Novel components at the periphery of long read genome assembly tools

Pierre Marijon

École Doctorale SPI

UMR 9189 - CRIStAL

Thèse

Présentée pour l'obtention du grade de DOCTEUR
DE L'UNIVERSITE DE LILLE

par

Pierre Marijon

Novel components at the periphery of long read genome assembly tools

---

Spécialité : Informatique

Soutenue le 2 décembre 2019 devant un jury composé de :

Rapporteur	**Tobias Marschall**	(Prof, Max Planck Institut)
Rapportrice	**Marie-France Sagot**	(DR, Inria Grenoble Rhône-Alpes)
Directeur de thèse	**Jean-Stéphane Varré**	(Prof, Université Lille, CRIStAL)
Co-encadrant de thèse	**Rayan Chikhi**	(CR, Institut Pasteur & CNRS)
Examinatrice	**Blerina Sinaimeri**	(Université Lyon I – INRIA)
Examinatrice	**Clarisse Dhaenens**	(Université Lille)
Examinateur	**Pierre Peterlongo**	(CR, Inria Rennes - Bretagne Atlantique)

Thèse effectuée au sein de **l'UMR 9189 - CRIStAL**

de l'Université de Lille
Bâtiment M3 extension
avenue Carl Gauss
59655 Villeneuve d'Ascq Cedex
France

Remerciment

Ces remerciments contrairement a cette thèse seront écris en Français et ne seront pas relue même part moi. En effet je suis dyslexique dysorthographique, je souhaite que ces remerciments ne soit pas corrigé pour montré le travail qui a été accomplis par toutes les personnes qui mon accompagné durant toutes ces années. Qui on relu mes lettres de motivation, rapport de stage, CV, la moindre de mes communications officiel. Donc je tiens a remercié ma mère qui c'est en tout premier chargé de ce travail titanesque, et mes camarades étudiant·e·s. Mais aussi les relecteurs de touts ou partie de cette thèse, Antoine Limasette, Camille Marchet, Nicolas Vinarnick, Yoann Dufresne, Matthieu Falce, Kevin Gueuti, Pierre Morisse, Maxime Garcia, Guillaume Devailly, Jérome Pivert, Aurélien Beliard, bjonnh, Sacha Schutz, Rayan Chikhi et Jean-Stéphane Varré.

Je souhaiterais remercié Tobias Marshall, Marie-France Sagot, Blerina Sinaimeri, Clarisse Dhaenens et Pierre Peterlongo d'avoir accepté de faire partie de mon jury.

Je souhaite remercié aussi Rayan Chikhi et Jean-Stéphane Varré qui on été mes directeurs de thèse, pour leurs aides durant ces trois années. Ils mon guidé et assisté dans mon entré dans le monde de la recherche.

Je pense aussi aux personnes qui mon supporté physiquement pendant ces 3 années je parle évidement de mes co-bureau. Tous d'abord Samuel, qui ma accueilli dans son bureau je me souviendrais toujours de nos discussions aussi bien scientifique que politique. Ensuite Antoine qui ma souvent aidée a réfléchire, enfin Mael et Arnaud qui mon supporté moins long temps certe mais durant la dernière phase de rédaction de ma thèse.

Je tiens aussi a remercie l'ensemble des membres de l'équipe Bonsai, du Laboratoire CRISTAL et de l'université de Lille et du centre INRIA Lille Nord Europe qui mon accueuil durant cette thèse.

Je souhaiterais aussi remercié le blog Bioinfo-fr et l'association des Jeunes Bioinformaticiens Français (JEBIF) a travers ces organisation j'aimerais remercié l'ensemble de la communauté bioinformatique Française pour son accueil bienviellant.

Je souhaiterais aussi remercie tous les amis que j'ai rencontré a Lille, les camarades du Syndicat Solidaires Étudiant·e·s, les biodouilleurs d'ABL, les libriste de Chtinux. Je prèfere ne cité personne nommément, pour être sur de ne froissé personne. Mais sachez que si vous vous reconnaissez dans un de ces groupes je vous doit et cette thèse vous doit beaucoup. Évidement j'ai une pensé particulière pour tous les individus qui on subis plus ou moins volontairement le compte twitter de ma thèse.

Mais aussi des amis plus vieux que je n'est pu voir que de temps en temps, que ce soit des amis d'enfance ou d'étude, sachez que vous m'avez aussi amener sur ce chemins.

Je ne peut rédiger ces remerciments sans cité, Léa, Zoé et Flavien, qui plus que d'autre mon aider a parfois oublié la thèse et qui l'on aussi un peut porté avec moi.

Je tiens a remercie évidement toutes ma famille pour leurs soutient et leurs assistance de tous les instants. Mais aussi mes nombreux coloc, et particulièrement Matthieux qui a fais attention a ce que je manger pendant la rédaction de cette thèse.

Je dédis cette thèse a Hélene ma tante et Jean-Paul mon grand père qui sont mort durant cette thèse.

Contents

Chapter 1

Introduction

"Only what is evolving is alive" [1] – this definition of life, like many others, is incomplete. And one could probably even find some counter-examples to it. One should first need to define what evolution is. We can try to define the evolution of a thing as its changes to optimize the capability to conserve itself. To do that, such a thing needs some form of memory.

In the majority of current known life, the physical support of this memory is DNA, which stands for DeoxyriboNucleic Acid. DNA is a molecule composed of two strands. Each strand is composed of a phosphate backbone. Along these backbones, we have a sugar linked to a nucleic acid. We have four types of nucleic acids: Adenine (A), Thymine (T), Cytosine (C) and Guanine (G), Figure 1.1 shows the 3D structure of DNA.

The two strands of DNA are linked by their nucleic acids, with some rules. In front of an A we will always have a T, in front of a C we will always have a G and vice-versa. A DNA strand is thus the complementary of the other. We say DNA is composed of two anti-parallel strands. By convention we will always represent DNA in one orientation and will omit the other.

In bioinformatics, we generally represent a DNA strand by a single string on a four letter alphabet (A, C, T, G). The properties described above allow us to reconstruct the composition of one strand from the other by using the complementary letters (replace A by T, T by A, C by G and G by C) and reverse the order.

With many complex mechanisms, not detailed here, information contained in DNA is used to build essential molecules to keep the organism alive, and to reproduce it. This information is therefore the basis of the organism's functioning. If this information is destroyed or modified, the living organism will behave differently or die. Thus, knowing and understanding the succession of DNA bases is an effective entry point for analyzing many biological phenomena, diseases, and evolution.

To read this information, we rely on many biochemical techniques that we group under the term sequencing techniques. These techniques allow to read fractions of DNA fragments that are more or less long, and with various error rates.

[1] Pierre Kerner translation from french, original quote "N'est vivant que ce qui évolue"

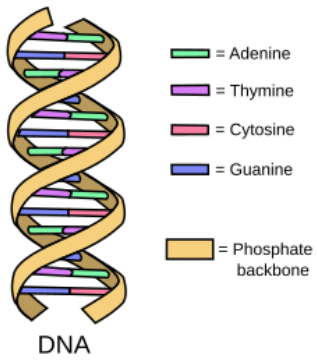


Figure 1.1: Structure and composition of DNA. Source: Wikipedia https://en.wikipedia.org/wiki/File:DNA_simple2.svg

1.1 Sequencing

Sequencing technologies evolved quickly since 1977 [91]. Today we distinguish three generations of sequencing technologies, based on their properties. In this section we focus on sequencing technologies properties and their impact on different bioinformatics tasks and do not detail the underlying biochemical methods.

The two most important properties of a sequencing technology are the size of the DNA fragments it can read, expressed in number of bases, and also the number of errors that the technology will produce, expressed in percentage. An error rate of 0.1% indicates that the sequencer will make one error every thousand bases. When sequencing can read large fragments we have more information about the original sequence, which facilitates downstream analysis. If a read contains many errors (replace a letter by an other one, insert random letter(s) or skip one or more letters), using the information provided by sequencing may be impossible or would require additional operations to correct those errors. Those operations will sometimes be very expensive, in terms of computer time.

Generation	Technology	Read length (bd)	Error rate	Source
First	Sanger	$\approx$ 2 kb	Low ($\approx$ 2%)	[76]
Second	ABI/Solid	75	Low ($\approx$ 2%)	[76]
Second	Illumina/Solexa	100–150	Low (<2%)	[76]
Second	IonTorrent	$\approx$ 200	Medium ($\approx$ 4%)	[76]
Second	Roche/454	400–600	Medium ($\approx$ 4%)	[76]
Third	Pacific Biosciences	$\approx$ 10 kb (max 100 kb)	High ($\approx$ 18%)	[76] [93]
Third	Oxford Nanopore	$\approx$ 10 kb (max 1 mb)	High ($\approx$ 12%)	[93] [86]

Table 1.1: This table presents length of reads and error rate of main sequencing technology. Pacific Biosciences and Oxford Nanopore evolve quickly and different papers may report diverse figures. .

Sanger technique produces long reads with very small error rate, but with a very low throughput and a very expensive cost per base. Second generation appeared in the mid-2000s. It increased the throughput and reduced the cost per base, but reduced dramatically the length of the reads and increases the probability of error ($\approx$ 1%). The most frequent error type for this technology is a substitution between two nucleotides, (i.e. sequencer reads `A` in place of a `T`). Third generation dates

back to the early 2010s. It has greatly increased the size of the reads but also the error rate while maintaining a good throughput. Errors in third generation are mostly insertion or deletion, sequencer didn't read a part of sequence or introduce random base not present in original sequence. Table 1.1 presents read lengths and error rates of many sequencing technologies. We would like to emphasize that both second and third generation technologies are still used today, and sometimes both technologies are used for a single experiment, as we will discuss later about hybrid techniques.

With sequencing one can read all information contained in a genome. But, no matter the technology, we get lots of (short) unordered fragments. Genome assembly therefore designates the task of reconstructing the original sequence from this set of unordered fragments.

1.2 The genome assembly task

If you want study an organism, knowing the complete genome sequence is very useful for a lot of tasks, such as as finding genes of interest, or study the sequence variations across a population ... Yet, the best sequencing technologies still provide reads that are at least 2 orders of magnitudes shorter than genomes. To understand the assembly problem, we provide a useful analogy which, to the best of our knowledge, has never been formulated before.

Imagine a crazy copyist monk. He is copying a book but he randomly chooses where he starts to copy. And he only copies small fragments of text at a time. The copyist monk makes errors, e.g. he would sometimes replace a symbol by another one, would skip a symbol, or would add a random symbol. We call these errors substitutions, deletions and insertions, respectively. Now imagine that there are multiple such copyist monks. They choose randomly where they begin to write. They can choose several times the same region of the book or never choose to copy a certain region. We refer by "coverage" the number of times a given chunk of the original book is copied. Coverage may significantly differ across the genome's regions. In this analogy, the book is the genome of the organism we want to study, and the copyist monks are our sequencer. The fragments of text are reads, and the operation to rebuild the book is assembly.

The assembly task can be seen as an ordering problem. We try to put the text fragments in the original book order, and merge common parts at the end. To carry out this ordering, we could randomly take a fragment of text, and search among all the others if there exists a fragment that begins with the end of the one we took. In other words, the prefix of the sought fragment corresponds to the suffix of the taken fragment. When we observe this phenomenon we say that the fragments overlap. This a key concept in assembly. Once we have found the best overlap (generally the longest) for a text fragment we can merge the two fragments into one and restart our search for a new fragment that overlaps with the one we just created. And so on until there are no more fragments. This presentation of how to perform assembly is very simple, and in fact it is what we will later refer to as the greedy algorithm. We will see more advanced assembly algorithms in chapter 3.

The genome assembly community, like any other scientific community, has its own set of concepts. A **read** designates a fragment of DNA produced by sequencer. An **overlap** occurs between two reads when the suffix of a read is similar to the prefix of another read. The length of the common part is called the length of overlap. A **contig** designates a sequence of DNA produced by an assembly

tool. The exact definition of what is a **contig** changes between each assembly tool. We can see in some publications the term **unitig**: we will not get into details here, but a common fact is that contigs are built from unitigs. A **scaffold** designates an ordering of contigs. Most of the times we cannot reconstruct each chromosome into a single contig. We describe some reasons for this fragmentation later. With external information such as restriction maps, linked reads, or targeted sequencing, one can order contigs and determinate approximately the number of bases in the gaps between contigs. Figure 1.2 gives a summary of how reads are processed to obtain an assembly.

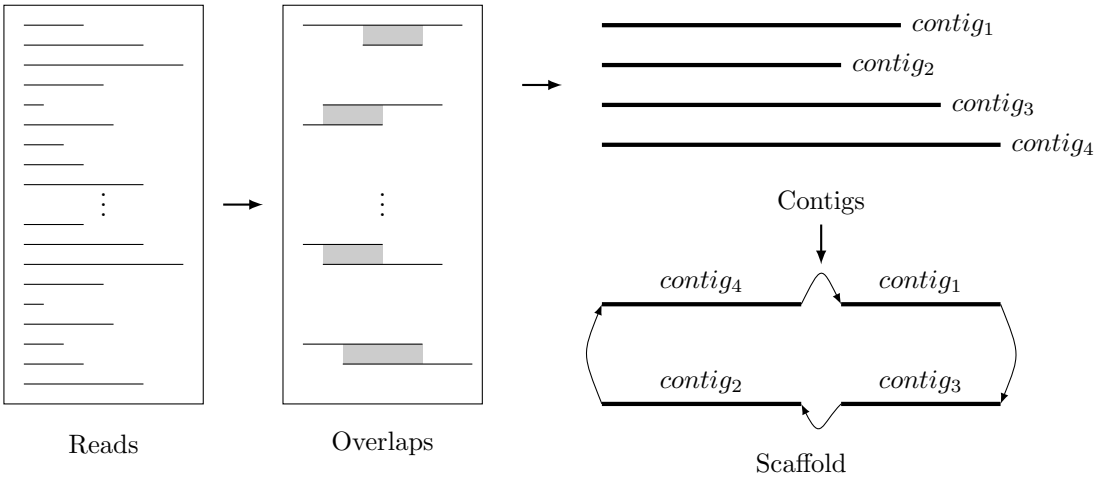


Figure 1.2: Schematic of DNA assembly. Each horizontal line represents a read, grey boxes represents overlaps found between reads. These overlaps are used to build contigs and finally these contigs are ordered into a scraffold.

1.3 Thesis outline

As stated above, latest sequencing technologies allow to sequence larger DNA fragments. One could think that the task of assembly becomes easier since we have to solve a puzzle with larger pieces. But this is not the case, as we will see in the following chapters. The main goal of this work is improve long-read genome assembly without creating a new assembly tool or modifying an existing one. The tools developed in this thesis can interface with existing long-read assembly tools or even with other bioinformatics analysis tools.

Chapter 2 addresses some of the key steps that are performed prior to assembly. The quality of the data provided to an assembler has a direct impact on the produced results. This chapter describes the state of the art of tools used to detect overlaps between DNA fragments. The first contribution is a discussion on how to compare such tools. The second contribution of the chapter is a paper that presents two tools we developed during this thesis [63]. `yacrd` detects and removes regions with very high error rates in reads. Experiments show that removing low quality regions from reads improve assembly tools results. `fpa` filters out uninformative overlaps in order to save disk space.

Chapter 3 presents a state of the art of several assembly methods, both from a theoretical point of view, and how they work in practice. This chapter introduces key concepts used in the next chapter.

Chapter 4 concerns some steps that occur after assembly. The first contribution is a blog post that presents the difficulties of evaluating the results of certain assembly tools that do not correct reads (or even polish contigs). The second contribution presents a tool for analyzing and improving assembly results, `KNOT`, that we developed during this thesis [62].

Chapter 5 will focus on various other scientific contributions: participation in the development of a graphical interface for genomic data analysis, participation in the contest is, and some work around 10X data.

Chapter 2

Preassembly

Two key aspects in long-read genome assembly are 1) the detection of overlaps between reads and 2) dealing with errors in reads. Computing overlaps is done prior to assembly can thus be considered as a preassembly task, even if some assembly pipelines compute overlaps several times. Computing overlaps is a hard task, even harder with high error rate reads. A number of tools have been designed in the last decade, with their own definition of what is an 'good' overlap. The section 2.1 gives a short overview of algorithmic ideas on which overlappers are designed. The next section 2.2 discusses about comparison of overlaps found by state-of-the-art overlappers for long read data.

After sequencing a usual task is to clean the set of reads, e.g.

- remove too short reads (less than 500 bp, 1000 bp or even 2000 bp)
- find and remove the sequencing adaptors (that is a short sequence added before DNA fragment, this short sequence is required for some biochemical consideration, but they can create trouble in assembly)

or perform some operations to improve the quality of reads, e.g.

