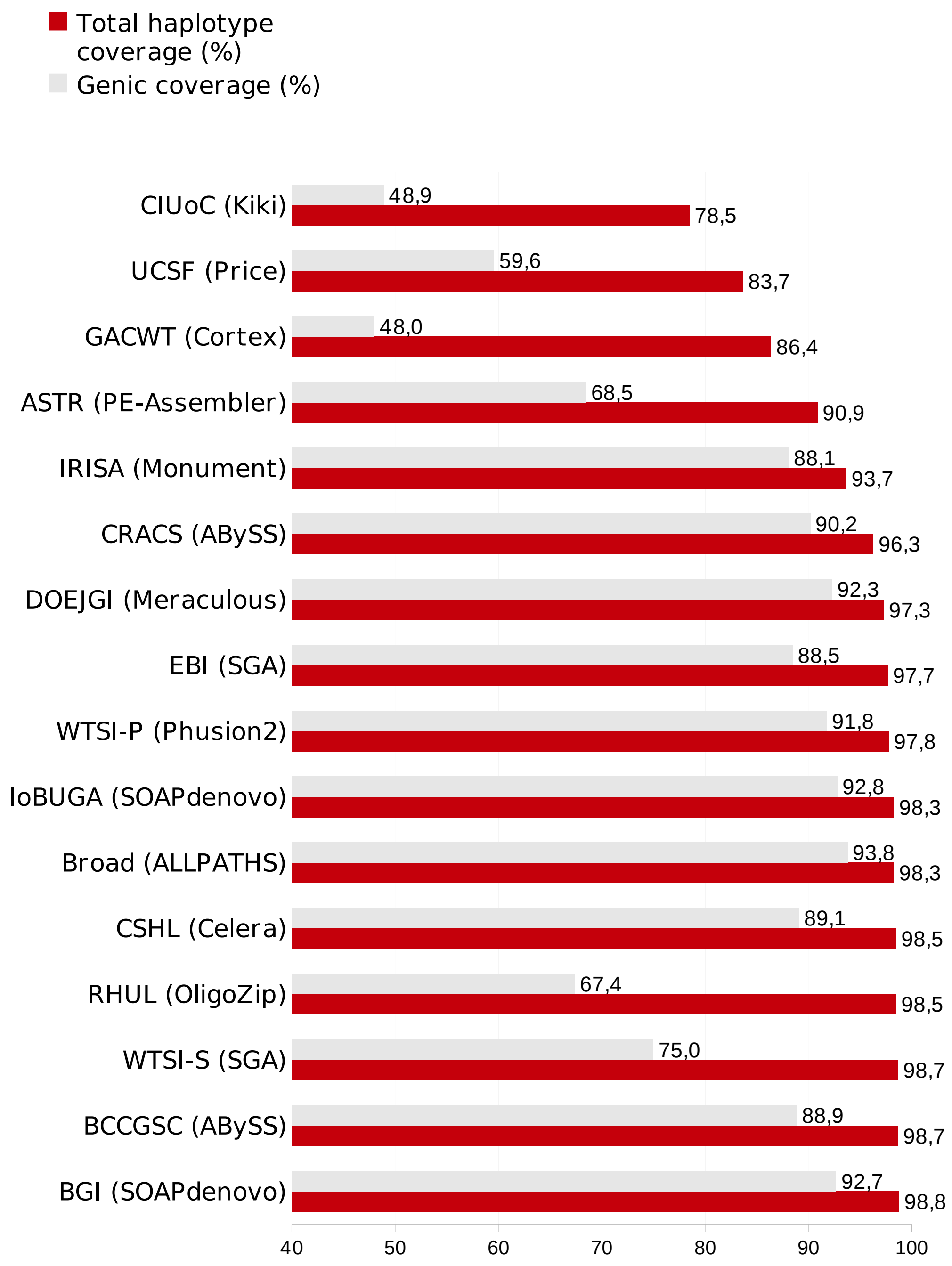

Figure 5-8: Haplotype coverage and gene coverage for Assemblathon 1 groups (taken from [16]). For each group, the assembly which maximizes an overall score of quality metrics is selected. Groups are sorted according to haplotype coverage.

Chapter 6

Beyond classical *de novo* assembly

Classical assemblers, including Monument, rely on a large in-memory structure to perform assembly. By introducing Mapsembler, an index-free targeted assembler, we show that versatile targeted assemblies can be performed on a simple desktop computer. We examine how that method could be used to perform whole-genome, index-free assembly. Finally, several NGS applications of succinct hash tables are presented.

6.1 Targeted assembly: Mapsembler

A general trend in NGS assembly software is the design of memory-efficient implementations. As larger genomes are being sequenced with increased coverage, the amount of data being processed grows faster than the available computational resources. Inevitably, with the sequencing of meta-genomes, even a fraction of a sequencing dataset will be impractical to index in terms of memory usage. Hence, to anticipate this issue, we propose to model a simplified form of de novo assembly, namely *targeted de novo assembly*, using an index-free method. Targeted *de novo* assembly can be defined as the following question: given a sequence s (prior knowledge) and a set of reads R, (i) decide whether s, or variations of s, are present in an assembly of R, and (ii) find which sequences follow and precede s in an assembly of R.

We present an algorithm, Mapsembler, which solves both aspects of this problem. This targeted assembly method aims to be a “Swiss army knife” tool to answer specific questions about NGS datasets.

6.1.1 Methods

The Mapsembler algorithm consists in performing localized *de novo* assembly around a set of starting regions (*starters*). Mapsembler proceeds in two steps:

1. Find if each starter corresponds to a likely assembly from the reads, possibly with variants.

2. Assemble a contig by incrementally extending sequences around each variant.

The first step is described in more details in the following. The second step consists in processing the entire reads dataset at each extension, yielding a local assembly graph. The local assembly graph can be seen as special case of practical non-branching paths with only overlap edges. For a detailed explanation of the extension step, refer to the Mapsembler article [40].

Mapping reads

Reads are first mapped to the starter. Formally, a read r is said to be mapped at position i on a sequence s if

$$\sum_{j=0}^{|s|} d(r[i+j], s[j]), \text{ where } d(\alpha,\beta) = \begin{cases} 0 \text{ if } \alpha = \beta \text{ or } \alpha = \epsilon, \\ 1 \text{ otherwise} \end{cases}$$

For convenience, the character $r[i+j]$ is set to ϵ if $i+j < 0$ or $i+j \geq |r|$, where d is a fixed threshold. In other words, a read is mapped to a sequence when their Hamming distance (the read is padded to fit the size of the sequence) is below d. The notation $s \parallel_i^d r$ is used to denote that r maps on s at position i, with threshold d. Figure 6-1 illustrates this definition.

$$\begin{array}{lccccccccc}
i = & \text{-2} & \text{-1} & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
s = & & & \mathrm{A} & \mathrm{T} & \mathrm{G} & \mathrm{C} & \mathrm{G} & \mathrm{G} & \mathrm{A} \\
 & & & | & | & \cdot & | & | & & \\
r_1 = & \mathrm{G} & \mathrm{A} & \mathrm{A} & \mathrm{T} & \mathrm{T} & \mathrm{C} & \mathrm{G} & & \\
 & & & & | & | & | & \cdot & | & \\
r_2 = & & & & \mathrm{T} & \mathrm{G} & \mathrm{C} & \mathrm{T} & \mathrm{G} &
\end{array}$$

Figure 6-1: Illustration of the definition of mapped reads. The reads r_1 and r_2 are mapped to the sequence s. The error threshold is set to $d = 1$. In our notation, the mapping is denoted by $s \parallel_{-2}^{1} r_1$ and $s \parallel_{1}^{1} r_2$

Multiple consensuses generation

Given a sequence s (called a *starter*) and a set of mapped reads R, we present an algorithm that generates a set of sequences (called *sub-starters*) $s_1, s_2, \ldots, s_n$ which (1) originate from the reads, (2) are coherent with the starter s and (3) are significantly represented. These conditions will be formally defined in the definition of Problem 11.

We are interested in retrieving the largest set of sub-starters for each starter s. To simplify the presentation, reads are assumed to contain no errors. In practice, a preliminary read correction step (included in Mapsembler) effectively corrects or discards erroneous reads. Without loss of generality, each read is assumed to map to s at a single position at most (multiple alignments can be seen as duplicated reads).