- found highly erroneous regions of reads and replace them by more correct one, this operation is called *scrubbing*
- correct reads using information from other sequencing technology (this is called hybrid correction) or with same technology, this operation is called *correction*

Cleaning preprocessing intends to improve the quality of the assembly or to help the task of assembly (e.g. by reducing the number of false overlaps). As overlapping tools are not aware of what the user will do with the computed set of overlaps, all information is reported (it's a good point). But one has to remember that the number of overlaps for a usual sequencing experiment is very large. Storing them may require more than terabyte for some large dataset. In section 2.3.1 we introduce in more details bottlenecks and the solution we have proposed.

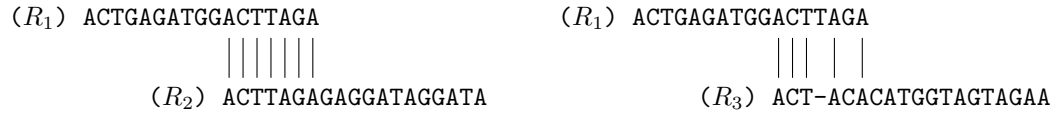
Correction and scrubbing seeks to perform the same target: reduce the error rate of reads. Scrubbing works on large region (around ten or hundred bases) while correction works at the level

of one base. This difference of scale implies different requirements in terms of computation time and memory usage.

Our work on overlap selection and on scrubbing tools was merged in a paper presented in section 2.4.

2.1 Overlaps and their impact on assembly

As we defined in the introduction, when two sequences share a common substring, we say that they overlap or that one of them maps on the other see 2.1a. It is possible for sequences to share a common substring just by chance (because of the 4-letter alphabet) but the probability of this event decreases when the length of the common substring increases. Intuitively, this probability gets smaller as common substring gets longer. If the reads does not contain sequencing errors, the only criteria to evaluate whether a common substring is "true" or not could be the length of this substring. However, the number of errors in the sequence of readings breaks this paradigm and forces us to integrate the errors when assessing the quality of an overlap. Figure 2.1b show an overlap with two mismatch. There are two base pairs that do not match between the two sequences - knowing if this overlap is "true" or not isn't obvious.



(a) R_1 shares 7 bases at its end with the beginning of R_2, without any error

(b) R_1 shares 5 bases at its end with R_3, with one substitution and one deletion

Figure 2.1: When reads don't contain error, overlaps look like (a), but sequencing technologies make errors and the overlap present in (b) can be a true overlap.

In this document we distinguish two tasks in similarity search between two sequences:

- mapping: one tries to find the position of a read in a larger sequence (e.g. comparing different datasets, experiments from different sequencing generations, or between the reads and an assembly, against a reference genome)
- overlapping: one tries to find which reads share a common substring with other reads (e.g. finding common substrings in the same dataset)

Even if mapping and overlapping can be seen as different tasks, one could observe that the same tools, and underlying algorithmic, can be used to solve both tasks.

The seed-and-extend strategy. Search of similarity between two or more DNA sequences has many links to plain text search. Seed-and-extend is an approach used by many tools to find similar sequences between a target (e.g. a reference genome) and a query (e.g. a read). The idea is to find a high similar subsequence (often exact), namely the seed (or anchor), and then to extend this seed to have a larger common subsequence. Tools that implement this approach usually create an index of

the target. This index need to answer to a simple question: is a given subsequence exist in target and at which position. Each query is processed and its substrings are searched in the index. This gives a set of seeds that can be extended through alignment techniques such as dynamic programming. If the alignment score reaches a given threshold, a hit is reported.

Many tools for mapping and overlapping use seed-and-extend strategy. The most popular is `Blast` [3, 4]. Specific tools dedicated to sequencing are `BWA` [52] or `blasr` [15]. Implementation details of indexes, size and number of anchors change between tools. For example `BWA` or `blasr` use a FM-index [26] to perform anchor search.

The seed-only strategy. With NGS technology development more and more data had to be processed and the seed-and-extend strategy was replaced by a seed-only strategy. Indeed, the extension step is still very time-consuming. With the seed-and-extend strategy, an overlap is scored by its length and the number of errors in the alignment. With the seed-only strategy we don't have an alignment. The overlap is thus scored using the number of seeds and their positions.

This strategy was used in `SGA` [98] assembly tools, during overlapping step `SGA` search exact overlap between low error read, by search a substring at end of read in a FM-index.

Specificity of long-reads (longer reads, high error rate) has relaunched this research field. Chu et al. produce an interesting review about some of third-generation overlap search in [22], discussed in next section. We can cite `Hisea` [39], `Daligner` [73], `MHAP` [44] and `Minimap2` [56, 57] as overlapping tools they use this strategy to found overlap between thrid generation overlap. We will give some details on `MHAP` and `Minimap2` in section 3.

For third generation reads, the length of the reads and the large number of errors make the choice of algorithm parameters even more complicated, particularly concerning how we choose the seed and length of seeds. But by removing the extend step the computation time was reduce and help to manage the high error rate of thrid generation reads.

On the importance of overlaps. As overlaps are the basic components to reconstruct the original sequence, a missing overlap may lead to a wrong assembly (entire pieces of the genome inverted) or to a high number of contigs. In [22], Chu et al. compare the state of the art third generation sequencing read overlappers on simulated datasets and on real datasets. A drop in the accuracy and recall of these algorithms can be observed between real and simulated data 2.1.

	Pacbio		Nanopore	
	Simulated	Real	Simulated	Real
Sensibility	88.9^{m} - 92.4^{d}	59.6^{m} - 83.8^{d}	90.4^{g} - 95.2^{b}	88.9^{b} - 92.9^{d}
Precision	81.9^{b} - 96.5^{g}	79.8^{h} - 96.5^{b}	75.1^{b} - 99^{m}	73^{b} - 95.4^{m}

Table 2.1: ^{m}`Minimap`, ^{d}`Daligner`, ^{g}`GraphMap`, ^{b}`BLASR`, ^{h}`MHAP`

In a blog post "State-of-the-art long reads overlappers comparison" [1] we take the same data as [22] but we didn't care if the overlappers found 'right' or 'wrong' overlaps. Instead we searched for comparing overlaps sets to decide whether or not overlappers compute the same overlaps. There

[1] https://blog.pierre.marijon.fr/long-reads-overlapper-compare/

were differences large enough to justify the idea of creating a kind of 'reconciliation' tool that merge information from several overlapper. This blog post was presented at the poster session of JOBIM (*Journée Ouverte de Bioinformatique & Mathematique*) 2018.

2.2 State-of-the-art long reads overlapper-compare

Originaly publish in: https://blog.pierre.marijon.fr/long-reads-overlapper-compare/

Author: Pierre Marijon

2.2.1 Introduction

In 2017, Chu et al. wrote a review [22] to present and compare 5 long-read overlapping tools, on 4 datasets (including 2 synthetic ones). This paper is very cool and clear. The authors compare overlappers with respect to peak memory, wall clock time, sensitivity and precision. Table 2 from this paper presents sensitivity and precision:

Table 2. An overview of sensitivity and precision on simulated and real error-prone long read datasets. In both the PB and ONT simulated datasets, the best values, shown in **bold** face, are statistically significantly better than the other values. We derived these values from the best settings of each tool (according to the best F1 score) after a parameter search. We calculated confidence intervals for the sensitivity, specificity and F1 scores using three standard deviations around the observed values. In the worst case, the error never exceeded ±0.1%, ±0.1% and ±0.2% respectively.

Tool	Simulated PB *E. coli*			Simulated ONT *E. coli*			PB P6-C4 *E. coli*			ONT SQK-MAP-006 *E. coli*		
	Sens. (%)	Prec. (%)	F1 (%)	Sens. (%)	Prec. (%)	F1 (%)	Sens. (%)	Prec. (%)	F1 (%)	Sens. (%)	Prec. (%)	F1 (%)
BLASR	91.0	81.9	86.2	**95.2**	75.1	84.0	66.0	**96.5**	78.3	89.9	73.0	80.6
DALIGNER	**92.4**	91.9	92.1	94.9	97.6	95.9	**83.8**	85.8	**84.8**	**92.9**	91.0	91.9
MHAP	91.5	88.0	89.8	**95.1**	86.5	90.6	79.8	79.8	79.8	91.2	82.0	86.3
GraphMap	90.1	**96.5**	**93.1**	90.4	96.0	93.1	71.7	94.0	81.4	90.6	93.4	92.0
Minimap	88.9	94.8	91.8	94.6	**99.0**	**96.7**	59.6	83.8	69.7	91.2	**95.4**	**93.2**

Figure 2.2: Table 2 of [22]

Overlappers show better results on synthetic datasets than on real data. We can observe an important loss of sensitivity: 59.6-83.8% on the Pacbio real dataset, compared to 88.9-92.4% on the simulated data.

So, ok, overlappers dont't achieve perfect sensibility, but **do they miss the same overlaps**?

2.2.2 Materials & Methods

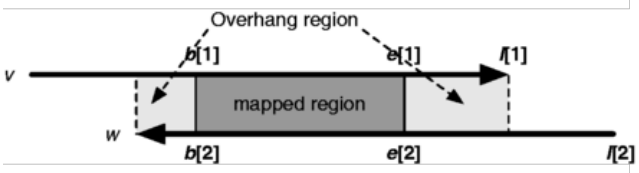
2.2.2.1 Datasets

I selected the two real sequencing datasets in Chu et al., because they had the highest variance in sensitivity, so we can see the most extreme effects in how long-read overlappers possibly find different overlaps.

2.2.2.2 What is an overlap

I will not bore you with formal definitions :)

We will consider 3 type of overlaps, according to Algorithm 5 presented in the minimap publication[56]



Algorithm 5: Mapping classification

Input: Read length l, mapping begin coordinate b and mapping end e of the two reads; max overhang length o (1000 by default) and max overhang to mapping length ratio r (0.8 by default).

Output: hashed p-bit integer

Function ClassifyMapping($l[2], b[2], e[2], o, r$) **begin**

$overhang \leftarrow \min(b[1], b[2]) + \min(l[1] - e[1], l[2] - e[2])$

$maplen \leftarrow \max(e[1] - b[1], e[2] - b[2])$

if $overhang > \min(o, maplen \cdot r)$ **then**

  return INTERNAL_MATCH

else if $b[1] \leq b[2]$ **and** $l[1] - e[1] \leq l[2] - e[2]$ **then**

  return FIRST_CONTAINED

else if $b[1] \geq b[2]$ **and** $l[1] - e[1] \geq l[2] - e[2]$ **then**

  return SECOND_CONTAINED

else if $b[1] > b[2]$ **then**

  return FIRST_TO_SECOND_OVERLAP

else

  return SECOND_TO_FIRST_OVERLAP

Figure 2.3: Algorithm 5 in minimap and miniasm article by Heng Li

Internal match: Just a short similarity localized in the middle section of both reads, which is probably due to a repetitive region and not a true overlap

Containment: One read is completely contained in another

Classic overlap: Deemed a regular suffix-prefix overlap

We will check the results of overlappers, and for each entry that isn't an internal match nor an containment overlap, we store the pair of reads as elements of the set of all overlaps found by the overlapper.

2.2.2.3 Overlappers

We used:

- graphmap v0.5.2 [100]
- hisea commit: 39e01e98ca [39]
- mhap 1.6 and 2.1 [9]
- minimap 0.2-r124 [56]
- minimap2 2.10 [57]

We used parameters recommended by Chu et al. and default parameters for HISEA.

2.2.2.4 Venn diagram generation

We used a Python script to parse the output file of each overlapper, filter overlaps, generate a Venn diagram, and compute the Jaccard index. All scripts and steps to reproduce this analysis are available in this repository.

2.2.3 Results

2.2.3.1 Nanopore real data

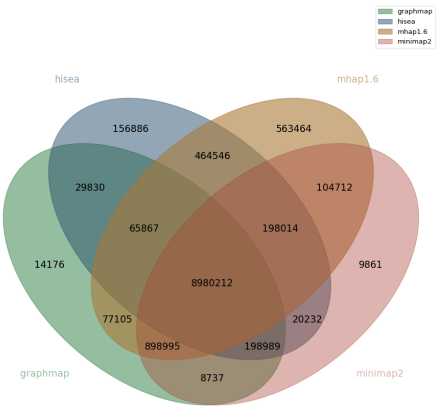


Figure 2.4: Venn diagram for nanopore real dataset

In the center of the above diagram we have the number of overlaps found by all overlappers. We call this set the *core overlaps*. Here for this dataset, core overlaps contain 8,980,212 overlaps. Around this center, we highlight some of the largest disparities between overlappers:

dataset composition	number of overlaps	% of core overlaps
core overlaps - hisea overlaps	898,995	10.01 %
hisea overlaps ∩ mhap overlaps	464,546	5.17 %
core overlaps - mhap overlaps	198,989	2.21 %
core overlaps - graphmap overlaps	198,014	2.21 %

In addition, out of the 11,352,915 overlaps found by mhap, 4.96 % of these are found only by this overlapper. For hisea, the corresponding value is 1.55 % (out of 10,114,576 overlaps).

	mhap	minimap2	graphmap	hisea
mhap		0.88	0.85	0.82
minimap2	0.88		0.94	0.84
graphmap	0.85	0.94		0.83
hisea	0.82	0.84	0.83	

The above matrix shows the Jaccard similarity coefficient (cardinality of intersection divided by cardinality of union) between pairs of overlappers.

2.2.3.2 Pacbio real data

For the Pacbio dataset, core overlaps contain 3,407,577 overlaps. Other large disparities between overlappers are:

Out of all overlaps found by minimap2 (5,640,643), 9.54% of these overlaps are found only by this overlapper, for mhap the corresponding value is 5.98% (out of 5,336,610 overlaps).

Figure 2.5: Venn diagram for pacbio real dataset

dataset composition	number of overlaps	% of core overlaps
core overlaps - graphmap overlaps	713,161	20.93 %
minimap2-only overlaps	538,118	15.79 %
mhap overlaps ∩ minimap2 overlaps	503,431	14.77 %
core overlaps - hisea overlaps	352,376	10.44 %
mhap-only overlaps	319,744	9.38 %

Again the above matrix shows Jaccard similarity coefficient.

2.2.3.3 Comparison across versions

At first we used mhap 2.1, using the same parameters as in Chu et al. But actually, Chu et al. used mhap 1.6. This version change yielded surprising results: many more overlaps were found only by mhap 2.1. Here is a comparison between the two executions of mhap 1.6 and 2.1 using the same command-line parameters, in terms of shared and exclusive overlaps.

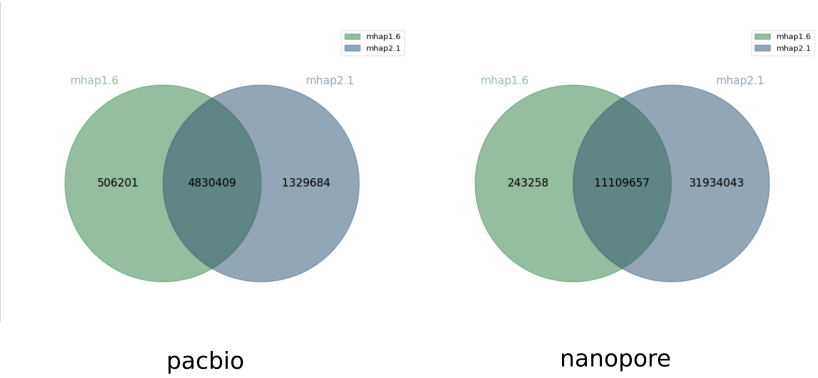


Figure 2.6: Jaccard similarity 0.72, 0.26

mhap 2.1 found many more overlaps than mhap 1.6. But it turns out that this is because mhap 1.6 calculates a similarity score between reads and mhap 2.1 calculates a distance between reads, the

	mhap	minimap2	graphmap	hisea
mhap		0.83	0.70	0.76
minimap2	0.83		0.67	0.74
graphmap	0.70	0.67		0.74
hisea	0.76	0.74	0.74	

meaning of the --threshold option is different between the two versions, so we should have not used the same parameter value for both versions (thanks to Sergey Koren for pointing this out). This explains why a user may get significantly different results between the two versions, when running them carelessly with identical parameters. Below, we plot the Venn diagrams of overlaps found only by mhap 1.6 with --threshold 0.02 for pacbio and --threshold 0.04 (like Chu. *et al*) and only by mhap 2.1 with --threshold 0.75 for pacbio and --threshold 0.78 for nanopore.

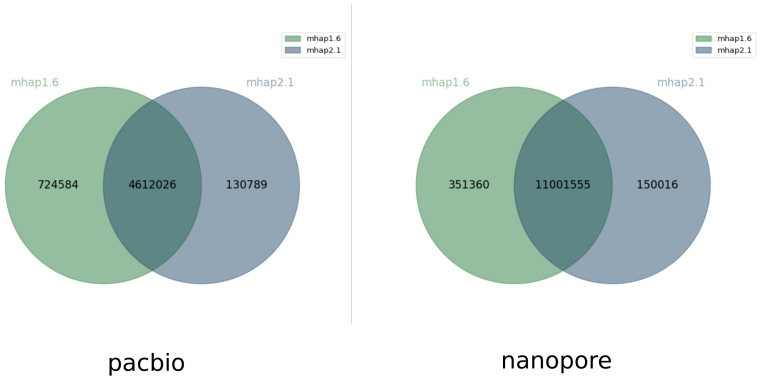


Figure 2.7: Jaccard similarity 0.84, 0.96

Both software find roughly the same set of overlaps, with the trend that mhap1.6 tended to find a bit more (it would be interesting to evaluate whether those were correct or wrong overlaps).

And another comparison between minimap and minimap2:

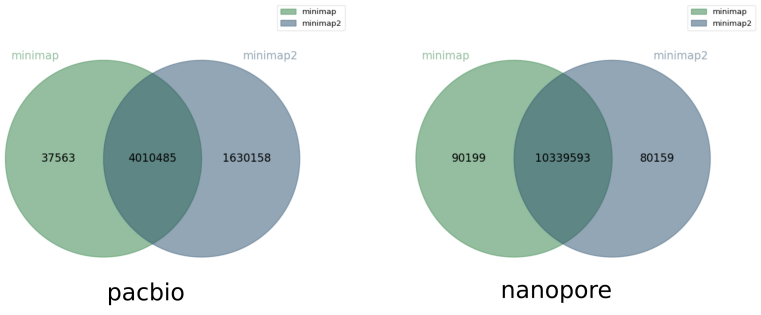


Figure 2.8: Jaccard similarity 0.71, 0.98

For the pacbio dataset, minimap2 finds significantly (1.6M) more overlaps than minimap (which found 4M overlaps). But for the nanopore dataset, both software roughly agree.

2.2.4 Conclusion

Overlapper tools behave quite similarly, but on real pacbio datasets sensibility, precision, and the set of overlaps found across tools can be very different. Such a difference can also exist between two versions of the same tool.

Comparison of overlappers based on a quantitative measurement (sensitivity, precision) is useful but isn't perfect: two tools with the same sensitivity for a given set could still detect a different set of overlaps, see e.g. mhap and minimap2 for the nanopore set.

Some publications use quality of error-correction, or results of genome assembly, as quality metrics to compare overlappers. It's a good idea but correction and assembly tools make additional choices in the overlaps they keep, and it's not easy to relate assembly or error-correction imperfections and wrong or missed overlaps.

From our tests, there is no clear best overlapper software so far.

It could by interesting to study whether certain tools have a bias when finding overlaps, linked to e.g length of reads, mapping length, error rate, %GC, specific kmer composition, etc ... A study like this could possibly reveal some intrinsic properties of the algorithms used in overlappers.

Is it a good idea to create a reconciliation tool for overlappers? We note that the correction and assembly tools seek to reduce the amount of overlaps they use, through e.g. graph transitivity reduction, Best Overlap Graph, the MARVEL approach (Supplementary information of [77]).

Acknowledgment

- Sergey Koren
- Rayan Chikhi
- Jean-Stéphane Varré

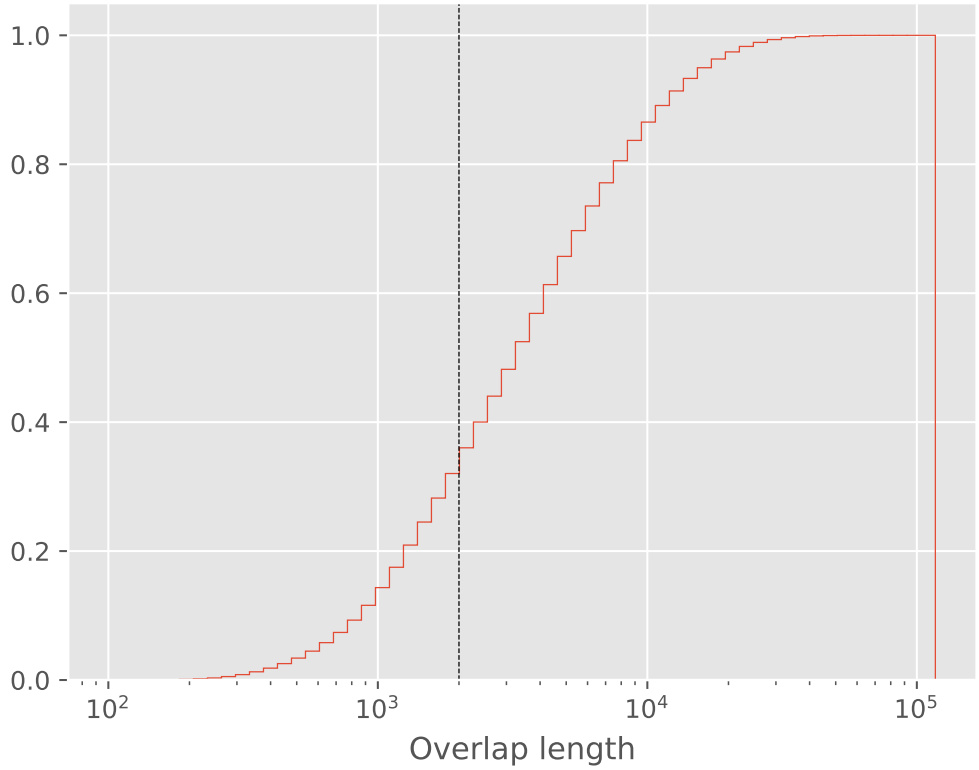


Figure 2.9: Histogram of overlap lengths found by `Minimap2`, the black line represents the `Miniasm` overlap length threshold. The fasta file weight 3.1 Go, complete PAF file generate by `Minimap2` weight 5.5 Go, without overlap lower than 2000 bases the weight is reduced to 3.7 Go.

2.3 Improving assembly by filtering out overlaps and scrubbing

2.3.1 Improve genome assembly efficiency by reducing the quantity of information

Error in third generation reads make it more difficult to found overlaps between reads. Current techniques attempt to optimize results on real data [22]. Actually, a key observation is that within the overlaps found by state-of-the-art tools, not all of them are useful to downstream analysis. For example `Miniasm` keeps only end-to-end overlaps, and `Canu` keeps only the two longest end-to-end overlaps for each read (see 3 for more details).

Figure 2.9 shows a histogram of overlap lengths found by `Minimap2` on *E. coli* Nanopore dataset (acession number SRR8494940): 33 % of overlaps are shorter than 2000 bases. By default `Miniasm` ignores overlaps shorter than 2000 bases that is if we run a basic `Miniasm` pipeline, 33% of the overlap will not be used but they are written on the disk. There is definitively room for improvement. Can

we filter overlapping information with positive (or at least no negative) impact on assembly results ? One may hope to at least decrease the disk space and may be to increase the speed of assembly.

In section 2.4 we present `fpa` (for Filter Pairwise Alignment), our solution to filter out useless overlaps. An overlaper output can be piped directly to `fpa`. `fpa` can apply several filters based on length of read, length of overlap, type of overlap, read name. Some simple `fpa` filters reduce the computation time of assembly without effect (or a small positive effect) on assembly.

2.3.2 Read scrubbing: an alternative to read correction

Assembly tools are based on reads. If your reads are bad, your assembly will be bad. To continue on the analogy given in the introduction, you probably cannot reconstruct a book if crazy monks gave you only fragments with half of the letters being erroneous. Correction of reads, with a mix of sequencing technologies or with a single technology, can help to get better reads. But actually, correction tools have an important cost in term of computation time and memory usage. Moreover it's hard to distinguish mutations (e.g. true SNPs) from sequencing errors, and sometimes interesting mutations are consider as errors and are corrected (thus removed).

The pre-processing correction step is particularly important for long-reads data because of high error rates that can lead to more errors and misassemblies. Tools like `Mecat` [112], `CONSENT` [68] uses overlap information to pick reads that share same sequences and build a consensus from the alignements induced by overlaps. A similar task, called polishing, is run after assembly, as a post-processing task. Reads are mapped against assembled contigs and contig sequences is corrected using reads, we can cite `Racon` [104] and `CONSENT`.

The more a read contains errors, the more the correction step require reads. But the sequencing depth is not homogeneous. Thus the corrector will be more or less effective depending on regions and the depth of coverage thereof. If the sequencing depth is too low, the correction may discard some reads. To solve this problem it is necessary either to work without correction or to return to raw reads.

Correction of reads before assembly can generate some trouble in assembly by remove some important information. At the best of our knowledge the only one reads corrector that tries to keep the heterozygotie during correction is falcon [20]. Heterozygotie is very useful to understand genetic diversity in population or some genetic diseases. Another example concerns genomes that contain approximate repeats. The correction step tends to correct both region in order to make them identical. By the way, correction creates a repetition that cannot be solved by the assembler although regions could be distinguished prior to correction.

Nevertheless, long-reads still contains very low quality region [74] that can lead to fragmented assembly [109]. It is thus necessary to filter out thos regions. An alternative to correction can be scrubbing: one removes only very low quality region and keep all other information.

To found and remove this very low quality region and read we created `yacrd` (for Yet Another Chimeric Read Detector). `yacrd` uses self overlapping information to compute a coverage curve and identifies regions of low coverage. We hypothesise taht such low coverage regions are of low quality (see section 2.4 for more details on this tool).

Our paper "`yacrd` and `fpa`: upstream tools for long-read genome assembly" presents two tools, `yacrd`, and `fpa`. `yacrd` focuses on the detection and elimination of very poor quality regions. `fpa` focuses on filtering 'useless' overlaps.

2.4 `yacrd` and `fpa`: upstream tools for long-read genome assembly

Currently under review but avaible in bioRxiv 10.1101/674036

Author: Pierre Marijon Rayan Chikhi and Jean-Stéphane Varré

2.4.1 Abstract

Motivation: Genome assembly is increasingly performed on long, uncorrected reads. Assembly quality may be degraded due to unfiltered chimeric reads; also, the storage of all read overlaps can take up to terabytes of disk space. **Results:** We introduce two tools, `yacrd` and `fpa`, to respectively perform chimera removal/read scrubbing, and filter out spurious overlaps. We show that `yacrd` results in higher-quality assemblies and is two orders of magnitude faster than the best available alternative.
Availability: https://github.com/natir/yacrd and https://github.com/natir/fpa
Contact: pierre.marijon@inria.fr
Supplementary information: Supplementary data are available online.
Acknowledgements: This work was supported by Inria and the INCEPTION project (PIA/ANR-16-CONV-0005) and the University of Lille HPC facility. The authors thank Maël Kerbiriou for algorithmic help.

2.4.2 Introduction

Third-generation DNA sequencing (PacBio, Oxford Nanopore) is increasingly becoming a go-to technology for the construction of reference genomes (*de novo* assembly). New bioinformatics methods for this type of data are rapidly emerging.

Some long-read assemblers perform error-correction on reads prior to assembly. Correction helps reduce the high error rate of third-generation reads and make assembly tractable, but is also a time and memory-consuming step. Recent assemblers (e.g. [56, 87] among others) have found ways to directly assemble raw uncorrected reads. Here we will therefore focus only on **correction-free assembly**. In this setting, assembly quality may become affected by e.g. chimeric reads and highly-erroneous regions [74], as we will see next.

The `DASCRUBBER` program [75] introduced the concept of read "scrubbing", which consists of quickly removing problematic regions in reads without attempting to otherwise correct bases. The idea is that scrubbing reads is a more lightweight operation than correction, and is therefore suitable for high-performance and correction-free genome assemblers.

`DASCRUBBER` performs all-against-all mapping of reads and constructs a pileup for each read. Mapping quality is then analyzed to determinate putatively high error rate regions, which are replaced

by equivalent and higher-quality regions from other reads in the pileup. `miniscrub` [47] is another scrubbing tool that uses a modified version of `Minimap2` [57] to record positions of the anchors used in overlap detection. For each read, `miniscrub` converts anchors positions to an image. A convolutional neural network then detects and removes of low quality read regions.

Another problem that is even more upstream of read scrubbing is the computation of overlaps between reads. The storage of overlaps is disk-intensive and to the best of our knowledge, there has never been an attempt at optimizing its potentially high disk space.

In this paper we present two tools that together optimize the early steps of long-read assemblers. One is `yacrd` (for Yet Another Chimeric Read Detector) for fast and effective scrubbing of reads, and the other is `fpa` (for Filter Pairwise Alignment) which filters overlaps found between reads.

2.4.3 Materials & Methods

Similarly to `DASCRUBBER` and `miniscrub`, `yacrd` is based on the assumption that low quality regions in reads are not well-supported by other reads. To detect such regions `yacrd` performs all-against-all read mapping using `Minimap2` and then computes the base coverage of each read. Contrarily to `DASCRUBBER` and `miniscrub`, `yacrd` only uses approximate positional mapping information given by `Minimap2`, which avoids the time-expensive alignment step. This comes at the expense of not having base-level alignments, but this will turn out to be sufficient for performing scrubbing. Reads are split at any location where coverage drops below a certain threshold (set to 4 by default), and the low-coverage region is removed entirely. A read is completely discarded if less than 40% of its length is below the coverage threshold. `yacrd` time complexity is linear in the number of overlaps.

`yacrd` performance is directly linked to the overlapper performance. We tuned `Minimap2` parameters (especially the maximal distance between two minimizers, `-g` parameter) to find similar regions between reads and not to create bridges over low quality regions (see Supplementary Section B.3). `yacrd` takes reads and their overlaps as input, and produces scrubbed reads, as well as a report.

`fpa` operates between the overlapper and the assembler. It filters out overlaps based on a highly customizable set of parameters, such as overlap length, length of reads names, etc. `fpa` can remove self-overlaps, end-to-end overlaps, containment overlaps, internal matches (when e.g. two reads share a repetitive region) as defined in [56]. `fpa` supports the PAF or BLASR m4 formats as inputs and outputs, with optional compression. `fpa` can also rename reads, generate an index of overlaps and output an overlap graph in GFA format.

`yacrd` and `fpa` are evaluated on several datasets (details provided in Supplementary Section B.1), and here we highlight their performance on two of them: *H. sapiens* chromosome 1 Oxford Nanopore (ONT) ultra-long reads, and *C. elegans* PacBio reads. All tools were run on a single cluster node with recommended parameters (see Supplementary Section B.2). Scrubbed reads were then assembled using both `Miniasm` and `wtdbg2` with recommended parameters for each sequencing technology.

		H. sapiens chr1 (ONT ultra-long R9.4)			*C. elegans* (Pacbio P6-C4)		
		raw	dascrubber	yacrd	raw	dascrubber	yacrd
Reads	# reads	1,075,867	819,798	1,044,848	740,776	660,766	751,750
	Relative # bases	1.00	0.71	0.80	1.00	0.84	0.84
	N50	10,568	9,858	9,520	16,572	15,667	15,845
	# chimera	25,888	6 %	20 %	71,704	13 %	21 %
	Time		**3 days 2 hours**	**27 mins**		**1 day 20 hours**	**33 mins**
`Miniasm`	# contigs	184	184	394	226	131	154
	NGA50	96,225	410,37	453,748	432,112	544,677	440,776
	Asm/Ref size	81 %	78 %	81 %	113 %	108 %	110 %
	# misassemblies	1,745	209	432	1,396	754	1,015
`wtdbg2`	# contigs	810	496	485	139	100	122
	NGA50	1,513,450	545,902	1,482,513	565,278	578,041	593,039
	Asm/Ref size	87 %	80 %	84 %	106 %	104 %	106 %
	# misassembly	1,316	177	582	614	485	577

Table 2.2: Performance of `yacrd` compared to `DASCRUBBER` on an ONT and a PacBio dataset. Relative #bases indicates the proportion of raw read bases kept after scrubbing. # chimera indicates the number of chimeric reads detected in the dataset using `Minimap2` (see Supplementary Section B.4) and the proportion of remaining chimeric reads after scrubbing. NGA50 is the N50 of aligned contigs, and # misassemblies are the number of misassemblies, both metrics were computed by QUAST [30]. Asm/Ref size indicates the relative length of the assembly divided the reference length.

2.4.4 Result & Discussion

Table 1 compares the results of `yacrd` and `DASCRUBBER`. We also evaluated `miniscrub` (see Supplementary Section B.2 and B.5), but its memory usage exceeded 256 GB on the two datasets of Table 1.

The main feature of `yacrd` is that its total execution time, which is essentially that of `Minimap2`, is two orders of magnitude faster than `DASCRUBBER`. We next evaluate whether running `yacrd` results in higher-quality reads and assemblies. `yacrd` removes 20-27% of the bases in raw reads, comparably to `DASCRUBBER`. Both scrubbers significantly reduce chimeras: only 6-13% of those in raw reads remain with `DASCRUBBER` and 18-20% with `yacrd`. The impact of removing chimeras is directly seen on assembly metrics: both scrubbers produce significantly less misassemblies with `Miniasm` and `wtdbg2` than with direct assembly of raw reads. Both `yacrd` and `DASCRUBBER` resulted in increased contiguity (NGA50) with `Miniasm`, and equivalent (or significantly degraded for `DASCRUBBER`) contiguity with `wtdbg2`, and comparable assembly lengths.