Problem 11. Multiple consensuses from reads alignment. Given a starter s, two parameters $c, d \geq 0$ and a set of error-free mapped reads $R = \{r_i \text{ such that } s \parallel_{p_i}^{t} r_i\}$ (by hypothesis, each read r_i aligns to s at a single position p_i with at most t substitutions), find all maximal (with respect to the inclusion order) subsets S_i of R satisfying:

1. each subset S_i admits a single perfect consensus s_i, i.e. each read r_i aligns to s_i at position p_i (relative to s) with no mismatch: $s_i \parallel_{p_i}^{0} r_i$,

2. the consensus s_i of S_i aligns to s with at most d mismatches: $s \parallel_{0}^{d} s_i$,

3. each position of s is covered by at least c reads in S_i.

Note that only substitutions with the starter are considered in this formulation. Indels are more difficult to incorporate, and will not be considered here.

Algorithm

A trivial (exponential) solution is (i) to generate the power set (all possible subsets) of R, (ii) remove sets which do not satisfy one of the propositions above, and (iii) keep only maximal sets (ordered by inclusion). The exponential complexity of this solution clearly comes from step (i). In Algorithm 6, we give a polynomial time (in the number of mapped reads) procedure which subsumes (i), in the sense that it generates a solution which includes all the correct subsets.

Algorithm 6 Generating candidate subsets S_i for solving the multiple consensuses from reads alignment problem

```
Require: Instance of the multiple consensuses from reads alignment problem.
Ensure: Set $S$ of candidate subsets.
 1: $S = \emptyset$.
 2: for each read-position $(r, p)$ in $R$, ordered by alignment position $p$ do
 3:     for each subset $S_i$ in $S$ do
 4:         if r overlaps without substitutions with the right-most read of $S_i$ then
 5:             Add $r$ to $S_i$.
 6:         else
 7:             if $r$ overlaps without substitutions with one of the reads of $S_i$ then
 8:                 Let $(r', p')$ be the last read of $S_i$ overlapping with $r$.
 9:                 Let $T$ be the subset of $S_i$ of all reads up to $(r', p')$.
10:                 Create a new subset $S' = T \cup \{r\}$.
11:                 Insert $S'$ into $S$.
12:     if $r$ was not appended to any subset then
13:         Create a new subset with $r$ and insert it into $S$.
14:     Remove any subset from $S$ if its consensus has more than $d$ differences with $s$, or a
    position before $p$ is covered by less than $c$ reads.
    return $S$.
```

The correctness proof that Algorithm 6 finds all maximal subsets corresponding to correct sub-starters is as follows.

Proof. The proof is by contradiction: let s be a correct sub-starter not found by the algorithm. Let $\{r_1, \dots, r_n\}$ be the maximal subset of reads which yields s, sorted by increasing mapping positions to f. We show by induction that the algorithm returns a subset which includes $\{r_1, \dots, r_k\}$, for $k \in [1..n]$. For $k = 1$, notice that a subset is assigned to each read. Assuming that $\{r_1, \ldots, r_k\}$ is part of a returned subset S_0, we show that $\{r_1, \ldots, r_{k+1}\}$ is also returned. Since r_{k+1} is part of a subset which yields s, it overlaps perfectly with r_k. However, r_{k+1} does not necessarily belong to S_0. Let r'_{k+1} be the read which follows r_k in S_0. In the ordering of the reads by increasing position, if the read r_{k+1} is seen before r'_{k+1}, then the algorithm selects $r'_{k+1} = r_{k+1}$; else, as r_{k+1} perfectly overlaps with r_k, a new subset is created from S_0, which contains exactly $\{r_1, \ldots, r_{k+1}\}$. Eventually, from the induction, a subset which contains $\{r_1, \ldots, r_n\}$ is constructed. Since $\{r_1, \ldots, r_n\}$ is itself maximal, the subset found by the algorithm is exactly $\{r_1, \ldots, r_n\}$. □

Note that Algorithm 6 may return extra subsets, which do not satisfy all the three conditions (e.g. coverage of s after the last aligned read position p is not checked), hence steps (ii) and (iii) are still required. However, this algorithm returns at least all the correct subsets, hence it does not miss solutions.

The running time of the algorithm is now analyzed. Observe that during the algorithm execution, the consensus sequence of each intermediate subset S_i in S has either (i) d or less differences with s or (ii) strictly more than d differences with s. In case (i), the subset S_i is included in a distinct maximal subset. There are at most 4^d maximal subsets, one for each combination of substitutions with s. In case (ii), the subset will be removed as soon as it is created (step 14). Assuming bounded read length, the overlap detection steps 4 and 7 can be performed in $O(r)$ time. Hence, the time complexity of Algorithm 6 is $O(4^d r^2)$, where in practice d is a small constant. This algorithm has been applied in the sub-starter generation and read coherence step of Mapsembler.

6.1.2 Results

Recovering variants: Gene detection in a different strain The folA gene (dihydrofolate reductase) is present in several strains of *E. coli*, including K-12 (chr:49,823-50,302) and O157:H7 (chr:54,238-54,717). The sequence of this gene is not exactly similar between the K-12 and O157:H7 strains: there are 10 single-nucleotide mutations across 479 bp. We attempted to recover the O157:H7 gene sequence of the folA gene from only sequencing reads, using prior knowledge of the K-12 sequence. To this end, we analyzed a dataset of 15.7 M raw reads of length 70bp (SRA:ERR018562) from *E. coli O157:H7*. The K-12 allele of the folA gene (length 479 bp, NCBI ID:944790) was used as the starter. The sub-starter generation module of Mapsembler confirmed the presence of the gene, and furthermore recovered the exact O157:H7 gene sequence of folA from the reads (100% identity with O157:H7 reference). Mapsembler produced these results in 572 seconds and using 1.5 MB of memory.

6.1.3 Towards index-free whole-genome assembly

Mapsembler could be used as a building block to perform whole-genome assembly with an arbitrarily low memory usage. The initial function of Mapsembler is to compute targeted assemblies around a known region of interest, using near-zero memory. An extension of this usage would be to perform whole-genome assembly, by repeatedly assembling around well-chosen reads (*seeds*). Although we have not performed software experiments to validate this usage, a simple theoretical motivation is presented here.

The localized assembly approach presented in Section 4.3 on page 54 aims at assembling disjoint sub-graphs. Here, a similar idea is used. Any read can be associated to the contig it would generate with a targeted assembly (under the classical assembly assumption that repetitions are collapsed, and that each read belongs to at most one contig). Note that the contig is only known *a posteriori*. In Figure 6-2, reads represented by black nodes inside each colored cluster will generate the same contig. The reads represented by white nodes will generate singletons under the PNBP approach due to the graph topology.

Balls and bins model for index-free assembly

The balls and bins model is a classical framework in random processes [36]: m balls are thrown into n bins, the location of each ball is chosen independently and uniformly

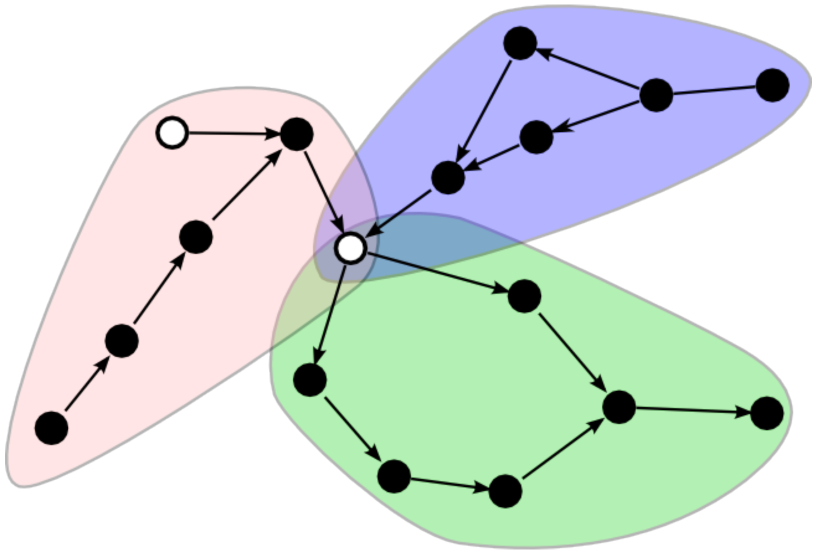

Figure 6-2: Viewing reads as contigs-generating starters. White nodes are those which are likely to generate singletons, all other nodes can be chosen as starting nodes to construct contigs.

at random. Contigs, considered as a cluster of reads, can be viewed as bins. Similarly, the reads can be viewed as balls.

contigs	$\leftrightarrow$	bins
reads	$\leftrightarrow$	balls

Table 6.1: Balls and bins model for index-free assembly.

The aim of index-free assembly is to generate all the contigs (throw at least one ball into all the bins) using as few seeds as possible (use as few balls as possible). If the reads are chosen uniformly at random, a simple analysis gives the expected number of reads to select as seeds.