On ONT reads, `DASCRUBBER` reduces the number of misassemblies by a factor of 2-3 more than `yacrd`. However, given that all assemblies in Table 1 completed in less than an hour and `DASCRUBBER` took 3 days, running this tool on larger datasets would become a significant performance bottleneck. In Supplementary Section B.3 we examine the behavior of `yacrd` across its parameter space. We observe that different parameters worked best for different datasets, one of which is actually a parameter for `Minimap2`.

`fpa` reduced the size of reads overlap file (PAF file produced by `Minimap2`) by 40-79% on the evaluated datasets, without any significant effect on quality assembly. As a consequence this reduces the memory usage of `Miniasm` by 13-67%. Other performance metrics are presented in Supplementary Table B.5.

Finally, we examine the effect of combining both `yacrd` and `fpa`. We propose a pipeline based

on `Miniasm` (Supplementary Section B.7) and show that it results in improved assembly contiguity, comparable assembly size, less mismatches and indels, less misassemblies, at the cost of a reasonable increase in running time (around 2x).

2.5 Chapter conclusion

In this chapter we have proposed a benchmark of overlappers, a filtering tool for these overlap and a scrubbing tool.

The blog post on overlapping tools comparison demonstrates that they do not found same overlaps. We should be able to improve the quality of the overlaps we found between reads by combining results from several tools. This is the idea of an overlap consensus generator. In the blog post we considered that if overlapping tools found an overlap between two reads, the overlap should be roughly the same. Actually this is not true. Considering two reads `A` and `B` and three overlapping tools, it's possible that:

- the first tool find that the end of read `A` overlaps the beginning of read `B`
- the second tool find that the end of read `B` overlaps the beginning of read `A`
- the third tool find that reads `A` and `B` share an internal match

A number of other situations can occur. If we want to build an overlap consensus generator we need to found a method able to say this two overlap found by two different overlapping tools, concern the same region of read `A` and the same region of read `B`, we can increase our confidence in this overlap is a *true* overlap and it's is between this region of `A` and this region of read `B`. Or all overlapping tools found an overlap between read `C` and `D` but all this overlap concern different region of `C` and `D`, we can say they are probably no overlap between `C` and `D`. A work has been made in the context of a student project I supervised (PFE - *Projet de Fin d'Étude* End of Study Projects by Yann Grabe). He built a tool that computes a consensus of several overlap files. By comparing overlaps, i.e. computing overlaps between read overlaps, the tool computes a confident score on each read overlap by evaluating the number of overlapping tools that found the same read overlap. For the moment this tool is only a prototype and would still require a lot of work before it can be finalized.

Chapter 3

Long reads assembly tools state of the art

In the previous chapter we have seen how we can clean data before running assembly. In this chapter we present methods to perform an assembly and how these methods are applied on long-read assembly tools. We selected some tools for which we give a detailed description because methods and algorithms used are representative on how other tools work. In addition, these tools are recognized by the community for their quality. We can see that assembly tools can be split in steps. Assembly tools share similar steps. But we can observe that in the most recent assemblers (see section 3.7 and 3.8), the interdependence between each step of an assembly pipeline is more and more important.

3.1 *Greedy* assembly algorithm

The **Greedy** assembly algorithm is the first type of assembly tools, used on Sanger data. For example, GigAssembler was used to assemble the first human genome [38]. Algorithm 1 presents the general idea of how the **Greedy** algorithm works.

The BEST_OVERLAP function is the main part of algorithm. The best overlap is the larger one or the overlap with less error. Each algorithm have its own method.

Algorithm 1 A greedy assembly

```
1: function GREEDY(reads)                          ▷ reads is a set of read
2:     choose r1 in reads
3:     sequence ← r1
4:     while r2 ← BEST_OVERLAP(r1) do   ▷ BEST_OVERLAP() is a function for r1 they get read r2
   the best overlap for read r1 in reads
5:         CONCATENATE(sequence, r2)
6:         DROP(r1, reads)
7:         r1 ← r2
8:     end while
9: end function
```

Moreover the **Greedy** algorithm, by focusing on the local problem, which overlap is the best

for this read, cannot manage repetition. Genome contains many repetitions, like in a book some words are used several times or a whole part of a sentence can be present multiple times.

Figure 3.1 presents a case where reads R_0, R_1 and R_2 contain a repetition. R_0 has two possible overlaps: if overlap with R_1 is chosen, the assembly sequence matches with the green path; if overlap with R_2 is chosen, the assembly sequence matches with the red path. Each of these paths corresponds to a different region of the original genome. We can't know which path is the good one and we didn't see the repetition. So assembly tools based on **Greedy** algorithm can produce many misassemblies.

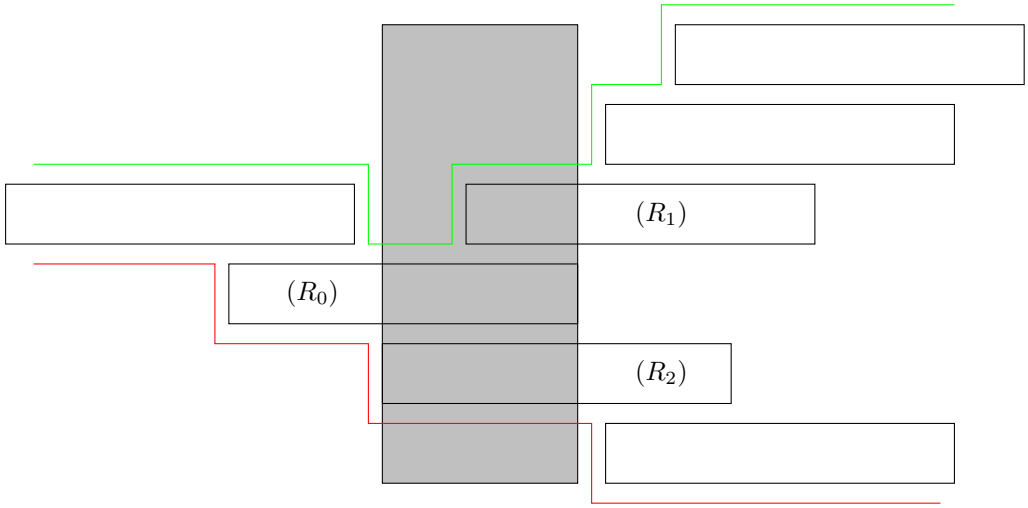


Figure 3.1: Each black box is a read, the grey box marks the position of a repetition. The beginning of R_1 and R_2 are in repetition: they share the same beginning but do not match at their ends. This repetition creates an ambiguity in assembly.

3.2 Overlap Layout Consensus

An alternative to the **Greedy** approach is the Overlap Layout Consensus (`OLC`). We can find a first definition of `OLC` in [71] in 1995. The most popular assembly pipeline based on OLC is probably Celera [66, 69]. This approach is based on a graph where a read is a node and we build an edge between nodes if reads share an overlap. Figure 3.2, presents the `OLC` corresponding to the overlap seen in 3.1.

If we reuse analogy we introduce in section 1.2 we can see this graph as an ordering of the chapters of a book provided by a crazy copyist monk. An edge indicates this piece of text was before that piece of text in the original book.

As we can see in Figure 3.2 a repetition creates a fork in `OLC`, a node with two successors. It's easy to detect this case in the graph and stop this contig construction. The assembly result of this graph is 3 sequences with white nodes, green nodes and red nodes. The assembly is more fragmented than with the *Greedy* algorithm but does not contain any misassembly.

By analyzing the graph, we will be able to detect the paths without branching node and to reconstruct the corresponding sequence by merging the sequences present in the graph.

`OLC`-based tools help to avoid misassemblies but the search for overlaps between reads is still time-expensive. The graph construction consumes a lot of memory, and more cleaning steps and graph

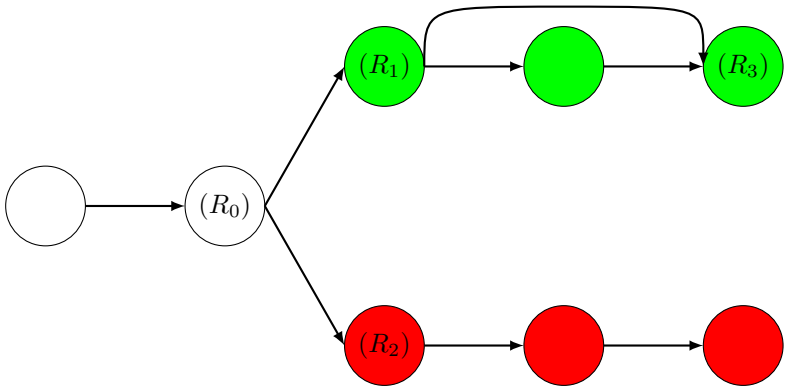


Figure 3.2: Each node is a read and an edge is built between two reads if they share an overlap.

analysis are expensive in computation time compared to a **Greedy** approach.

3.3 Algorithms and heuristics to simplify assembly graphs

The graph structure was useful to get a comprehensive view of all the information provided by reads, but having too much information can create problems, slow down the assembly tools and increase their costs in memory or at worst lead to misassembly or to unnecessary fragmentation of the assembly.

3.3.1 Transitive edge

In Figure 3.2 you can notice the edge from R_1 to read R_3, this overlap is exact. We can find an overlap between R_1 and R_3. But this edge does not provide new information, we know R_1 is before R_3, this edge is called a transitive edge. We can give a more formal definition of a transitive edge: in a directed graph, if we have a set of edges (a, b) (b, c) and (a, c), the edge (a, c) is transitive.

Myers proposed in [70] another assembly graph, the **string graph**, which is an overlap graph with no transitive edge. By reducing the number of edges in the graph, the string graph simplifies the traversing of the graph and decreases the memory impact.

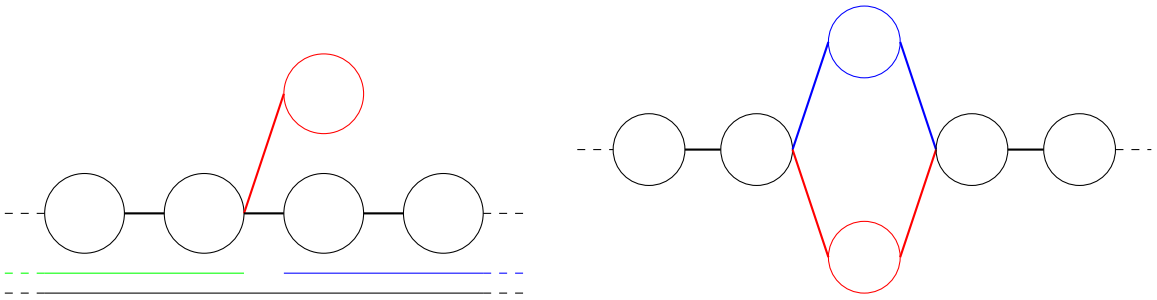
With string graphs, we just need to follow a simple path (a path in which each node has only one successor) to build assembly without misassembly.

3.3.2 Contained reads

In third generation technology, the crazy copyst monk (see section 1.2) provides fragments of different sizes and chooses the beginning of a fragment randomly, so it is possible to have a read that is contained in another one. More formaly a read A is contained in another one B, if A and B share an overlap where A starts after the start of B, and A end before the end of B. All information (kmer or overlap with other reads) in contained reads was present in the container read for assembly task and so we can remove the contained read to save memory and time.

3.3.3 Bubble and tips

We have seen how OLC was built, but this graph can include some specific paterne, they can lead to misassemblies or fragmentation of assembly. A cleaning step was required.



(a) An example of tip in an assembly graph, the tips node is represented in red, the green, blue and black lines underline different possible assembly scenarios.

(b) An example of a bubble in an assembly graph, each path is in a different color. The length of each path can be different and there can be more than two paths in a bubble.

Figure 3.3

A tip in an assembly graph is a node with only one edge. A tip can be created by many things: trouble during DNA extraction, DNA duplication, an artifact created by the sequencer, a read with too many errors...

As we can see in Figure 3.3a a tip can create a branching node in the middle of a simple path. If we keep this tip, generally assembly creates two contigs, one before the tip and one after a two contig assembly scenarios (one for the green path another for the blue). If we remove this tip we can run the black scenario.

It is easy to detect and remove tips in a graph. In many assembly tools, tips are considered as errors and are removed.

We can define a bubble as a set of subpaths in a graph with the same parent and the same children. Figure 3.3b gives an example with two paths of equal length. The bubble can be created by repetition or heterozygosity, when one or more version of this sequence contains a substitution or a more complex mutation.

Larger bubbles can be harder to detect. With smaller bubble, only one version of the path is kept, the choice can be random or based on coverage or another other specific method.

Rugly we can say assembly tools use all simple path in OLC graph to generate a contigs.

3.4 The advantages of long reads

We said in Section 1.1, that the main properties of reads technology are length and error rate. The impact of error rate on read mapping and overlap search, was easy to understand. If reads contain a lot of errors, it is harder to find the right mapping position and overlap.

Reads length has a very important impact on assembly quality. Bresler et al. in [12] introduce the notion of genome assembly **feasibility**, whether it is possible to reconstruct the genome from a

Species	Short read N50 (bp)	Long read N50 (bp)	factor
Gorilla gorilla gorilla	913,458 [92]	23,141,960 [29]	25
Schistosoma japonicum	176,869 [1]	1,093,989 [60]	6
Escherichia coli strain CFT073	88,381 *	4,721,099 [61]	50
Ambystoma mexicanum	256 [37]	216,366 [99]	845

Table 3.1: contigs N50 (define in section 4.1) of some genome assembly with short and long reads. * GenBank Id 6313798

reads set with a given length and a given coverage. To summarize very roughly, to get a good assembly reads need to bridge the repetition, so reads must be larger than the largest repetition. The idea was extended to reads with errors in [96] and demonstrated that we need increased coverage when the error rate increases.

Before third generation sequencing, the maximum length of a read was less than 2 kb (for a Sanger read) but a majority of repetitions in the genome are longer than this. Koren and Phillippy in [43] indicate a theoretical length of read that is necessary to obtain a perfect genome assembly; for most bacteria, a read needs to be over 7 kb. But this limit does not work in all concrete situations. If reads start before a repetition cover all repetition and end after this repetition we can solve this repetition see Figure 3.4.

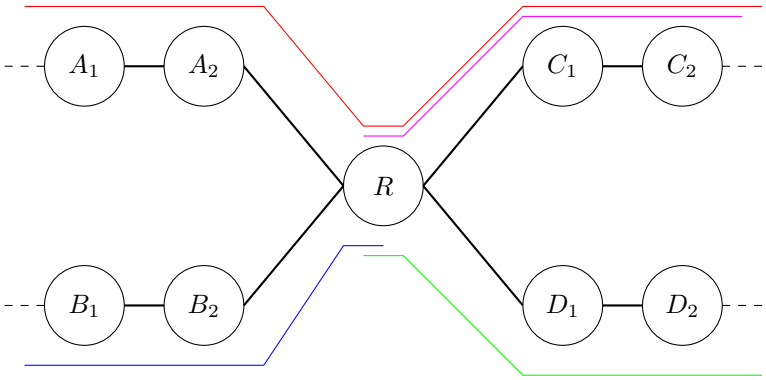


Figure 3.4: We have a part of assembly graph node `R` represent a repetition node `A`, `B`, `C`, `D` represent basic sequence. Red, purple, green and blue line represent reads. Red read was larger than repetition and span over it and indicate $A_1 \to A_2 \to R \to C_1 \to C_2$ was a good path, with out this read we can solve this repetition.

Third generation reads are not larger than all repetitions, but they are larger than many repetitions and help to produce better genome assembly. Table 3.1 shows the improvement in terms of N50 between short-read assembly and long-read assembly in a few instances.

Moreover, Yavas et al. in [113] perform an assessment of different versions of well known assemblies. Yavas et al. notice an important improvement in this assembly after the introduction of third generation reads and 10X data (for more information on 10X data read Section 5.3).

Recently a high quality human genome assembly (CHM13 cell line), telomere to telomere gapless assembly, was produced with a combination of Nanopore and Pacbio reads [65]. The authors of this paper focused their efforts on X chromosome, reconstructed a 2.8 megabase centromeric satellite DNA array and closed all 29 remaining gaps in the current X chromosome *H. sapiens* reference. Nanopore

data by analysis of raw signal provides an access to DNA methylation. This study confirm previous epigenomic results observed on the X chromosome.

Long read technology not only helps to improve genome assembly, it also has a significant impact on RNA study. Sequencing mRNA from beginning to end helps to detect new isoforms and splicing structures, by sequencing without PCR long reads help to remove bias in RNA quantification. But long read sequencing error rate and large input material requirements (compared with short-read RNA-seq) require new analysis methodology development [31].