The expected number of assembled contigs $A_{m,n}$, given m seeds and n total contigs, is the expected number of non-empty bins. Considering that the probability that each bins remains empty is

$$(1-\frac{1}{n})^m \approx e^{-m/n},$$

by linearity of expectation,

$$A_{n,m} = m - ne^{-m/n}.$$

The expected number of seeds m required to assemble all the n contigs can now be computed. To this end, we consider that all contigs are assembled when the expected number of empty bins is lower than 1,

$$ne^{-m/n} < 1 \iff n\log(n) < m. \tag{6.1}$$

Another way to derive this result is to frame index-free assembly as a coupon collector problem [36]. The problem is now: how many boxes of cereal (reads) need to be opened (selected as seeds) before the n different types of coupons (contigs) are found? A classical solution to coupon collector problem yields the same expected number of boxes, $n\log(n)$.

In terms of minimizing the total number of assembled seeds, selecting seeds uniformly at random is certainly not the best strategy. Motivated by the fact that Mapsembler can assemble several seeds in parallel, we consider a variation.

Assembly is now performed in r rounds. During each round, a fixed number of s seeds are assembled. These seeds are selected uniformly from the read pool. At the end of each round, reads which align to assembled contigs are removed from the read pool. Thus, assembling the remaining read pool would yield exactly the contigs which have not been assembled so far. Given a fixed number of seeds s and a fixed number of total contigs n, how many rounds are necessary? The number of un-assembled contigs at round t is noted u^t. By setting $n = u^t$ and $m = s$ in Equation 6.1, the following relation is derived:

$$u^{t+1} = u^t e^{-s/u^t}.$$

Initially, there are $u^0 = n$ un-assembled contigs. All the contigs are assembled when $u^t = 0$.

A rough estimation of the number of rounds needed to assemble an human genome can be made. A typical human genome assembly using Illumina reads consists in $n = 3{\cdot}10^6$ contigs [31]. Table 6.2 shows how many rounds are necessary, depending on the number of seeds assembled at each round. Each round would take much less time to complete than the previous round (since the read pool is gradually diminishing).

s	200	1000	10,000	100,000	1,000,000
r	15,004	3005	305	34	6

Table 6.2: Projected number of rounds (r) for an index-free human assembly ($n = 3 \cdot 10^6$ contigs), given that s seeds are assembled in parallel during each round.

Ideally, as many seeds as possible should be assembled in parallel. But indexing the seeds during each round consumes memory. Hence, let us assume that $s = 1 \cdot 10^5$ seeds are selected at each round, yielding a reasonably low memory usage ($\approx$ 10 MB, assuming index information consumes 100 bytes per seed). Then, assembly converges quite rapidly, as at most $r = 34$ rounds of assembly are required. Should one index ten times more seeds ($s = 1 \cdot 10^6$), only $r = 6$ rounds are necessary. This estimation indicates that index-free assembly seems tractable with massively parallel, cloud architectures with virtually no memory usage.

Discussion

This analysis is only a preliminary step towards practical index-free assembly. First, the uniform choice of bins is only valid if contigs have the same size and the genome is repeat-free. In practice, this is not the case. The probability that a randomly chosen read will assemble a certain contig depends on the length and repeat content of the contig. This probability could be computed if one had information on the repeats of the genome.

Second, this assembly approach may yield a significant amount of redundancy, due to the lack of index. Unless at any given time, reads which align to already constructed contigs are prevented from being selected as seeds, the same contig will be assembled several times. Repetitions would also yield redundancy. For instance, assume that a repetition is present in two copies differing by a single nucleotide. Targeted assembly around reads specific to each copy would yield both copies of the repetition. In traditional assembly methods, both copies are merged into a single consensus sequence, typically the most abundantly sequenced copy. To fix this problem in a targeted assembly framework, a procedure to efficiently remove redundancy in assembled sequences can be implemented using methods described in the following section.

6.2 NGS toolbox supported by static succinct hash tables

The static, succinct k-mer index presented in Section 4.4.5 on page 69 can be used in more applications than just de novo assembly indexing. We show that instances of a static k-mer hash table can be applied straightforwardly to the following problems: short reads read correction, removal of repeated assembled sequences and merging of assemblies. This structure has the potential to enable memory-efficient processing of large datasets, which would be impractical with other classical structures (hash tables, suffix arrays). A related approach (Gk-arrays [41]), based on suffix arrays, has recently been published. In this section, we describe how to design structures which are more specialized (less versatile) than Gk-arrays.

The generic mechanism for constructing a static index can be summarized as follows: (a) a pre-processing phase constructs a dynamic set of k-mers, typically using a hash table or a list of redundant k-mers. The k-mers space can be partitioned to save memory. (b) The keys are either sorted and exported to external memory, or sorted in-memory. (c) Keys of the succinct rank/select index are inserted from the sorted, non-redundant set of k-mers. (d) Values of the static index are computed and inserted in a rank-indexed list. One can see that the full index is only stored in-memory as a whole in the last step.

6.2.1 Error correction

Error correction is perhaps the most immediate application supported by a k-mer table, besides k-mers counting. As a preliminary short digression, k-mers counts can be represented using the static index as follows:

Key:	k-mer
Value:	count

However, storing k-mers counts is not a necessity. Typically, a prior k-mer counting procedure outputs an abundance plot, which provides information for counts quantization. For instance, as we see next, error correction does not necessarily require in-memory storage of k-mer counts.

Methods

Correction is implemented in virtually every assembly software, and also has notable stand-alone implementations [27]. In this paragraph, the details of a error-correction algorithm will not be given. Instead, we sketch the possible interface with a static k-mer index. The static index used is the following:

Key:	k-mer
Value:	quantized count

The quantized count can typically be a solidity bit, determining whether the k-mer is *solid*, i.e. it appears sufficiently many times in the reads to not be considered erroneous. A more elaborate quantized count stores 3 states: non-solid, solid non-repeated, and repeated. The repeated state indicates that k-mers has been observed twice more than the average expected coverage. Such k-mers typically have ambiguous genomic locations, hence yield special cases in error correction.

Note that a prior k-mer counting scheme is necessary to determine solidity. However, this counting step can be efficiently implemented in t turns, using $1/t$ the memory of a global index.

A generic correction mechanism based on this index is described. Each read from the dataset is processed independently. The solidity of each k-mer of the read can be determined using the static index. A read is considered to be correct if all its k-mers are solid. Error-correction consists in finding a minimal set of editions (nucleotide

Method	Memory usage (Mb)
Complete bit array (SOAPdenovo)	16,000
Naive hash table	191
Static succinct hash	60

Table 6.3: Memory usage of data structures supporting error-correction algorithms for a *E. coli* genome. The complete bit array (as used by SOAPdenovo) is compared against a hash table and the static succinct hash. Note that the size of the bit array fixed, while the other two structures will grow according to the genome size.

substitutions or insertions/deletions) that transforms the read into a correct read. Reads that cannot be corrected are discarded.

An immediate improvement for this structure, when using 1-bit solidity, consists in storing only the set of solid k-mers as keys. The values are then unnecessary, and membership queries are performed on the rank/select index. When using three-state solidity, storing a value of 1 bit is sufficient: non-solid k-mers are not indexed, solid non-repeated have a value of 0 and solid repeated have a value of 1.

Memory usage results

We implemented the three-state solidity structure using the previously described improvement. This scheme is compared against two popular methods. The first method is the SOAPdenovo correction module. It implements a complete 17-mer byte-array (using 4^{17} = 16 GB). The second method is a hash table (implemented using the `sparsehash` library), which associates each k-mer with a pointer-sized integer.

These three structures are benchmarked on a *E. coli* dataset consisting of solid 17-mers. Table 6.3 shows the memory usage of each data structure. The bit array is very large, yet its size will never vary. This is a strong advantage for larger genomes. The other two data structures will grow according to the genome size. For a human genome, the size of naive hash table will grow beyond the bit array, as it occupies a constant number of bytes multiplied by around 3 billion. However, because of its entropic bound, the static succinct hash will always be smaller than the complete bit array. Hence, this structure is guaranteed to be also effective for large genomes.

6.2.2 Repeats identification

Identifying repeated substrings between two different scaffolds from a set of assembled sequences is a slightly different problem than presented in the second Chapter. First, inexact repeats are considered, as exact repeats are not supposed to occur inside assemblies (they would be collapsed to a single instance). Second, larger repeats than the read length (gene-sized) are sought. These repeats can typically be retrieved using Mummer, a software which computes pair-wise alignments using a suffix array. We show that the suffix array used in Mummer can be replaced by a static k-mer index. First, assembled sequences (scaffolds) are indexed as follows using this static index:

Key:	k-mer
Value:	number of scaffolds containing the k-mer

As observed in the previous section, an optimized value would be a bit indicating whether 1 or > 2 scaffolds contain this k-mer. An even further optimized table would store only the set of repeated k-mers.

Assembled sequences are processed sequentially. Repeated regions inside each sequences are found using the following algorithm. First, the sequence of nucleotides is transformed into a sequence of bits by setting the i-th position at 1 if and only if the i-th k-mer is repeated (i.e., belongs to 2 or more scaffolds). To detect inexact repeats, a heuristic linear-scan algorithm is used. Specifically, setting a maximum gaps tolerance threshold of g nucleotides, consecutive exactly repeated regions that are separated by less than g nucleotides are returned as an inexact repeat.

This repeat identification process has an important use case. The ABySS and SOAPdenovo software are prone to producing artificially larger assemblies [45]. This can be partly fixed by removing artificially redundant parts using the above procedure. The repeated parts are assumed to be complete suffixes or prefixes of assembled sequences. Sequences strictly contained in larger sequences will also be removed. The values (number of scaffolds containing the k-mer) are stored in a classical integer array and can be modified with no memory penalty. This property allows to decrease the k-mer value each time it is seen in a scaffold. Hence, in the last sequentially processed scaffold containing the k-mer, the k-mer will not be identified as repeated.

6.2.3 Merging assemblies

Suppose that assembler A produces a high-coverage assembly with low NG50, and assembler B produces a low-coverage assembly with higher NG50. Given these two assemblies, how can one construct a hybrid assembly C which has a strictly higher coverage than assembly B and strictly higher NG50 than assembly A?

Methods

A procedure to construct assembly C is based on repeat identification, as performed in the precedent section. First, scaffolds from assembly B are indexed. For each scaffold s of assembly A, overlaps between s and any scaffold of assembly B are retrieved using the repeat identification algorithm above.

Regions of s which overlap with assembly A are discarded, and long enough non-overlapping regions are kept. The length threshold is experimentally set to $2(k+1)$, i.e. two non-redundant nucleotides with flanking k-mers. Assembly C consists of all these non-overlapping regions as well as scaffolds from assembly B. It is easy to see that assembly C has a higher coverage than assembly B (as it includes assembly B) and higher NG50 than assembly A (as appending any sequence to assembly B does not decrease its NG50).

Results

We implemented the above methods in a stand-alone software. Two assemblies produced by Monument of the Assemblathon 2 snake genome (5.3.4 on page 90) with different parameters were merged using the procedure. The results are summarized in Table 6.4. As expected, the merged assembly contains strictly more bases than the high-NG50 one. However, the merged assembly has significantly less bases than the high-coverage one. The reason behind this is that the high-coverage assembly

Assembly	High-N50	High-Coverage	Merged
Total bases (Mbp)	1,479	**2,046**	**1,810**
NG50 (kbp), assuming a 1,479 Gbp genome	**1,297**	1,051	**1,297**
N50 (kbp)	**1,297**	641	**991**

Table 6.4: Merging a high-coverage assembly with a high-NG50 assembly. The resulting merged assembly has an increased number of bases with a similar N50 metric to that of the high-NG50 assembly.

contained self-redundant sequences.

The index used to produce the merged assembly referenced 1.4 billions of k-mers (those of the high-NG50 assembly), and occupied 14 GB of memory. It contained essentially the same information as the structure defined in the repeats identification section. The whole merging process took 1 hour and 40 minutes using one CPU core.

Chapter 7

Conclusion and perspectives

7.1 Conclusion

This conclusion sums up our contributions and results. In the idealized context of perfect re-sequencing, we introduced a suffix array-based algorithm to analyze the gap between single and paired reads in terms of genome coverage (Chapter 2 and [10]). This analysis yields two take-home messages for re-sequencing experiments:

- Paired reads of length l enable to cover a significantly larger portion of the genome than reads of length $2l$.
- Larger insert lengths compensate for shorter read lengths.

We incorporated paired reads explicitly into classical assembly formulations (Chapter 3). This does not change the picture in terms of computational complexity, as both unpaired and paired assembly problems are shown to remain NP-hard. However, the paired assembly problem is shown to become polynomially solvable when repeated regions are interspersed, and shorter than the insert size.

Practical assembly aspects play a major role, as theoretical models do not address assembly ambiguities (multiple solutions), and sequence graphs hardly fit in memory (Chapter 4). A novel, localized assembly approach has been elaborated (Section 4.3 on page 54 and [11]). It combines the memory efficiency of greedy assembly with the locally complete structure of graphs. Furthermore, this approach has been extended to include pairing information, enabling targeted assembly of scaffolds. The following new methods may benefit to other greedy assemblers:

- Constructing a local sequence graph (extension graphs, Section 5.2.1 on page 79) enables to assemble complex variants.
- Using pairing information in the local sequence graph enables to jump over short repetitions, yielding scaffolds and longer contigs (Section 4.3 on page 54).

Two new ideas have been introduced for the indexing of sequencing data (Section 4.4 and [9]):

- Any k-mer-based indexing scheme can be made more memory-efficient by detecting and filtering erroneous k-mers early (Section 4.4.2 on page 61).
- Single and paired reads can be directly indexed in a k-mer-based hash table by dynamic filtering of redundant reads (Section 4.4.3 on page 63).

These mechanisms have been implemented in the Monument assembler, and the whole pipeline is compared against current assembly methods (Chapter 5).

Finally, several problems related to assembly can be efficiently solved using new algorithms (Chapter 6). The Mapsembler algorithm performs targeted assemblies around regions of interest, and recovers read-coherent variants of known fragments (Section 6.1.2 on page 104 and [40]). Succinct hash tables can be used to perform short read error correction, repeats identification in assemblies and merging of assemblies (Section 6.2 on page 109).

7.2 Released software

This thesis resulted in the following open-source software implementations:

pairedRepetitions [1] Computes the ratio of exact, paired (and unpaired) repeated reads within a genome. This is the source code used to produce the results of Chapter 2.

[1] https://github.com/rchikhi/pairedRepetitions

Mapsembler [2] Targeted *de novo* assembly software, presented in Chapter 6.

deBruijn [3] Software that constructs the de Bruijn graph of a set of reads in a memory-efficient manner. This software is used in the KisSplice pipeline[4] to perform *de novo* detection of alternative splicing in RNA-seq data.

Additionnally, two internal programs have been developed:

Monument [5] Localized *de novo* assembly software for paired reads, presented in Chapter 5.

SuperScaffolder Scaffolding algorithm used in conjunction of Monument during the Assemblathon 2 benchmark, under development.

7.3 Perspectives

A common misconception is that genome assembly is a solved problem, at least for small genomes. Perhaps this misconception stems from satisfactory assembly results of ever-improving software and sequencing technologies. There are, however, several areas that, in our opinion, have not been given enough attention.

Scaffolding Classical genome assembly follows the reads→contigs→scaffolds pipeline. While this thesis showed that scaffolds can be built directly from reads, such scaffolds are still interrupted by large repetitions. In general, current assembly methods make no attempt to solve larger repetitions, while theoretical results indicate that repetitions can be solved unambiguously in some cases [38]. Specifically, state of the art scaffolding algorithms only output simple paths. We believe that exploring more complex paths in the contigs graph may lead to better scaffolding algorithms.

[2] http://alcovna.genouest.org/mapsembler/
[3] https://github.com/rchikhi/debruijn
[4] http://alcovna.genouest.org/kissplice/
[5] http://www.irisa.fr/symbiose/people/rchikhi/monument.html

Gap-filling Filling the gaps (undetermined nucleotides, "N") in scaffolds yields significantly longer contigs, hence assemblies of better quality. Some assemblers perform such step using heuristics, especially SOAPdenovo which includes a stand-alone gap-filling module [31]. However, no formal framework of gap-filling has been proposed. A first sketch of such framework could be formulated as follows: finding the set of paths between contig extremities (nodes s and t) which satisfy reads and pairing constraints. A solution is said to exist if there is a unique path between s and t, or a unique sub-path common to every paths.

Index-free assembly The Mapsembler software is a proof of concept that targeted assembly can be performed without any indexing structure. The analysis in Section 6.1.3 on page 105 suggests that the approach can be extended to complete genomes, assuming a high degree of parallelism. Also, the approach has the potential to be extended to perform a complete scaffolds assembly with zero memory.

Polynomial-time theoretical assembly In Chapter 3, we gave a polynomial time reduction of the Paired Assembly Problem in the presence of short interspersed repeats. It is conjectured that a polynomial-time algorithm exists in the unpaired case assuming constant-sized overlaps between reads [38]. Finding weaker cases of polynomial reduction would be an important step towards closing the gap between practical assemblers and theoretical models.