After this overview of how `OLC` assembly tools work, let us look at the details of two long-read `OLC` assembly tools, `Canu` and `Miniasm`.

3.5 A Pipeline with correction `Canu`

Canu [44] was proposed in 2016, it is one of the first long reads assembly pipelines and it works with Pacbio and Nanopore reads after HGAP [19]

`Canu` is based on `Celera` [66, 69], we can split the `Canu` pipeline in three steps which will be described below: correction, trimming and assembly. Nevertheless, before each of these steps `Canu` searches overlaps between reads. We will thus start by explaining how overlaps are computed.

3.5.1 Overlapping

In `Canu` pipeline overlap is computed by `MHAP` (for MinHash Alignment Process). We have seen that overlap between all reads takes a lot of time and requires a lot of memory. To avoid all versus all alignment, `MHAP` tries to estimate which reads share a common part with another by estimating a Jaccard distance between the set of `k-mers` of two reads. A `k-mer` was a substring of sequence with a fixed size equal to k. The Jaccard distance, present in equation 3.1, evaluates the distance between two sets by dividing the intersection of the sets by the union of the sets.

$$J_\delta(A,B) = 1 - J(A,B) = 1 - J(A,B) = 1 - \frac{|A \cap B|}{|A \cup B|} \tag{3.1}$$

In this equation `A` and `B` represent the `k-mer` set of read `A` and read `B`, $J_\delta(A,B)$ represent the Jaccard distance and $J(A,B)$ represent the Jaccard index. If $J_\delta(A,B)$ is low, we can suppose read `A` and read `B` share a common part. Enumerating all the `k-mers` of each read and computing the intersection and union of each set takes a lot of time. `MHAP` selects a subset of `k-mers` to represent the read and computes a mash distance; [80] see equation 3.2

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} \approx \frac{|S(A \cup B) \cap S(A) \cap S(B)|}{|S(A \cup B)|} \tag{3.2}$$

$S(A)$ is a `k-mers` set composed by s a subset `k-mers` of set A. Ondov et al. evaluate the error between mash distance and Jaccard distance is in $\mathcal{O}(\frac{1}{\sqrt{s}})$, by default in `MHAP` $s = 512$ so the error is smaller than 0.05.

`MHAP` to choose which `k-mer` participate to the subset, assign to each `k-mer` a `tf-idf` score, see

equation 3.3. The tf-idf score comes from the field of text search. tf-idf evaluates if this term is specific to this document. tf for term frequency indicates if the term is present many times in the document, $n_{i,j}$ is how many time the term i is present in document and is j divided by the number of terms in document j. idf for inverse document frequency evaluates if the term is present in many documents or just a few, $|\mathcal{D}|$ is the number of documents in the dataset divided by $|\{d_j : t_i \in d_j\}|$, the number of documents where the term i is present.

$$\mathrm{tf} - \mathrm{idf}_{\mathrm{i,j}} = \mathrm{tf}_{\mathrm{i,j}} \cdot \mathrm{idf}_{\mathrm{i}} = \frac{n_{i,j}}{\sum_k n_{k,j}} \cdot \log \frac{|\mathcal{D}|}{|\{d_j : t_i \in d_j\}|} \tag{3.3}$$

In MHAP, terms are k-mer and documents are reads, this technique allows to reduce the number of k-mer in a set and keep k-mer specific to a read. If two reads share specific k-mer they probably share a common part.

If two reads have a small mash distance, MHAP compares the position of each k-mer in reads to determinate the overlap position.

The size of k-mer is very important as well. If k is too large, many k-mer contain errors, the size of intersection is reduced and MHAP can miss the overlap. Moreover, size of sketch has a huge impact. If it is too small, the read is sub-sample. If it is too large, compute mash distance takes more time, but with long-reads dataset the length of reads can be very different and choosing a good sketch size for this type of data is not easy. To find the optimal value for these two variable, the authors of MHAP perform many empirical tests.

3.5.2 Correction

In Canu correction was performed by a part of FALCON [20], falcon_sense. FALCON and Canu were developed simultaneously, we chose to describe Canu in detail instead of FALCON because we work mainly with Canu. In this section we did not cover the details of how falcon_sense work but only the main idea.

Some correction tools such as falcon_sense use a Partial Order Alignment (POA) (introduced in [50]) to perform long read correction. For each read R_1, we recruit all the reads with which it shares an overlap, and perform an pairwise alignment with it. This alignment was used to build a POA graph. In a POA graph each base was a node and a direct edge was created between two bases if the first base was before the second one in an alignment. If an edge was present in two alignments, its weight was incremented. After all the alignments had been added to the POA graph, we searched for weighed path in the graph, and followed them to reconstruct the corrected sequence. An example of POA graph construction is present in figure 3.5

3.5.3 Trimming

The trimming step will remove the parts of the reads that are not supported by the other reads, see Figure 3.6. For each read we will analyze its coverage curve and remove the parts of the read that are not sufficiently covered (by default this value is set to 1). For trimming, Canu uses a homemade tool.

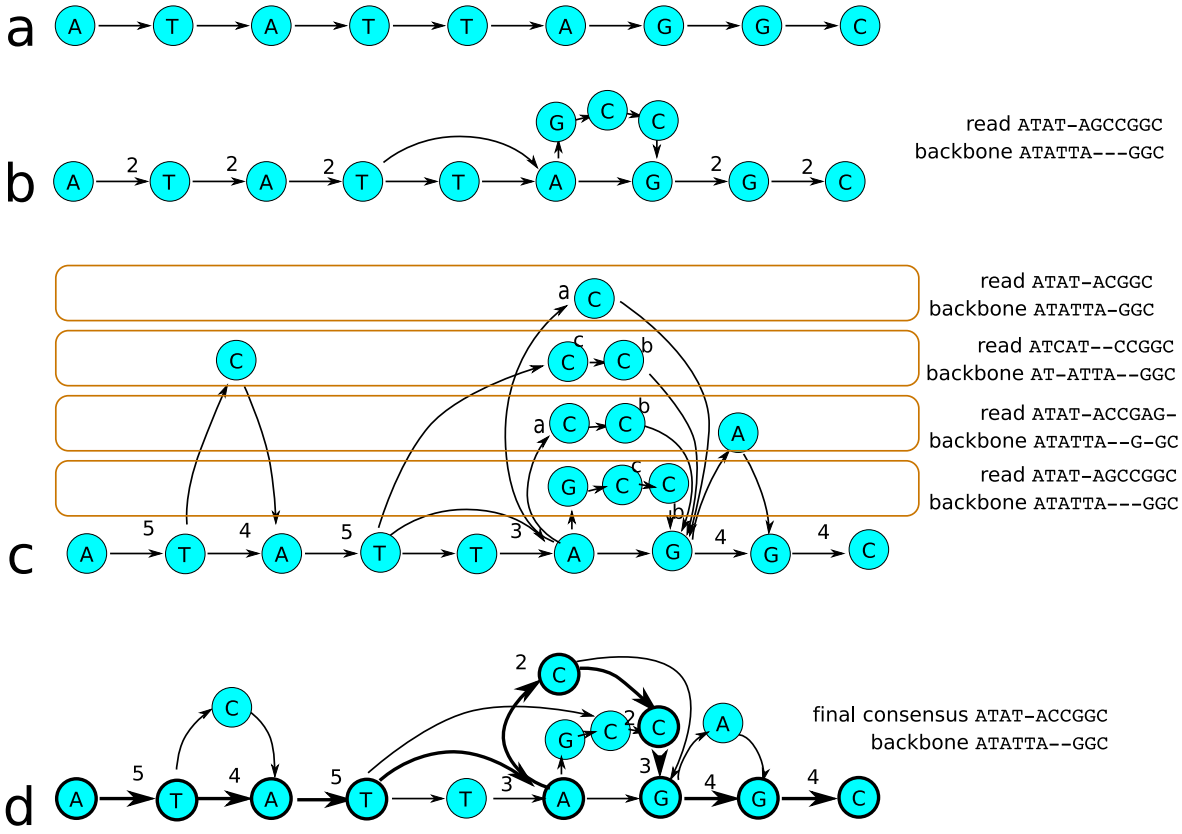


Figure 3.5: A sequence that needed to be corrected was represented by a graph, each base was a node and if a base was followed by another one a directed edge was built. (b) was the representation of sequence `ATATTAGGC` (called backbone in this figure), (b) We add the result of the alignment of one read in the graph. The number above the edge is its weight. If an edge exists in 3 alignments, its weight is equal to 3, (c) We add all other alignments in the graph, (d) the bold path was chosen as the correct path because it was supported by more alignments. This figure was originally present in Supplementary material of `HGAP` [19]

3.5.4 Assembly

The assembly step in `Canu` pipeline is based on the `OLC` paradigm (see Section 3.2 for a definition of `OLC`), with some specificities. `Canu` builds a Best Overlap Graph (BOG) for each non-contained read only two overlaps are kept in the graph, the best overlap for each read extremity, in `Canu` the best overlap was the longest overlap. Use of a BOG instead of a classic `OLC` graph is an aggressive strategy, in BOG we cannot observe a transitive edge and the number of edges is limited by the number of nodes. We avoid a cleaning step and reduce the memory impact of the graph. Once this graph construction step is performed, a clean step is run, removing tips and little bubbles (see Section 3.3.3).

This BOG was used as a scaffold to generate assembly. By remapping the reads against this scaffold, `Canu` tries to detect larger-than-read repetitions, which do not show as loops in BOG (see Figure 3.7). Afterwards, this mapping is used to build the consensus sequence of contigs. Each simple path in BOG was used to build a contig.

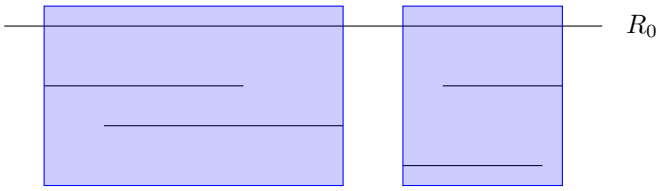


Figure 3.6: The black line is a read, the Canu trimming step keeps only the blue parts of read R_0, the parts that are covered by other reads.

By remapping reads on the BOG, Canu can build a consensus and detect repetitions not observed in the graph. BOG was an aggressive strategy to avoid transitive edge and reduce graph size, but it could hide an edge that would have indicated a repetition. This check was required too.

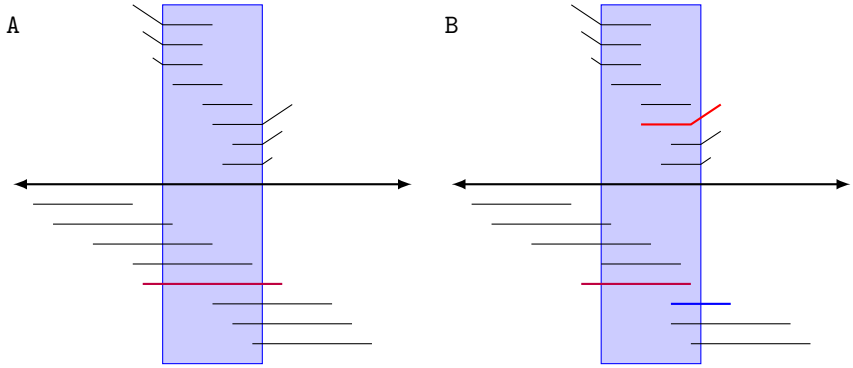


Figure 3.7: Black arrow line was the path chosen by Canu, the other line was a read mapped against this path, the blue box indicates a repeat region. In case A the purple read spans all the repetition and indicates that the path chosen by Canu was the good one. In case B no read spans the repetition and the purple read have non-congruent overlaps between the red and blue reads, so Canu needs to break the path in order not to create a misassembly

3.6 Pipeline without correction Miniasm

Minimap2 and Miniasm are an assembly pipeline proposed in [55] and [57], the main purpose of this pipeline is to demonstrate that we can perform a long read assembly without correcting the long reads before.

The miniasm pipeline is more simple than the canu pipeline because it does not incorporate correction and consensus building. It is made of steps:

- overlap search, performed by minimap
- trimming, by miniasm
- graph construction
- graph cleaning
- contig generation

3.6.1 Minimap2

The main idea with Minimap2 is that we can represent a read as a set of minimizer, and if two reads share the same succession of minimizer we can suppose these two reads share an overlap.

A minimizer is define (in Minimap2 publication) as the k-mer with the minimal hash value of a set of consecutive k-mer.

If we keep the same hash function, two set of k-mer from different reads but with same k-mer composition, have the same k-mer minimizer. Moreover, a k-mer can be the minimizer for several consecutive sets of k-mer if no k-mer with a lower hash value comes in the window.

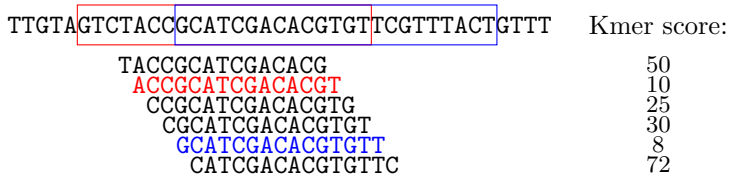


Figure 3.8: The red kmer has the lowest hash of the red window, so it is the minimizer of this window. But when the window slice arrives on the blue kmer, this one has a lower hash, the blue kmer become the minimizer of this window.

The minimal k-mer can represent many other k-mer, this technique can be compared to a lossy compression.

Minimap2 builds an index in which each minimizer is associated to the reads where a minimizer is present and the position of the minimizer in the reads.

With this index Minimap2 can collect the positions of similar minimizers between two reads. With this collection of positions Minimap2 looks for the largest co-linear match, a succession of similar minimizers in each read with coherent position, same order of minimizer and similar distance between each minimizer. Figure 3.9 shows an overview of an overlap of two reads in Minimap2.

Minimap2 reports overlap where the number of matches is sufficient (greater than a threshold, 3 by default) and total length of putative overlap is sufficient.

3.6.2 Miniasm

Miniasm did not perform correction but it did not take all the information from reads and overlaps either; a filtering operation was performed.

For each read Miniasm performs coverage analysis of reads based on mappings identified by Minimap2, by default only the longest part of reads with a coverage greater than three is kept. Minimap2 reports for each read, read length, position of first and last kmer, number of bases in kmer exact match, and a mapping quality and some option fields in SAM-like format can be present too.

Each overlap was classified in three categories, in order to keep only true end-to-end overlaps to build the OLC and filter out containment reads:

- internal match, this type of overlap probably corresponds to a repetition smaller than reads length

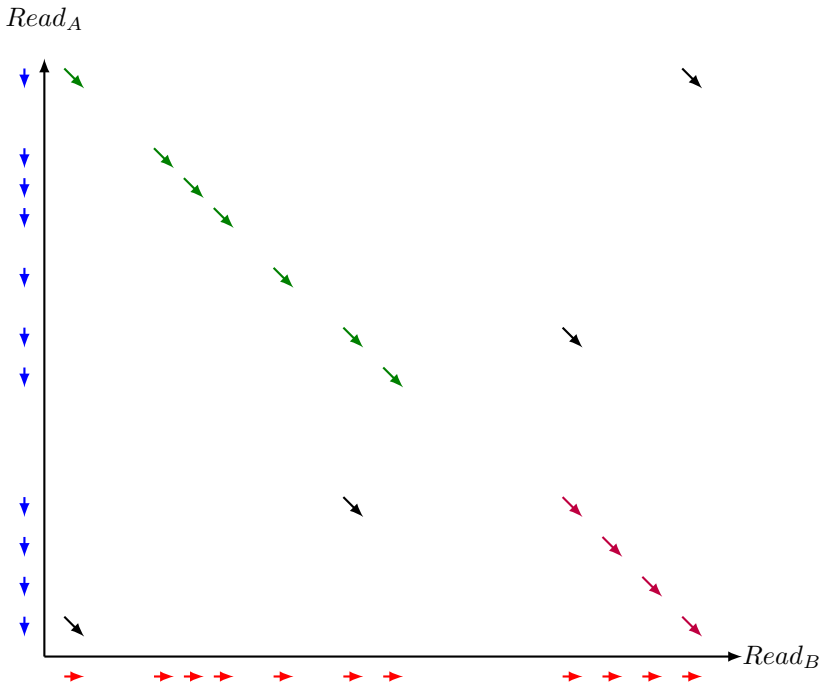


Figure 3.9: $Read_A$ and $Read_B$ are represented by black arrows. The common minimizers of $Read_A$ and $Read_B$ are represented by blue and red arrows respectively. The green arrows are a co-linear chain, the purple arrows another co-linear chain, the black arrows do not participate in a co-linear chain. The longest colinear chain is the green one. The end of $Read_A$ probably overlaps with the beginning of $Read_B$.

- containment, a read of this overlap is contained in the other read, it is the same sequence
- dovetail, it is an end-to-end overlap

Here we give intuitive definitions of these categories without being mathematically rigorous. One would argue that being rigorous here is not necessary, as these definitions turn out to depend on arbitrary criteria in practice (e.g. in `Miniasm`).