7.4 In a future context

7.4.1 Future of sequencing

The field of DNA sequencing is ever-changing. Only seven years ago, high-throughput sequencers were introduced. Hence, it is hard to make predictions about the state of sequencing, even for the near future. However, the following sequencing scenarios are possible, if not likely, at one point of time:

1. immense-throughput short (< 1000 bp) paired reads, very high coverage ($> 200\times$) of human genome at each run

2. low-throughput longer (> 1000 bp) single or paired reads, medium coverage ($10 - 50\times$)

3. unbounded read length, low coverage ($2 - 5\times$, only to correct sequencing errors)

Each of these scenarios indicates a satisfactory, final state of a sequencing technology. The first trend is the direction followed by the Illumina company. Illumina sequencing is becoming much cheaper, however read lengths appear to be unlikely to become an order of magnitude higher, due to polymerase-based sequencing. Given the recent announcements of nanopore sequencing (as of March 2012), either the second or the third scenarios may happen in the near future with nanopores. The second trend is specifically what {Sanger, 454, Ion Torrent, Pacific Bioscience} sequencing is expected to become. A final scenario is the natural final state of sequencing: a single chromosome is sequenced in a single read, at zero error rate.

7.4.2 Future relevance of this work

Until sequencing achieves unbounded read lengths, *de novo* assembly will still be relevant. However, the assembly problem will become trivial with low-coverage, very long reads instances. The read length threshold at which assembly becomes trivial (for paired reads) can be determined using methods from Chapter 2.

The models presented in Chapter 3 are relevant to the context of assembly, as long as paired reads are produced. Specifically, our methods will certainly fully apply to future Illumina sequencing reads. Our indexing scheme (Section 4.4) has been shown to scale well to higher volumes of data. However, nanopore sequencing is unlikely to produce paired reads. Hence, longer single reads will possibly be assembled using novel methods. The transition from a high volume of short single reads to a high volume of longer single reads may be facilitated by using our assembly model based on local string graph traversals (Sections 4.3 on page 54 and 5.2.1 on page 79).

As long as sequencing technologies are prone to errors, reads error correction will be relevant. Our k-mer indexing method (Section 4.4.2 on page 61) and efficient index for read correction (Section 6.2 on page 109) will scale well with higher volumes of data. Furthermore, the indexing structure are designed to be versatile, and can be applied to more applications than just error correction. For instance, reference-free read compression is a promising application.

Finally, methods developed for Mapsembler apply to a more general context. As metagenomic sequencing will become more popular, determining the diversity of a set of sequenced genomes will be a more pressing problem, independent of the sequencing technology (Section 6.1 on page 100).

List of Figures

List of Tables

Bibliography

[1] P. N. Ariyaratne and W. K. Sung. PE-Assembler: de novo assembler using short paired-end reads. *Bioinformatics*, 27(2):167, 2011. (pages 15, 51, 53, 59, 78).

[2] S. Batzoglou, D. B. Jaffe, K. Stanley, J. Butler, S. Gnerre, E. Mauceli, B. Berger, J. P. Mesirov, and E. S. Lander. ARACHNE: a whole-genome shotgun assembler. *Genome research*, 12(1):177, 2002. (page 51).

[3] M. Boetzer, C. V. Henkel, H. J. Jansen, D. Butler, and W. Pirovano. Scaffolding pre-assembled contigs using SSPACE. *Bioinformatics*, 27(4):578, 2011. (page 91).

[4] S. Boisvert, F. Laviolette, and J. Corbeil. Ray: Simultaneous assembly of reads from a mix of high-throughput sequencing technologies. *Journal of Computational Biology*, 17(11):1519–1533, 2010. (pages 15, 51, 59, 78, 86).

[5] D. Bryant, W. K. Wong, and T. Mockler. QSRA–a quality-value guided de novo short read assembler. *BMC bioinformatics*, 10(1):69, 2009. (page 78).

[6] J. Butler, I. MacCallum, M. Kleber, I. A. Shlyakhter, M. K. Belmonte, E. S. Lander, C. Nusbaum, and D. B. Jaffe. ALLPATHS: de novo assembly of whole-genome shotgun microreads. *Genome Res*, 18:810 – 820, 2008. (pages 83, 85).

[7] 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. (pages 19, 62, 83).

[8] J. A. Chapman, I. Ho, S. Sunkara, S. Luo, G. P. Schroth, and D. S. Rokhsar. Meraculous: De novo genome assembly with short paired-end reads. *PloS one*, 6(8):e23501, 2011. (pages 15, 51, 59).

[9] G. Chapuis, R. Chikhi, and D. Lavenier. Parallel and memory-efficient reads indexing for genome assembly. In *Proceedings of the 9th international conference on Parallel Processing and Applied Mathematics - Volume Part II*, pages 272–280, Torun, Poland, 2012. Springer-Verlag. (pages 60, 67, 116).

[10] R. Chikhi and D. Lavenier. Paired-end read length lower bounds for genome re-sequencing. *BMC Bioinformatics*, 10(Suppl 13):O2, 2009. (pages 115, 139).

[11] R. Chikhi and D. Lavenier. Localized genome assembly from reads to scaffolds: practical traversal of the paired string graph. *Algorithms in Bioinformatics*, page 39–48, 2011. (pages 54, 115, 139).

[12] F. Y. L. Chin, H. C. M. Leung, W. L. Li, and S. M. Yiu. Finding optimal threshold for correction error reads in DNA assembling. *BMC bioinformatics*, 10(Suppl 1):S15, 2009. (page 62).

[13] T. C. Conway and A. J. Bromage. Succinct data structures for assembling large genomes. *Bioinformatics*, 27(4):479, 2011. (pages 59, 70).

[14] E. D. Demaine and M. L. Demaine. Jigsaw puzzles, edge matching, and polyomino packing: Connections and complexity. *Graphs and Combinatorics*, 23:195–208, 2007. (page 42).

[15] N. Donmez and M. Brudno. Hapsembler: an assembler for highly polymorphic genomes. In *Research in Computational Molecular Biology*, page 38–52. Springer, 2011. (pages 52, 53).

[16] D. A. Earl, K. Bradnam, J. S. John, A. Darling, D. Lin, J. Faas, H. O. Yu, B. Vince, D. R. Zerbino, and M. Diekhans. Assemblathon 1: A competitive assessment of de novo short read assembly methods. *Genome Research*, 2011. (pages 91, 94, 99, 123).

[17] J. Edmonds and E. L. Johnson. Matching, euler tours and the chinese postman. *Mathematical programming*, 5(1):88–124, 1973. (pages 35, 48).

[18] P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM (JACM)*, 52(4):552–581, 2005. (pages 30, 59, 73).

[19] S. Gao, N. Nagarajan, and W. K. Sung. Opera: reconstructing optimal genomic scaffolds with high-throughput paired-end sequences. In *Research in Computational Molecular Biology*, page 437–451. Springer, 2011. (page 37).

[20] S. Gnerre, I. MacCallum, D. Przybylski, F. J. Ribeiro, J. N. Burton, B. J. Walker, T. Sharpe, G. Hall, T. P. Shea, and S. Sykes. High-quality draft assemblies of mammalian genomes from massively parallel sequence data. *Proceedings of the National Academy of Sciences*, 108(4):1513, 2011. (page 51).

[21] M. Hossain, N. Azimi, and S. Skiena. Crystallizing short-read assemblies around seeds. *BMC Bioinformatics*, 10(Suppl 1):S16, 2009. (page 51).

[22] D. Huson, K. Reinert, and E. W. Myers. The greedy path-merging algorithm for contig scaffolding. *Journal of the ACM (JACM)*, 49(5):603 – 615, 2002. (page 51).

[23] B. Jackson, P. Schnable, and S. Aluru. Parallel short sequence assembly of transcriptomes. *BMC bioinformatics*, 10(Suppl 1):S14, 2009. (page 59).

[24] D. S. Johnson and M. R. Garey. Computers and intractability: A guide to the theory of NP-completeness. *Freeman&Co, San Francisco*, 1979. (pages 33, 41).

[25] V. Kann. *On the Approximability of NP-complete Optimization Problems*. PhD thesis, Royal Institute of Technology Stockholm, 1992. (page 39).

[26] J. Karkkainen and P. Sanders. Simple linear work suffix array construction. *Automata, Languages and Programming*, page 187–187, 2003. (page 23).