Figure 3.10 shows examples of these overlaps. Containment read was removed, only dovetail overlap was used to build the overlap graph. Tips, small bubbles and transitive edges were removed after this step. `Miniasm` takes each simple path and concatenates substring of read between the beginning and the first position of overlap.

`Miniasm` was design to work on uncorrected reads and did not perform a consensus step, so contigs generated by `Miniasm` contains many errors and cannot be used directly. We can run the `Minimap2 Miniasm` pipeline with corrected read and a polishing tool on contigs generated by `Miniasm`.

Very recently, another assembly tool `Ra` [103] was a created to replace `Miniasm` in `Minimap2 Miniasm` pipeline. `Ra` uses an analysis of coverage curve of each read to trim non-supported regions (like `Miniasm` does) it includes the detection of chimera and repeated regions. Overlaps on regions marked as repeated are marked in a string graph and not trusted. `Ra` performs a real consensus step and runs many polishing step with `Racon`. According to the authors and to another study[109], Ra performs good assembly on bacteria and plant genome, but the overlap step still could still be

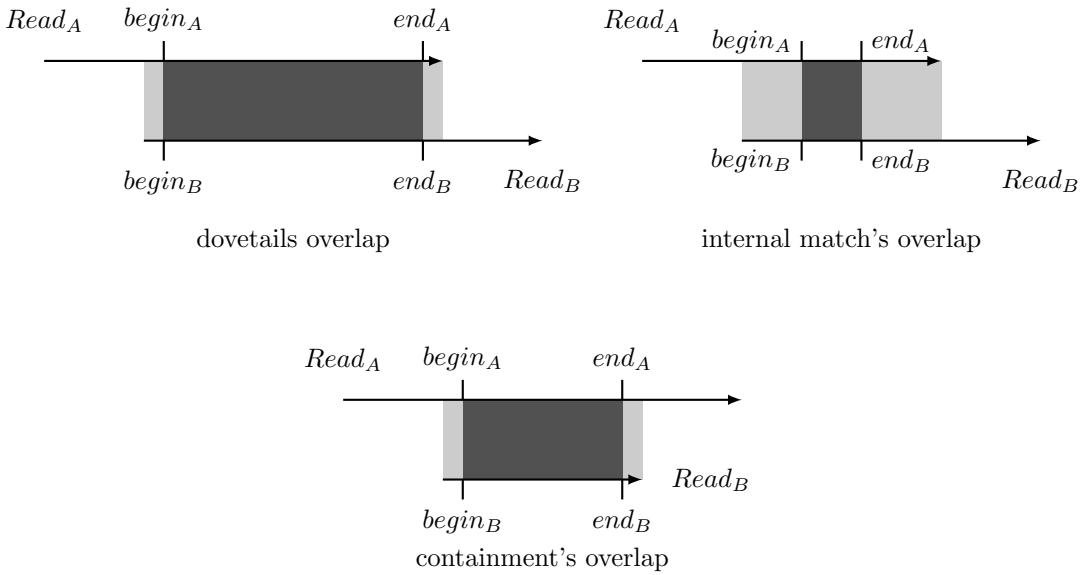


Figure 3.10: `Miniasm` classifies overlaps in three types of dovetail, internal match and containment overlap. The dark grey region corresponds to the part of the read between the first and last minimizer. The light grey region is called the overhang region, it is out of minimizer range. If overhang is large compared to the overlap region, we can suspect the overlap is not a true overlap.

optimized in terms of memory usage and computation time.

3.7 Long read assembly approaches using methods inspired by de Bruijn graphs

Another class of tools try to speed up tools try to speed up assembly by simplifying the overlap search step. This method was proposed in `EULER` [83].

This approach is based on a DeBruijn Graph (or `DBG`). For an alphabet with n symbols, a `DBG` represents each word of length k as a node and builds a directed edge if nodes share $k-1$ symbols at their extremities. For example, in Figure 3.11, node `ATCG` and `TCGG` share `TCG`. A word of length k is called a `k-mer`.

In assembly problems, $n = 4$ (A, C, T, G), and we can choose a value of k between 1 and the read length. In practice, the size is often smaller than the size of a read. The choice of the right values for k, depending on the use that we will have of the `DBG`, could be the subject for a whole thesis.

To build the `DBG` we chose a value for k and added all `k-mer` present in reads in the `DBG`. The `DBG` used in assembly contains only the `k-mer` present in the dataset, not all possible `k-mer`, and edges can be only edges that are present in the dataset or all possible edges.

Like `OLC` we can detect repetition by inspecting the number of successors of a node. Figure 3.11 presents a `DBG` with a repetition. After building the `DBG` we can follow the simple path to rebuild the original sequence.

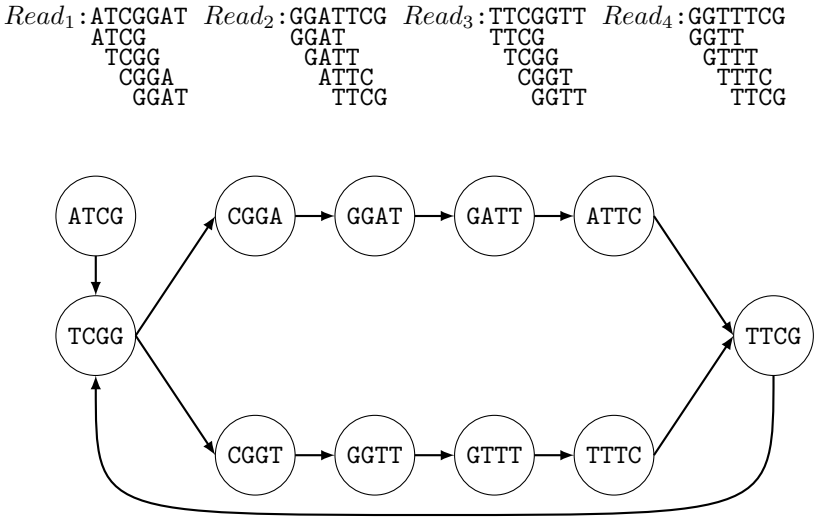


Figure 3.11: We have a dataset of 4 reads with length equal to 7, we choose a value of k equal to 4, kmer are present under each read. A DBG built from this kmer set is present under reads, each node is a kmer and if a word shares $k-1$ symbol at its end with $k-1$ symbols at the beginning of another node, we build a directed edge. This DBG contains a cycle. This cycle probably matches a repetition in the original sequence.

With the DBG strategy we did not compute overlaps between reads, but the word length in the graph was shorter. And all repetitions with a size greater than k create a cycle in the graph and fragment the assembly.

Moreover the overlap between words in DBG must be exact (there must be no error) and with a fixed length $(k-1)$. These two constraints are particularly problematic when the reads contain a lot of errors or when the coverage of the region is low.

The DBG approach was used successfully for short-read assembly. We can mention the tools Spades [7], Minia [16] and Megahit [51] but these methods are not very effective for long reads assembly:

- reads contain a high error rate and therefore finding error-free kmers was hard, these errors can lead to expand the size of the graph or misconnections between parts of graph.

- to use the size of the long reads, these tools would have to support values of k greater than, for example, 7000 bp (bacterial repetitions).

If we use DBG naively for long read assembly, we can miss the main advantage of long-reads: their length.

Flye and wtdbg2 use the DBG approach with some modifications to adapt the idea to long-read assembly. Flye creates a A-Bruijn Graph (ABG), in which an edge does not signal an overlap with a k-1 length between nodes but the overlap can be shorter. In the wtdbg2, method kmer are replaced with k-bin, where a bin is a substring (256 bases of length by default) of reads. An edge is created between k-bin if they are successive in a read.

3.7.1 Flye

Flye [41] was based on ABruijn [58] assembly tools. ABruijn did not use a DBG but a similar concept: a ABG. Instead of using a set of kmers as nodes, ABG uses a set of chosen kmers. Instead of building an edge between each kmer they share a k-1 overlap, they build an edge between succesive kmer in a read without creating a transitive edge. A weight was added to edge this weight correspond to the length of k minus the length of overlap between kmer.

To build the chosen k-mers set, ABruijn selects kmers present many times in the dataset. These kmers are called in many tools *solid* k-mers. The more present a k-mer is in the dataset, the more confident you can be that the k-mer does not contain a sequencing error. If the genome coverage is 40x we can hope to see a k-mer, not included in a repetition, roughly 40 times. Because when we sequence at 40x, we do not actually read each base 40 times; and when a sequence error appears, we lose an occurrence of the kmers where this base was present. Choosing the number of times a kmer has to be present in the dataset to be *solid* was a difficult task.

This modification helps to clean sequencing error, but reduces the set of kmer fragments the DBG graphs. This is why ABG creates edges not only when kmers share a k-1 overlap.

During the ABG construction, ABruijn stores which read generates which graph path. This structure was useful to find quick overlaps between reads. Reads participating in the same path of ABG probably have the same sequence, so they probably share an overlap. To build contigs, ABruijn choose a read, search all overlap with help of ABG. If this local overlap graph didn't denote a fork (we have one read without successor and all reads have path in the local overlap graph to this read), ABruijn extend the contig.

ABruijn can be roughly summed up as mix of all the assembly strategies, using DBG to find overlaps between reads, building contigs by extension as with *greedy* method but using OLC to make sure they do not integrate a repetition and a potential missassembly in contigs.

Flye was built on top of ABruijn. After the ABruijn assembly, Flye concatenate ABruijn contigs in pseudo-genome (contigs order is arbitrarily chosen). This pseudo-genome is alignment against it self. This self-alignment is analyse to detect and tag repetitions. Flye builds a repetition graph, with repetition extremities as nodes and an edge is built when two repetition extremities are linked in a contig. Flye uses coverage information to take a clue on contig succession over untangle repetition. By analysing the topology of repetition graphs, Flye can find a unique traversal path to explain all repetitions and find the genomic order of a contig.

3.7.2 wtdbg2

wtdbg2 [87] uses a DBG approach to solve long-read assembly. It is not really a DBG, but a "Fuzzy-Bruijn graph" (FDBG) and was defined for the first time in this article. To build this graph, wtdbg2 splits a read in a bin with a fixed size (256 base pairs) and stores the kmer present in each bin in a hash table. To find the overlap between reads, wtdg2 uses a hash table to compare the kmer compositions between each read and performs a pairwise alignment between each bin of reads.

After this alignment step, wtdbg2 only keeps in memory which bin is aligned to which bin, and it builds k-bins. Each k-bin is a sequence of k successive bin in a read. wtdbg2 can infer if two k-bins

overlap if one or more bin in this two k-bins shares an ovelap.

A group of k-bins are a node of FDBG. wtdbg2 builds an edge between two nodes if the k-bins from each node are successive in a read, after some cleaning step (pops bubbles, tips cleaning) wtdbg2 builds a consensus sequence with each simple path in FDBG.

3.8 New long read assembly method

Very recently, two assembly tools have been presented that focus on the ability to produce good long read assembly with a very low cost in computation time and memory usage.

3.8.1 Peregrine

Peregrine [18] uses the SHIMMER overlapper. SHIMMER extends idea of minimizer (introduced in Section 3.6.1), by creating a minimizer of minimizers. Given a set of minimizers, one of them can be chosen as a representative minimizer. These minimizers representing a set of kmer, we can have many layers of minimizers, each layer reducing the size of the minimizer set and the space of search to find similarities between reads.

The layer-0 of minimizers was a basic minmizer process, like Minimap2. After this step, SHIMMER selects the minimizers that will participate in the layer-1, it uses a reduction factor, for a reduction factor x, x minimizers are represented by the minimal minimizer of this set. This process can be repeated with many layers. When it chooses the minimizers of $layer_n$ among the minimizers of $layer_{n-1}$, SHIMMER checks the distance between each $layer_n$ minimizer to make sure they represent a distinct part of the read. SHIMMER by default uses three layers of minimizing this value, reduces the number of minimizers that have to be compared to find similarities between read, without increasing the number of missed overlaps (value found empirically).

After this indexing step, SHIMMER brings together reads that share many last layer minimizers and performs a classic alignment to confirm overlap between reads. After this step, Peregrine runs a classic OLC strategy to perform assembly.

SHIMMER overlapping tools can be used to perform a mapping of read against contig or genome. After this remapping a polishing step was performed, without taking into account heterozygosity.

Peregrine was actually tested only on Circular Consensus (CCS) Pacbio data. Reads were sequenced multiple times and a consensus was performed on all this sequencing. This technique reduces the read length but reduces the error level of sequencing too. Finding overlaps between reads with less error was easier and faster. Methods created by Peregrine tools by reducing the minimizer space of search speed up the search for overlaps, but they were tested on low error rate long reads. Even if the error rate of long reads decreases, will it decrease enough for this method to maintain a good sensitivity? Peregrine needs to be validated on other types of data before its method can be generalized.

3.8.2 Shasta

`Shasta` is a recently published assembly tools that was used to assemble Nanopore data of eleven human genomes [94].

`Shasta` uses run-length representation of reads. Run-length representation is a loss-less compression method for text that contains a large repetition of the same character. For example, the sequence `ACCTTTGAA`, was represented by two strings $Sb = \{$`A, C, T, G, A`$\}$ and $St = \{$`1, 2, 3, 1, 2`$\}$. To reconstruct the original sequence, we repeat St_i time the Sb_i letter.

This representation was interesting for long-reads data, because DNA contains sometimes the same character repetition (called an homopolymer) and long-reads often make errors in homopolymer. Run-length representation by squashing this region can avoid this type of error and facilitates the alignment of long-reads.

To perform read overlapping, `Shasta` did not use a minimizer approach but something very close, the Sb string of read was split in kmer and some kmer were selected randomly, and called markers. The set of markers was the same for all data set. Reads was now represented by a succession of markers: it is a lossy compression.

Before looking for a colinear match of marker, in order to select reads with a higher match probability, `Shasta` computes a modification of the MinHash Jaccard estimation (see Section 3.5.1) to avoid the bias created by the difference of length between reads.

To perform assembly `Shasta` creates a marker graph. It is something similar to `DBG`, in which a kmer is a marker and an edge is built between two markers if a read contains this succession of markers. Each edge is weighted by the number of reads that contains this succession. After a cleaning step of marker graph (removing transitive edges, tips and bubbles), a path in the marker graph is selected and the reads that have helped build the edges for this path are used to build a consensus sequence of contigs.

`Shasta` contains many interesting ideas and the authors plan improvements for heterozygosity detection, resolution and performance improvement.

3.9 Chapter Conclusion

Long read assembly is an active field of research, many tools are created each year and long read assembly tools are used to improve genome and build draft genome.

To perform overlap detection, a majority of tools use k-mer to find reads with a high similarity and avoid all-versus-all overlapping search. To further reduce the space search, some tools use filtering based on minimizing: we keep the kmer with the lowest score, this score can be based on information contained in the dataset or be determined by an arbitrary function. The choice of k-mer size, filtering method and minimizing function, can have a great impact on result of each tool.

The `OLC` approach has proven its effectiveness in assembling third generation reads. Several modifications have been made to support these new reads but efforts are mainly focused on reducing the computation time of memory usage. These modifications have led to the idea of a hybrid `OLC` algorithm with `DBG` and Greedy (cf `Flye` and `wtdbg2`). This hybridisation of method create tools where

interdependance between each step was more and more important. For exemple in `Flye`, `wtdbg2` and `Shasta` the search of overlap was linked to a assembly graph construction.

The only assembly that tried to take heterozygosity into account was `FALCON`, by making the difference between a sequencing error and a variant or heterozygosity. To get heterozygosity and variant phased or genome graph after assembly would be interesting. But the tools to extract all this information from a read do not exist yet.

Another step of assembly improvement would be to improve the contiguity, assembly tools by reducing the information to reduce computation time and memory usage can have an impact on assembly quality. Correction of long-read can by trimming insufficiently-covered data can increase the size of coverage gap. Coming back to all read information or using overlaps found by another tool, helps to solve assembly troubles created by heuristic in assembly and correction tools. The next chapter focuses on this point.

Chapter 4

Post Assembly

In the previous chapter we saw several third generation assembly tools, each one having its own specificity and method to produce a long read assembly. Each assembly tool produces different output files, but all of them produce a contigs file that store contigs sequences built during assembly. Other files generally contains information about contigs, coverage, if a contig is circular or not, which reads were used to build a given contig, ...

All this information is useful to assess the assembly quality, or to integrate other information to improve the assembly. In this chapter we will briefly review some methods for evaluating an assembly and will especially focus on the most commonly used method for evaluating a new assembly: the alignment of the contigs of the assembly against a known reference.We will see that this method requires some adjustment when evaluating an uncorrected assembly pipeline.

In a second part we observe how recent long-read assembly tools still fail to produce a good assembly on data although it should theoretically succeed. We thereafter present our solution `KNOT`. `KNOT` is a tool which by returning to the original information reads, tries to find information that could not be used by the assembly pipelines.