[27] D. R. Kelley, M. C. Schatz, and S. L. Salzberg. Quake: quality-aware detection and correction of sequencing errors. *Genome Biol*, 11(11):R116, 2010. (page 110).

[28] S. R. Kosaraju and A. L. Delcher. Large-scale assembly of DNA strings and space-efficient construction of suffix trees. In *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, page 169–177. ACM, 1995. (page 33).

[29] V. Kundeti, S. Rajasekaran, H. Dinh, M. Vaughn, and V. Thapar. Efficient parallel and out of core algorithms for constructing large bi-directed de bruijn graphs. *BMC bioinformatics*, 11(1):560, 2010. (page 59).

[30] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, and R. Durbin. The sequence alignment/map format and SAMtools. *Bioinformatics*, 25(16):2078, 2009. (page 86).

[31] R. Li, H. Zhu, J. Ruan, W. Qian, X. Fang, Z. Shi, Y. Li, S. Li, G. Shan, and K. Kristiansen. De novo assembly of human genomes with massively parallel short read sequencing. *Genome research*, 20(2):265, 2010. (pages 50, 51, 59, 68, 84, 91, 107, 118, 135).

[32] A. L. McGuire, M. K. Cho, S. E. McGuire, and T. Caulfield. The future of personal genomics. *Science (New York, NY)*, 317(5845):1687, 2007. (page 8).

[33] P. Medvedev, K. Georgiou, G. Myers, and M. Brudno. Computability of models for sequence assembly. *Algorithms in Bioinformatics*, page 289–301, 2007. (pages 14, 35, 36, 41, 42).

[34] P. Medvedev, S. Pham, M. Chaisson, G. Tesler, and P. Pevzner. Paired de bruijn graphs: a novel approach for incorporating mate pair information into genome assemblers. In *Research in Computational Molecular Biology*, page 238–251. Springer, 2011. (pages 52, 53).

[35] J. R. Miller, S. Koren, and G. Sutton. Assembly algorithms for next-generation sequencing data. *Genomics*, 95(6):315–327, 2010. (pages 10, 50, 57, 59).

[36] M. Mitzenmacher and E. Upfal. *Probability and computing: Randomized algorithms and probabilistic analysis*. Cambridge Univ Pr, 2005. (pages 105, 107).

[37] E. W. Myers. Toward simplifying and accurately formulating fragment assembly. *Journal of Computational Biology*, 2(2):275–290, 1995. (page 34).

[38] N. Nagarajan and M. Pop. Parametric complexity of sequence assembly: theory and applications to next generation sequencing. *Journal of computational biology*, 16(7):897–908, 2009. (pages 35, 46, 117, 118).

[39] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. *Arxiv preprint cs/0610001*, 2006. (pages 59, 70).

[40] P. Peterlongo and R. Chikhi. Mapsembler, targeted and micro assembly of large NGS datasets on a desktop computer. *BMC Bioinformatics*, (1):48, 2012. (pages 79, 87, 101, 116, 140).

[41] N. Philippe, M. Salson, T. Lecroq, M. Leonard, T. Commes, and E. Rivals. Querying large read collections in main memory: a versatile data structure. *BMC Bioinformatics*, 12(1):242, 2011. (page 109).

[42] M. Pop, D. S. Kosack, and S. L. Salzberg. Hierarchical scaffolding with bambus. *Genome Research*, 14(1):149, 2004. (page 51).

[43] M. Pop, S. L. Salzberg, and M. Shumway. Genome sequence assembly: Algorithms and issues. *Computer*, 35(7):47–54, 2002. (page 13).

[44] R. Raman, V. Raman, and S. R. Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms (TALG)*, 3(4):43–es, 2007. (page 59).

[45] S. L. Salzberg, A. M. Phillippy, A. Zimin, D. Puiu, T. Magoc, S. Koren, T. J. Treangen, M. C. Schatz, A. L. Delcher, and M. Roberts. GAGE: a critical

evaluation of genome assemblies and assembly algorithms. *Genome Research*, 2011. (page 112).

[46] F. Sanger, S. Nicklen, and A. Coulson. DNA sequencing with chain-terminating inhibitors. *Proc Natl Acad Sci USA*, pages 5463 – 7, 1977. (pages 12, 31).

[47] B. Schmidt, R. Sinha, B. Beresford-Smith, and S. J. Puglisi. A fast hybrid short read fragment assembly algorithm. *Bioinformatics*, 25(17):2279, 2009. (page 78).

[48] T. Shea, L. Williams, S. Young, C. Nusbaum, D. Jaffe, I. MacCallum, D. Przybylski, S. Gnerre, J. Burton, I. Shlyakhter, A. Gnirke, J. Malek, K. McKernan, and S. Ranade. ALLPATHS 2: small genomes assembled accurately and with high continuity from short paired reads. *Genome Biology*, 10(10):R103, 2009. (pages 65, 88, 90).

[49] J. Shendure and H. Ji. Next-generation DNA sequencing. *Nat Biotech*, 26(10):1135–1145, Oct. 2008. (pages 9, 11, 64, 65).

[50] J. T. Simpson and R. Durbin. Efficient construction of an assembly string graph using the FM-index. *Bioinformatics*, 26(12):i367, 2010. (pages 51, 59, 73).

[51] J. T. Simpson, K. Wong, S. D. Jackman, J. E. Schein, S. J. M. Jones, and I. Birol. ABySS: a parallel assembler for short read sequence data. *Genome Research*, 19(6):1117–1123, 2009. (pages 51, 59, 60).

[52] G. Sutton, J. R. Miller, A. L. Delcher, S. Koren, E. Venter, B. P. Walenz, A. Brownley, J. Johnson, K. Li, and C. Mobarry. Aggressive assembly of pyrosequencing reads with mates. *Bioinformatics*, 24(24):2818–2824, 2008. (page 51).

[53] R. L. Warren, G. G. Sutton, S. J. M. Jones, and R. A. Holt. Assembling millions of short DNA sequences using SSAKE. *Bioinformatics*, 23(4):500 – 501, 2007. (page 59).

[54] J. Wetzel, C. Kingsford, and M. Pop. Assessing the benefits of using mate-pairs to resolve repeats in de novo short-read prokaryotic assemblies. *BMC Bioinformatics*, 12(1):95, 2011. (page 13).

[55] N. Whiteford. *String Matching in DNA Sequences: Implications for Short Read Sequencing and Repeat Visualisation.* PhD thesis, UNIVERSITY OF SOUTHAMPTON, 2007. (pages 29, 30).

[56] 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 Research*, 33(19):e171, 2005. (pages 18, 23, 24, 28, 132).

[57] D. R. Zerbino and E. Birney. Velvet: algorithms for de novo short read assembly using de bruijn graphs. *Genome Res*, 18:821 – 829, 2008. (pages 51, 57, 68, 85).

Chapter 8

Extended summary in French

Chapitre 1: Introduction

Dans cette thèse, nous présentons des méthodes de calcul (modèles théoriques et algorithmiques) pour effectuer la reconstruction de séquences d'ADN. Il s'agit de l'assemblage *de novo* de génome à partir de *lectures* (courte séquences ADN) produites par des séquenceurs à haut débit. Ce problème est difficile, aussi bien en théorie qu'en pratique.

Du point de vue théorique, les génomes sont structurellement complexes. Chaque instance d'assemblage *de novo* doit faire face à des ambiguïtés de reconstruction. Autrement dit, les lectures permettent de calculer plusieurs reconstruction possibles, une seule étant correcte. Comme il est impossible de déterminer laquelle, une approximation fragmentée du génome est retournée.

Du point de vue pratique, les séquenceurs produisent un énorme volume de lectures, avec une redondance élevée. Une puissance de calcul importante est nécessaire pour traiter ces lectures. Le séquençage ADN évolue désormais vers des génomes et méta-génomes de plus en plus grands, dépassant la taille du génome humain de plusieurs ordres de grandeurs. Ceci renforce la nécessité de méthodes efficaces pour l'assemblage *de novo*.

La nouvelle génération des techniques de séquençage produit des lectures plus courtes que la précédente. En compensation, une majorité des organismes séquencés le sont avec des lectures *pairées*. Ces lectures pairées proviennent du séquençage des extrémités d'une lecture plus longue (appelée *insert*). Ainsi, l'information de distance entre deux lectures d'une paire permet de résoudre certaines ambiguïtés de positionnement de ces lectures. Cependant, à notre connaissance, peu de travaux précédents cette thèse se sont intéressés à évaluer les différences entre l'assemblage avec et sans les lectures pairées.

Cette thèse présente de nouvelles contributions en informatique autour de l'assemblage de génomes. Ces contributions visent à incorporer plus d'information pour améliorer la qualité des résultats, et à traiter efficacement les données de séquençage afin de réduire la complexité du calcul. Plus précisément, nous proposons un nouvel algorithme pour quantifier la couverture maximale d'un génome atteignable par le séquençage, et nous appliquons cet algorithme à plusieurs génomes modèles. Nous formulons une série de problèmes informatiques qui incorporent l'information des lectures pairées dans l'assemblage, et nous étudions leur complexité.

Cette thèse introduit le concept d' *assemblage localisé*, qui consiste à construire et parcourir un graphe d'assemblage partiel. L'assemblage localisé combine les avantages des algorithmes gloutons, en terme d'utilisation mémoire, avec l'information localement complète des algorithmes à base de graphes. Nous avons développé le premier assembleur (Monument) qui construit des *scaffolds* (séquences pouvant contenir des nucléotides indéterminés) directement à partir des lectures. Monument se base sur un nouvel objet mathématique, les graphes de chaînes de caractères pairées. Pour économiser l'utilisation de la mémoire, nous utilisons des structures de données optimisées spécifiquement pour la tâche d'assemblage. Nous avons aussi étudié la possibilité d'assembler des génomes sans aucune structure d'indexation. Un outil, Mapsembler, a été développé pour illustrer cette technique.

Chapitre 2

Le re-séquençage d'un génome consiste à aligner les lectures sur une séquence de référence. L'objectif est d'améliorer la qualité de la séquence de référence et/ou de détecter des variations (par exemple des SNPs ou indels). Whiteford *et al.* [56] ont établi que la *faisabilité du re-séquençage* est déterminée par le pourcentage de lectures représentées à un endroit unique dans la séquence de référence. Leurs simulations ont montré que des lectures de 50-100 pb (paires de bases) sont suffisamment longues pour re-séquencer un génome humain.

En procédant à une analyse étendant celle de Whiteford *et al.*, nous étudions dans quelle mesure le re-séquençage d'un génome est possible avec des lectures ultra-courtes et pairées. Dans un premier temps, la notion d'unicité de lectures non pairées est rappelée. Puis, cette notion est étendue à l'unicité des lectures pairées. Dans un second temps, l'algorithme permettant de calculer efficacement le pourcentage de lectures uniques est rappelé. Une variation plus efficace en mémoire de cet algorithme est introduite. Un algorithme permettant de calculer le pourcentage de lectures pairées uniques est présenté. Cet algorithme permet d'obtenir des bornes inférieures sur la taille des lectures pairées pour le re-séquençage de différents génomes modèles.

Nous constatons que 97,4 % du génome d'*E. coli* est couvert par des lectures pairées uniques de taille 8 bp. Pour le génome humain, 90 % des lectures pairées de taille 11 bp et d'insert de 300 pb sont uniques. Ces résultats suggèrent que pour de grands génomes, le re-séquençage nécessite des lectures pairées beaucoup plus courtes (pour *H. sapiens* , environ 70 % plus courtes) pour atteindre une couverture comparable à lectures non pairées. Par ailleurs, la taille des *inserts*, c'est à dire la distance entre deux lectures dans une paire, joue un rôle crucial dans le re-séquençage. Par exemple, le génome complet de *E. coli* est entièrement couvert par des lectures pairées uniques dès l'instant que la taille des inserts dépasse 5000 pb, et que cette taille est fixe. Cette analyse met en évidence que la taille d'insert a un rôle aussi important que la taille des lectures.

Chapitre 3

L'assemblage *de novo* consiste à retrouver la séquence ADN d'une génome uniquement à partir d'un ensemble de lectures courtes. Dans un premier temps, les modèles classiques d'assemblage sont rappelés: problème de la plus petite sous-chaîne de caractères commune, assemblage avec le graphe de de Bruijn, assemblage avec le graphe

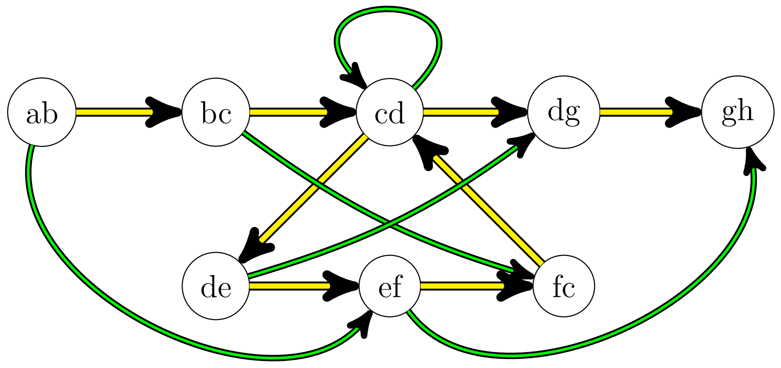

Figure 8-1: Exemple d'un graphe de chaînes de caractères pairées construit à partir de lectures pairées (taille d'insert de 6) couvrant la séquence $S = ab\mathbf{cde}f\mathbf{cd}gh$. Les arcs verts représentent des liens pairés, les arcs jaunes représentent les chevauchements entre les lectures de taille 1.

de chaînes de caractères (*string graph*).

Nous étudions la complexité de calcul des nouveaux modèles d'assemblage, liés au contexte du séquençage de dernière génération: l'assemblage des lectures pairées. Dans ce Chapitre, nous montrons que le problème de recherche de super-chemins dans un graphe de de Bruijn, ainsi que la recherche de chemins Hamiltoniens, peuvent être étendus pour intégrer l'information des lectures pairées. Des variantes pairées de deux problèmes classiques sont aussi étudiées: la recherche de plus courte super-chaîne de lectures pairées, ainsi que les puzzles où les pièces sont pairées. Il est montré que la complexité de tous ces problèmes pairées est NP-dure, c'est à dire que l'information pairée ne permet pas de significativement simplifier le problème.

Nous introduisons un nouveau modèle de graphe, le graph de chaînes de caractères pairées, défini comme une extension du graphe de chaînes de caractères classique, sur un ensemble de paires de lectures (Figure 8-1). Ce modèle permet de formuler de manière naturelle l'assemblage des lectures pairées.

Avec ce modèle, nous formulons un résultat de complexité paramétrique. Ce résultat indique que l'assemblage peut être résolu en temps polynomial lorsque les répétitions du génome sont plus courtes que la taille de l'insert, et suffisamment espacées. Ceci montre que les lectures pairées facilitent l'assemblage pour une classe particulière de génomes.

Chapitre 4

Les résultats du Chapitre précédent montrent que l'assemblage, même en utilisant des lectures pairées, demeure un problème difficile à cause des répétitions génomiques. En pratique, calculer un assemblage, même approximatif, est une tâche difficile en terme de temps de calcul et d'utilisation mémoire. Pour un séquençage réel, l'assemblage ne peut être pas résolu exactement comme une instance de l'un des problèmes définis dans le chapitre précédent. En effet, ces problèmes ont typiquement de nombreuses solutions équiprobables, et seulement une seule solution est biologiquement correcte (celle correspondant au génome réel). Les répétitions génomiques plus longues que la longueur des lectures sont le principal facteur d'un grand nombre de solutions. Les heuristiques classiques d'assemblage consistent à retourner en sortie un ensemble de chemins linéaires (*contigs*) à partir d'un graphe de chaînes de caractères ou d'un graphe de de Bruijn.

Ce Chapitre introduit un nouveau mécanisme visant à intégrer l'information pairée dans un algorithme d'assemblage. L'information pairée permet d'éliminer les ambiguïtés liées aux répétitions qui sont plus longues que la longueur de lecture, mais plus courtes que la taille de l'insert. Pour cela, la notion de chemin non-branchant est introduite. Les chemins non-branchants sont des chemins qui traversent des parties du graphe où aucune branchement ne se produit, par rapport au type d'arc entrant et sortant. Par exemple, le chemin

$$\{ab \dashrightarrow ef \dashrightarrow gh\},$$

est un chemin non-branchant dans le graphe de la Figure 8-1.

Un autre problème important lié aux instances pratiques de l'assemblage est l'énorme utilisation de mémoire des structures de données. Pour un génome humain, un assembleur optimisé basé sur les graphes de de Bruijn nécessite des centaines de gigaoctets de mémoire [31]. Nous proposons une procédure d'indexation conçue spécifiquement pour les lectures pairées. Deux nouvelles méthodes de filtrage sont introduites afin de réduire l'utilisation de mémoire: une procédure pour supprimer les k-mers erronés à la volée, ainsi qu'une procédure pour éviter de référencer les lectures redondantes. En substance, cette structure référence un ensemble représentatif des lectures pour chaque k-mer correct.