4.1 Assembly evaluation

Several metrics exist to compare and evaluate assembly. The most common metric used is the N50 that evaluates the contiguity of assembly. For example we take a genome with one chromosome and two assemblies. The first assembly contains one large contig (approximately the length of the chromosome) and many short ones. The second contains only contigs of average size one or two order of magnitude smaller than the chromosome. The first one has an higher contiguity. We have more information about the genome with the first assembly than the second one. We don't need perform an hard scaffolding step to have an idea of genome organisation.

To compute N50, we create a list of your contigs length and sort them. When the cumulative sum of contigs length (starting with the largest) is larger than the sum of all contigs, the length the

last added contig is the N50 value. For example, L is the sorted list of contigs length:

$$\begin{aligned} L &= \{20, 30, 40, 50, 70, 80\} \\ L_{sum} &= \sum_{i=0}^{|L|} L_i = 290 \\ \frac{290}{2} &> 20 + 30 + 40 + 50 \\ \frac{290}{2} &< 20 + 30 + 40 + 50 + 70 \\ N_{50}(L) &= 70 \end{aligned} \tag{4.1}$$

70 was the last length added in cumulative length before this cumulative length is larger than half of the total sum of assembly. N25, N75 or NX correspond to the same metrics as N50 for 25%, 75%, or X% of total length of contigs. L50 is the rank of the N50 contig in the sorted contigs list, L50 of our example is 5.

NG50 is the same thing as N50, but the total sum of contigs length is replaced by the genome length (estimated or get from a previous assembly). NGA50 is the same as NG50, but the contigs length is replaced by the length of contig that map against the reference genome. We can cite U50 as another metric similar to N50 where overlapping region between contigs was ignored [14].

N50 family metrics are not perfect, but they help to represent the contigs length distribution, and to compare the results of different assembly tools on the same dataset. N50 is useful to analyze assembly quality without any external information.

By adding other information, we can evaluate assembly not only on size of contigs. `BUSCO` [97] evaluates the assembly completeness with the presence or the absence of core genes. By mapping contigs against reference genome or close reference genome, `Quast` [30] computes many metrics like the number of misassemblies, NGA50, the identity level of contigs, ...

Some other tools and techniques exist and are useful. Some of them are presented in more details in [76]

4.2 Misassemblies in noisy assemblies

Originaly publish in: https://blog.pierre.marijon.fr/misassemblies-in-noisy-assemblies/

Author: Pierre Marijon

4.2.1 Introduction

I think that all the people who have ever done a genome assembly one day say: "Ok my assembly is cool, but now how I can be sure that it's the best and it doesn't contain a lot of errors ?"

We have many technics to evaluate the quality of assemblies (it isn't a complete review, sorry):

- with only assembly information:

  - with N50 family metrics
  - by analyzing reads remapping against assembly AMOSValidate, REAPR, FRCbam, Pilon, VALET
  - by computing the probability of the reads given the assembly (ALE, CGAL, LAP)

- by using external information:
  - count the number of core genes present in an assembly, BUSCO
  - transcriptome information, for example, *Bos taurus* genome validation
  - synteny information Lui et al
  - map assembly against a near reference genome, quast or dnAQET

Note that for the last bullet point, if you are using quast with a reference genome you already have, by definition, a reference genome. So why perform an assembly?

The main reason to perform reference-assisted evaluation is when testing different assembly pipelines on the same read data set. To evaluate a new assembly pipeline, one also has to test different sets of parameters, and evaluate the impact of adding or changing the tools that are part of the pipeline.

Quast is a very useful tool and now it integrates many other assembly evaluating tools (BUSCO, GeneMark, GlimmerHMM, barnap)

Recently, with Rayan Chikhi and Jean-Stéphane Varré, we published a preprint about yacrd and fpa, two new standalone tools. These tools can be included in assembly pipelines to remove very bad reads regions, and filter out low-quality overlaps. We evaluated the effect of these tools on some pipelines (miniasm and redbean). Using quast, we compared the results with the assembly quality of different pipelines.

We sent this paper to a journal, and one of the reviewers said something along the lines of: "quast isn't a good tool to evaluate high-consensus-error assemblies, the number of misassemblies was probably over evaluated."
And it's probably true.

Miniasm and redbean perform assemblies without read correction steps (and without consensus step for miniasm). The low quality of a contig sequence is a real problem: quast could confuse a misaligned low-quality region with a misassembly.

In this blog post, I want to answer the following questions:

1. how to run quast on long-read uncorrected misassemblies
2. is the quast misassemblies count a good proxy to evaluate / compare assemblies?
3. can we find better metrics than just the number of misassemblies?

If you have no time to read all these long and technical details you can go directly to the TL;DR.

In this post I will talk about quast and not dnAQET, which has just been released, but dnAQET uses the same method (mapping the assembly against the reference) and the same misassembly definition as quast. It seems to me that what I am going to say about quast also applies to dnAQET. But go read the dnAQET publication, there are lots of super interesting ideas in it.

4.2.2 Datasets, assembly pipelines, analysis pipelines; versions and parameters

For our tests we are going to use two Nanopore datasets and one Pacbio dataset.

- Reads:
  - Oxford nanopore D melanogaster 63x coverage
  - Oxford nanopore H sapiens chr1 29x
  - Pacbio RS P6-C4 C elegans 80x
- References:
  - D. melanogaster 143.7 Mb
  - C. elegans 100.2 Mb
  - H. sapiens chr1 248.9 Mb

To perform assembly we use minimap2 (version 2.16-r922) and miniasm (version 0.3-r179) with recommended preset for each sequencing technology (`ava-ont` and `ava-pb`).

We use racon (v1.4.3) for polishing. For mapping reads against assembly we use minimap2, with recommended preset for each sequencing technology.

We use quast version v5.0.2.

All dotplots were produced by D-Genies.

4.2.3 Quast misassemblies definition

What are quast misassemblies? Do we have different misassembly types? How are they defined?

Quast defines three types of misassemblies: **relocation**, **translocation** and **inversion**.

4.2.3.1 Relocation

A relocation can occur based on signal from two mappings of the same contig against the same chromosome (cf Figure 4.1). We have two cases:

- either the two mappings are separated by an unmapped region (case **A**)
- or they map on the same chromosome with a shared mapping area (case **B**)

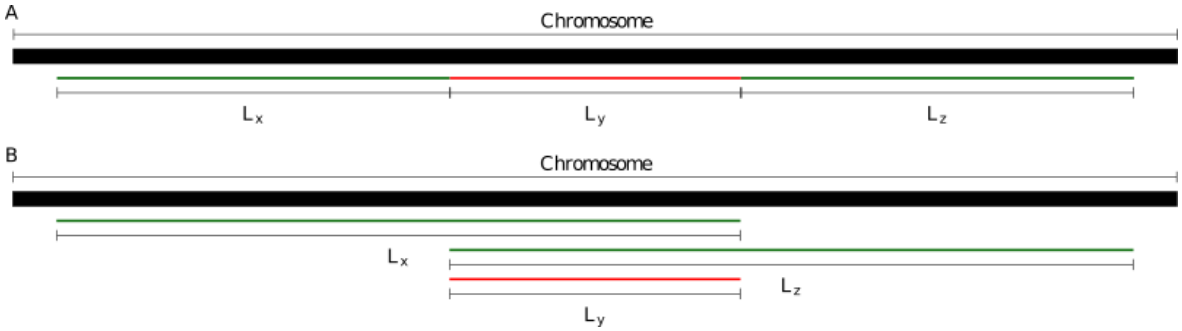


Figure 4.1: A schematic representation of a relocation

A misassembly is said to occur when L_x and $L_z > 1$kbp (this value can't be changed, it seems) and when $L_y >$ `extensive-mis-size` (1kbp by default).

Let's call L_y the length of the relocation.

- The relocation length is positive when the assembly missed a part of the reference (case **A**)
- Negative when the assembly includes a duplicated region (case **B**).

In both cases, this is an assembly error.

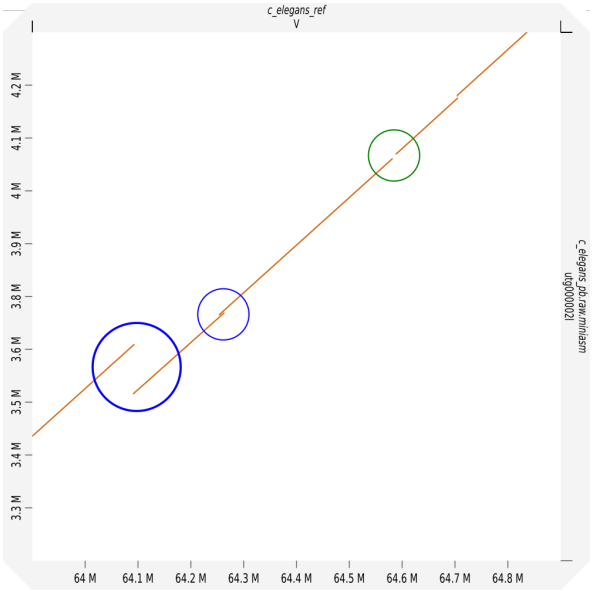


Figure 4.2: Thrid relocation observe in dotplot a long reads assembly against reference of *C. elegans*

In dotplot present in Figure 4.2 of contigs ctg000002L for our *C. elegans* miniasm assembly against the chromosome V of the reference. We can see two relocation events of type **B** circled in blue and one relocation event of type **A** (green). I have no idea on how to explain the other problem on the top right.

4.2.3.2 Translocations

A translocation occurs when a contig has mapped on more than one reference chromosome (cf Figure 4.3).

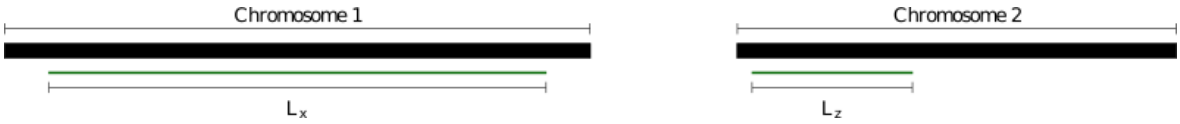


Figure 4.3: A schematic representation of a translocation

It's easy to spot this kind of misassemblies on a dotplot because of the multi-chromosome match.

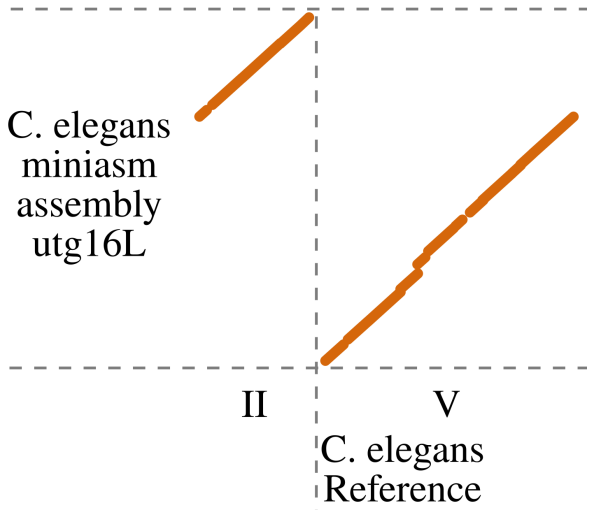


Figure 4.4: A translocation in a dotplot.

In Figure 4.4, two parts of contig 'utg16L' from our *C. elegans* miniasm assembly, map respectively on chromosomes II and V of the reference. This contig contains a translocation without any doubt.

4.2.3.3 Inversions

An inversion occurs when a contig has two consecutive mappings on the same chromosome but in different strands (cf Figure 4.5).

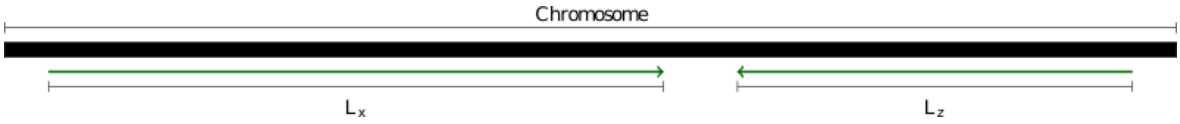


Figure 4.5: A schematic representation of a inversion

The dotplot present in Figure 4.6 shows an inversion between a reference genome and a miniasm assembly of *C. elegans*.

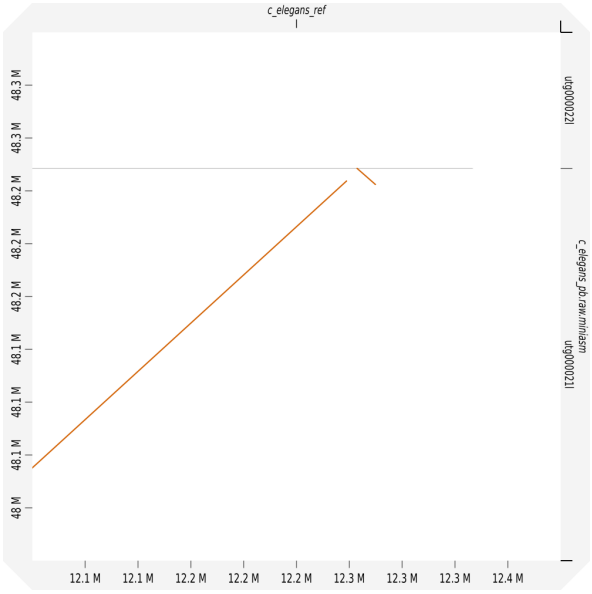


Figure 4.6: The contig utg0000021L maps on chromosome I, but it contains a small inversion at its end.

4.2.3.4 Important point

For more details on quast misassembly definitions, you can read this section 3.1.1 and section 3.1.2 of the quast manual.

Quast bases its misassemblies analysis on the alignmnt of contigs against a reference. To perform alignment, recent versions of quast use minimap2, with preset `-x asm5` by default, or `-x asm20` when min-identity is lower than 90%. After that, alignments with identity lower than `min-identity` are filtered out by quast (95% identity by default, but can be set to as low as 80%).

`min-identity` is a very important parameter. To consider a contig as misassembled, quast must have a minimum of two mappings for this contig. If the second mapping has an identity under the `min-identity` threshold, quast can't observe the misassembly. But even more, if a contig has three successive mappings, and assume also that the mapping in the middle has lower identity than the `min-identity` threshold, and the remaining gap between the two other mappings is larger than `extensize-mis-size`, then quast sees this as a misassembly, where in fact it isn't.

Parameters `min-identity` and `extensize-mis-size` have an important impact on misassemblies detection. So, what is the effect of changes in of these two parameters on the number of misassemblies found by quast?

4.2.4 Effect of min-identity

4.2.4.1 Low min-identity is required for uncorrected assembly

Quast only uses mappings with alignment identity higher than `min-identity`. So, what could be a good value for this parameter for long-read uncorrected assembly?

The file `contigs_reports/minimap_output/{output-name}.coords`, generated by quast, in the fourth column contains the alignment identity %. For each dataset, we extracted this value and plot it in an histogram (cf 4.7).

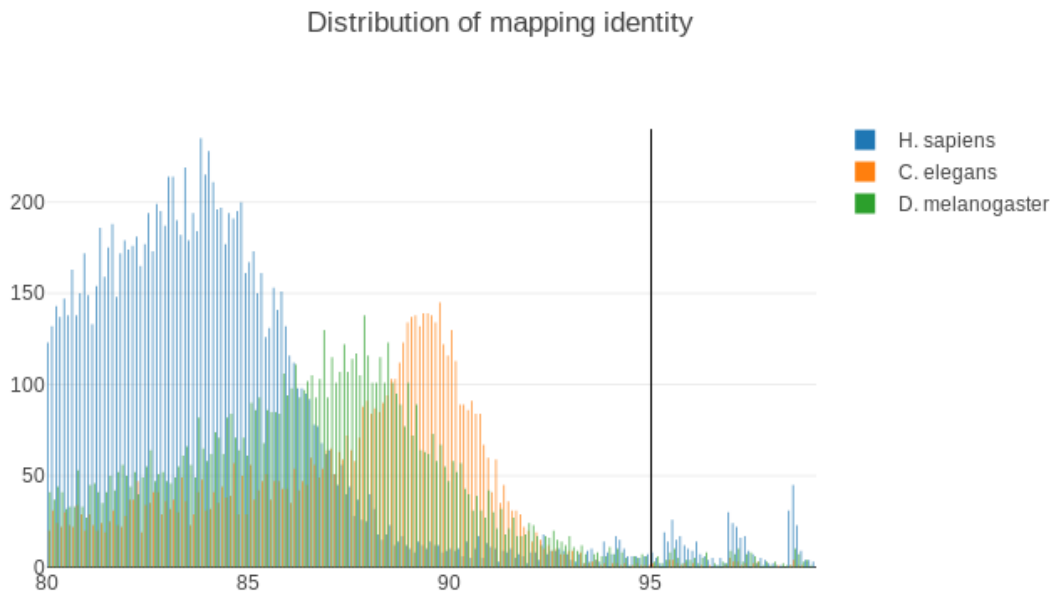


Figure 4.7: Horizontal axis: identity percentage bins, vertical axis: number of mappings in each bin.