Un prototype de cette méthode d'indexation est appliqué à des données Illumina réelles. Nos tests montrent que l'index nécessite 30-50% de moins de mémoire que SOAPdenovo, avec des temps d'indexation comparables. En remplaçant la table de hachage classique par une table statique optimisée, la réduction de l'espace mémoire atteint 86%.

Chapitre 5

Les deux concepts introduits au Chapitre précédent (chemins non-branchants et procédure d'indexation) sont combinés dans l'implémentation d'un nouveau logiciel d'assemblage, Monument.

Monument consiste en deux modules: indexation et assemblage. Le module d'indexation suit la procédure décrite dans le Chapitre précédent. Le module d'assemblage de Monument construit tous les chemins non-branchants possibles, en évitant de réutiliser plusieurs fois chaque noeud. Pour construire chaque chemin, le module construit un sous-graphe de chaînes de caractères pairées, uniquement à partir de l'information des lectures référencées dans l'index.

Algorithm 7 Procédure d'assemblage de Monument

```
1: Choisir une séquence non assemblée $s^{(0)}$
2: t ← 0
3: **Répéter**
4:     $s^{(t)'} \leftarrow$ l'extension $s^{(t)}$ avec un graphe d'extension
5:     $s^{(t+1)} \leftarrow$ l'extension de $s^{(t)'}$ avec une extension pairée
6:     t $\leftarrow t + 1$
7: **Tant que** l'extension pairée précédente a réussi
8: $s_g^{(t)} \leftarrow$ remplacer les nucléotides indéterminés dans $s^{(t)}$ (*gap-filling*)
9: **Retourner** le scaffold assemblé $s_g^{(t)}$
```

L'Algorithme 7 décrit les étapes d'assemblages de Monument. Pour plus d'efficacité, les régions où l'information pairée n'est pas nécessaire sont assemblées à l'aide d'une autre structure: un graphe d'extension. Le graphe d'extension est un graphe de de Bruijn, où les régions sans branchements sont compressées en un seul nœud au moment de la construction. Les régions où l'information des lectures pairées permet d'élucider une répétition génomique sont assemblées à l'aide d'un graphe de chaînes de caractères pairées (étape dite d'extension pairée). Les k-mers déjà assemblés sont marqués. Pour démarrer l'assemblage d'un chemin, une région non-assemblée est d'abord déterminée. Pour cela, un assemblage de courte longueur sur des k-mers non marqués, à l'aide d'un algorithme glouton, est effectué en utilisant seulement la structure d'indexation.

Monument est comparé avec d'autres assembleurs populaires pour lectures Illumina, sur divers ensembles de données. Tout d'abord, un ensemble de données bactériennes montre que Monument obtient des résultats légèrement meilleurs que Velvet et Ray en termes de N50 (médiane pondérée des scaffolds). Surtout, il est démontré que la méthode d'assemblage de Monument donne d'aussi bons résultats que celle de Velvet, basée sur un graphe complet. Contrairement à un autre algorithme heuristique glouton (Ray), Monument n'est pas sensible à des variants de séquence (SNPs, indels). Puis, l'assemblage d'un génome plus grand (fungus) démontre que Monument est capable d'assembler avec une diminution quasi-lineaire du temps de calcul en parallèle. Les résultats sont comparables ou meilleurs (en termes de N50, précision et couverture) à deux autres assembleurs (SOAPdenovo et Allpaths).

Enfin, les résultats de compétitions d'assemblage (Assemblathon 1 & 2, dnGASP) sont analysés. Pour Assemblathon 1, notre pipeline basé sur Monument obtient de bons résultats en termes de scaffold NG50, erreurs structurelles, et arrive premier en temps d'exécution et utilisation mémoire. Des faiblesses du pipeline ont été identifiées en termes de contig NG50 et de couverture. Pour Assemblathon 2, des résultats préliminaires indiquent que notre pipeline est désormais compétitif avec les meilleurs pipelines d'assemblage en termes de qualité des résultats.

Chapitre 6

Ce Chapitre traite d'algorithmes dépassant le cadre de l'assemblage de novo classique. Nous introduisons une forme simplifiée d'assemblage de novo, nommée assemblage de novo ciblé, en utilisant une méthode sans indexation. L'algorithme, Mapsembler, consiste à effectuer un assemblage localisé *de novo* autour d'un ensemble de régions de départ (*starters*). Plus précisément, Mapsembler détermine si chaque région de départ correspond à un assemblage des lectures, éventuellement avec des variants. Il assemble ensuite chaque région de départ, par extensions successives. L'algorithme principal de Mapsembler, décidant si une région de départ correspond à un alignement multiple de lectures, est décrit dans ce chapitre. Le problème combinatoire sous-jacent est tout d'abord défini: il s'agit de construire le plus grand ensemble de séquences (1) cohérentes avec les lectures, (2) cohérentes avec la région de départ et (3) confirmées par suffisamment de lectures. La preuve de complétude d'un algorithme polynomial de résolution est donnée.

Pour évaluer la méthode sur un cas réel, Mapsembler a été appliqué à la détection de la séquence génique du gène folA (issu de l'organisme *E. col K-12*) dans une autre souche d'*E. coli* (O157:H7). Mapsembler a confirmé la présence du gène, et a assemblé la séquence présente dans O157:H7 sans erreur, en dix minutes et 1,5 Mo de mémoire.

Mapsembler pourrait être utilisé comme un bloc de base pour effectuer un assemblage complet d'un génome, avec une empreinte mémoire arbitrairement faible. Une analyse dérivée du modèle *balls and bins* est présentée pour étayer cet argument. Avec cette analyse, nous estimons qu'en utilisant 10 Mo de mémoire, 34 étapes d'assemblages avec Mapsembler de $1 \cdot 10^5$ régions de départ seraient nécessaires pour assembler un génome humain complet. Un tel calcul peut être effectué efficacement sur un cluster.

Enfin, une boîte à outils méthodologique pour divers analyses NGS est présentée. Cette boîte à outils est basée sur une table de hachage succincte statique, qui est destinée à réduire l'utilisation mémoire des méthodes classiques. Plus précisément, les algorithmes de correction d'erreur peuvent être rendus plus économes en mémoire en stockant un comptage réduit des k-mers dans le tableau statique. Deux autres algorithmes sont présentés: l'identification de répétitions dans un assemblage, et la fusion des deux assemblages pour obtenir un troisième assemblage de meilleure qualité. Ces algorithmes sont efficaces en mémoire: la fusion de deux assemblages contenant approximativement 1.4 milliards de k-mers distincts a nécessité 14 Go de mémoire, et s'est effectuée en moins de deux heures sur un seul coeur CPU.

Chapitre 7: Conclusion et perspectives

Dans le contexte du re-séquençage, nous avons introduit un algorithme basé sur un tableau de suffixes pour analyser les différences entre les lectures unique et pairées, en termes de couverture du génome (Chapitre 2 et [10]). Cette analyse met en évidence deux points: (i) les lectures pairées de longueur l permettent de couvrir une partie beaucoup plus importante du génome que les lectures simples de longueur $2l$, et (ii) des inserts plus longs permettent de compenser des tailles de lectures plus courtes.

Nous avons incorporé l'information des lectures pairées dans les formulations classiques du problème d'assemblage (Chapitre 3). Cela ne change pas la complexité de calcul, les problèmes d'assemblage à la fois non pairés et pairés sont tous NP-durs. Cependant, il est établi que le problème d'assemblage des lectures pairée possède une solution en temps polynomial lorsque les régions répétées sont dispersées, et plus courtes que la taille des inserts.

Les aspects pratiques liés à l'assemblage jouent un rôle majeur. En effet, les modèles théoriques n'indiquent pas comment résoudre les ambiguïtés de reconstruction. Ils ne tiennent pas non plus compte du fait que les graphes (de chaînes de caractères ou de de Bruijn) ont une empreinte mémoire souvent élevée. Une approche d'assemblage dite localisée a été élaborée (Section 4.3 et [11]). Elle combine l'efficacité en mémoire d'un assembleur glouton avec une structure localement complète. En outre, cette approche a été étendue pour inclure les informations des lectures pairées, permettant un assemblage ciblé de scaffolds.

Enfin, plusieurs problèmes liés à l'assemblage peuvent être efficacement résolus en utilisant de nouveaux algorithmes (Chapitre 6). L'algorithme Mapsembler permet d'effectuer des assemblages ciblés autour de régions d'intérêt (Section 6.1.2 et [40]). Des tables de hachages succinctes donnent lieu à de nombreuses applications plus économes en mémoire: correction d'erreurs dans les lectures, identification de séquences répétées dans un assemblage, et fusion de deux assemblages (Section 6.2).