The black line marks quast default identity value threshold, we can see a majority of alignments are under this threshold for an uncorrected dataset. So, setting parameter `min-identity 80` seems necessary.

4.2.4.2 Effect on a polished assembly

To test the effect of correction on misassemblies count, we ran racon 3 times on *C. elegans* (the one with the best reference) dataset.

On the non-corrected assembly, quast makes use of 7049 mappings; for the corrected assembly, 30931 mappings (increasing ratio 4.38).

We can observe in Figure 4.8 an increase in alignment identity due to racon (unsurprisingly). Contrary to the uncorrected assembly, a majority of the mappings now have 95% or more identity.

To have an insight on the effect of `min-identity` on unpolished/polished assemblies, we run quast with default parameters and changing only `min-identity` (still the *C. elegans* dataset).

With `min-identity 80` the number of relocations and translocations is increased compared to the default value of `min-identity`. If quast has only one alignment of a contig, it cannot find misassemblies. By reducing the `min-identity` we increased the number of alignments and mechanically increased the number of detected misassemblies.

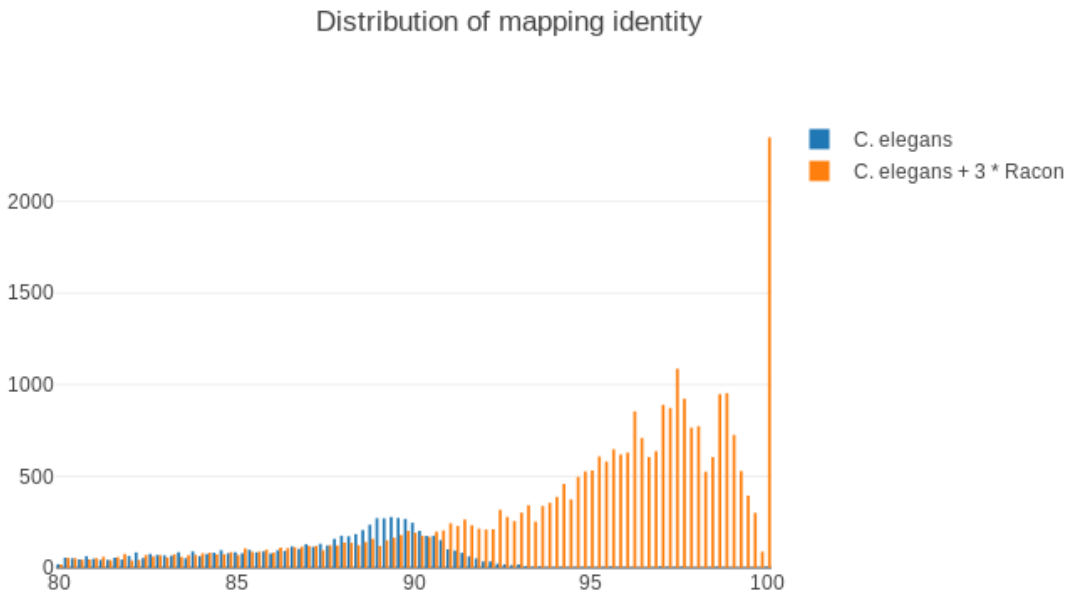


Figure 4.8: Horizontal axis: identity percentage bins, vertical axis: number of mappings in each bin.

racon	no	yes	yes
min-identity	80	80	95
relocation	1131	886	635
translocation	200	259	170
inversion	65	68	75
total	1396	1213	880

Table 4.1: This table shows the number of different types of misassemblies, whether we run racon on the assembly or not and according to the value of the threshold `min-identity`

We think that some of these misassemblies aren't real misassemblies. But if we use the same `min-identity` value for all assemblies that we want to compare, we can hope that the number of 'false' misassemblies will be similar.

For uncorrected long-read assemblies, we recommend to use a lower-than-default QUAST identity threshold parameter (80 %)

4.2.5 Effect of extensive-mis-size on misassemblies count

We observed that the `min-identity` parameter has a very important impact on the number of misassemblies for uncorrected long-read assemblies (-> need to set it to 80 %.) Now we want to observe what is the impact of another parameter: `extensive-mis-size`, which is a length threshold for the detection of relocation-type misassemblies.

We launch quast with different value for parameter `extensive-mis-size`: 1.000, 2.000, 3.000,

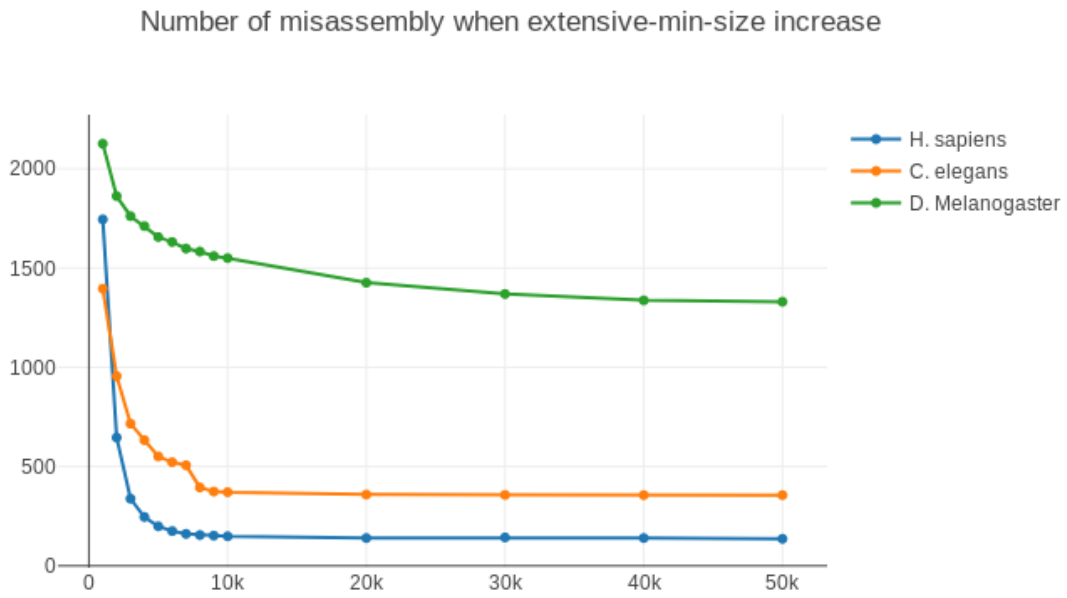


Figure 4.9: In the horizontal axis, we have the `extensive-mis-size` value. In the vertical axis we have the number of misassemblies.

4.000, 5.000, 6.000, 7.000, 8.000, 9.000, 10.000, 20.000, 30.000, 40.000, 50.000 (in base pairs). The parameter `min-identity` was set to 80 %.

The Figure 4.9 shows the evolution of the number of misassemblies in function of the `extensive-mis-size` value. After 10.000 base pairs, the number of misassemblies becomes quite stable.

This graph shows two regimes: with `extensive-mis-size` lower than 10.000 bp, it detects quite a lot of misassemblies. With `extensive-mis-size` higher than 10.000 bp, it detects less of them. **Yet we know that quast detects three type of misassemblies (relocations, translocations, inversions). Only relocation should be affected by `extensive-mis-size` parameter, but let's verify this assumption.**

4.2.5.1 Effect of parameter extensive-mis-size on the detection of each misassembly type

Quast defines three types of misassemblies **relocation**, **translocation** and **inversion**. Previously we observed the total number of misassemblies. Now we break down by group of misassemblies (cf Figure 4.10).

The *H. sapiens* dataset doesn't have any translocation because the reference is composed of only one chromosome. The majority of misassemblies are relocations, but when we increase the parameter `extensive-mis-size` the number of inversions also increases.

D. melanogaster reference contains many small contigs. This can explain the high number of translocations. Relocations and translocations drop at the same time.

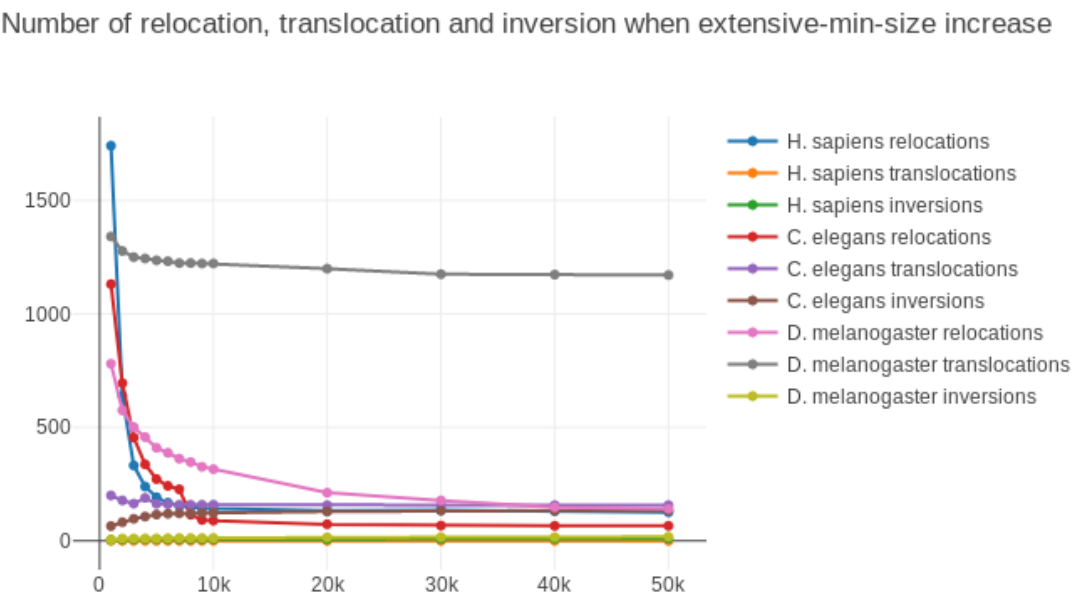


Figure 4.10: In the horizontal axis, we have the `extensive-mis-size` value. In the vertical axis, we have the number of misassemblies.

For *C. elegans* the number of translocations was quite stable, the number of relocations drops down rapidly and the inversions has only a little increase.

I can't explain why translocations and inversions numbers change with a different value of `extensive-mis-size`. By reading quast documentation and code I didn't understand the influence of this parameter on this group of misassemblies.

Relocation misassemblies are the most common type of misassemblies. We can impute the reduction of misassemblies, when `extensive-mis-size` grows, to a reduction of relocations.

4.2.5.2 Relocations lengths distribution

We see previously for our assemblies that a majority of misassemblies were relocations. We are now focused on this type of misassemblies. For each relocation we can attach a length, this length is the length of incongruence between assembly and reference genome. It's equal to L_y.

The file {`quast_output`}`/contigs_reports/all_alignements_`{`assembly_file_name`}`.tsv` contains information about mapping and misassemblies. For other information on how quast stores mapping and misassemblies information, read quast faq.

The Figure 4.11 shows a swarm plot of log of length associated to recombination. It's the size of the gap between mappings flankings a misassembly. If the length is positive, the assembly misses part of the reference (green point). If the length is negative, the assembly duplicates a part of the

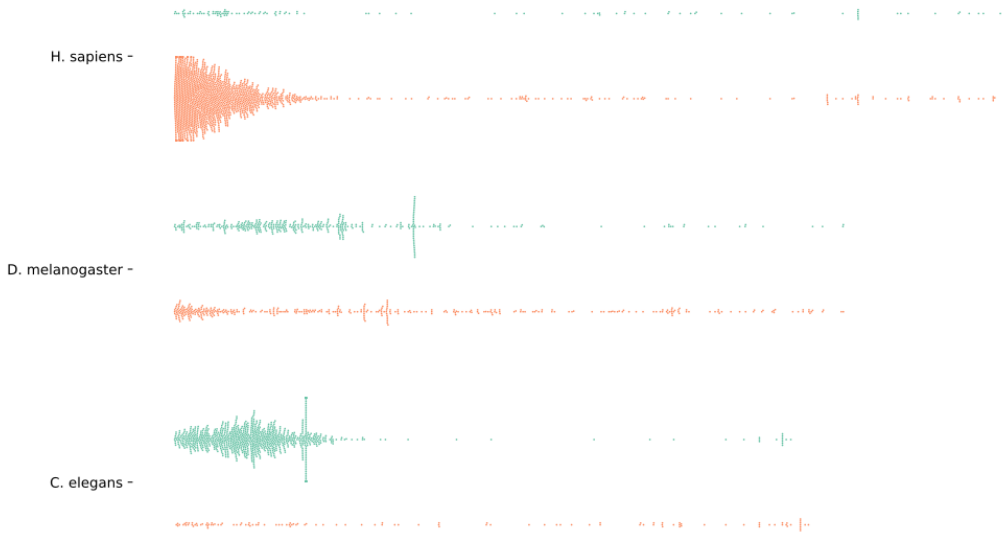


Figure 4.11: In the vertial axis, we have the log length of each relocation. Each raw is a species. Green points are for negative (<0 bp) relocations, orange points for positive relocations.

reference (orange point). Source code, data is available.

For *H. sapiens* a majority of relocations were positive and short (between 1000 and 5000 bases), with some very large relocations. For *C. elegans* it's different, the majority of relocations are negative and the largest relocation was shorter than in *H. sapiens*. For *D. melanogaster* the size of relocations was more spread out; the majority of relocations aren't short. This is confirmed by the look of the curve seen in the previous part, when `extensize-mis-size` is increased, the number of relocations decreases less quickly than for the other datasets.

With this representation, we can analyze the differences in relocations between assemblies, in terms of their numbers and more importantly the distributions of their lengths.

4.2.6 Conclusion

If you work with quast to evaluate an assembly made with miniasm, you need to set `min-identity` parameter to 80 %. It would be nice to have a lower minimum value, maybe 70%, but the quast code would have to be modified. And such a low identity is required only for a miniasm assemblies; for tools with a better consensus step (redbean for exemple), 80 % seems sufficient.

Translocations and inversions constitute a minority within misassemblies, yet when they are detected it's clear that they are 'true' misassemblies. I would be very surprised to see a translocation or inversion created by a mapping error, itself generated by error(s) in an uncorrected long-reads assembly. We can thus trust the count of translocations and inversions.

For relocations, the situation is different. They constitute the majority of misassemblies in our cases, and some of them are *true* some of them are *false*. Checking all misassemblies manualy is impossible, and finding a good `extensive-mis-size` value seems very hard for me. The easiest thing we can do is compare the series of lengths associated to relocations, as shown in this blogpost I used a swarmplot; I think statisticians could find better tools.

4.2.7 Take home message

You can use quast to compare uncorrected long-reads assemblies but:

- run quast with `--min-identity 80`
- rely on translocations and inversions counts
- for relocations, compare distributions of lengths associated to each assembly

4.2.8 Acknowledgements

For the creation of this very effective and useful tool all Quast contributors.

For their help in writing this blogpost:

- Rayan Chikhi

- Jean-Stéphane Varré
- Yoann Dufresne
- Antoine Limasset
- Matthieu Falce
- Kevin Gueuti

4.3 Trouble with heuristic algorithm

Assembly tools need to rely on heuristics. Due to theoretical limit: how many bases need to be share between two read to create an overlap, how many errors can we accept in this overlap. Due to technical limit: memory constraint, computation time limit. You can't search and store all overlap. Most of the time chosen heuristics perform very well, but in some cases a more complex analysis is needed.

Wick and Holt in [109] perform a comparison of five assembly tools on real data and simulated data bacterial data set. Some difficulties are injected in the input long-reads to stress assembly tools:

- Adaptor length. Sequencing techniques require the introduction of short sequence before reads. Because of their high error rates to detect and remove those adaptor from long-read sequences is not trivial. Those adaptors can generate assembly errors.
- Chimeric read. During DNA extraction and fragmentation, two fragments coming from different regions can be sequenced as a single read. This can lead to assembly fragmentation.
- Glitch level. Long-read error aren't uniformly distributed along the reads and sometimes sequencer create a region with only random sequence. A higher the glitch level indicates a larger region and a higher frequency.
- Random junk reads. Some reads are just a string of random character.
- Read depth, corresponds to genome coverage.
- Read identity, percent of error insertion, deletion, substitution.
- Read length.

This study focuses on assembly contiguity, the number of contigs vs. the expected number of contigs, and the number of contigs that can be mapped against the reference. According to this benchmark we observe that:

- reads length are upper than 10k and lower than 20k, this length can be reached by long-read sequencing technology but requests a particular attention be focused on the risk of DNA fragmentation
- read identity need to be upper than 85%

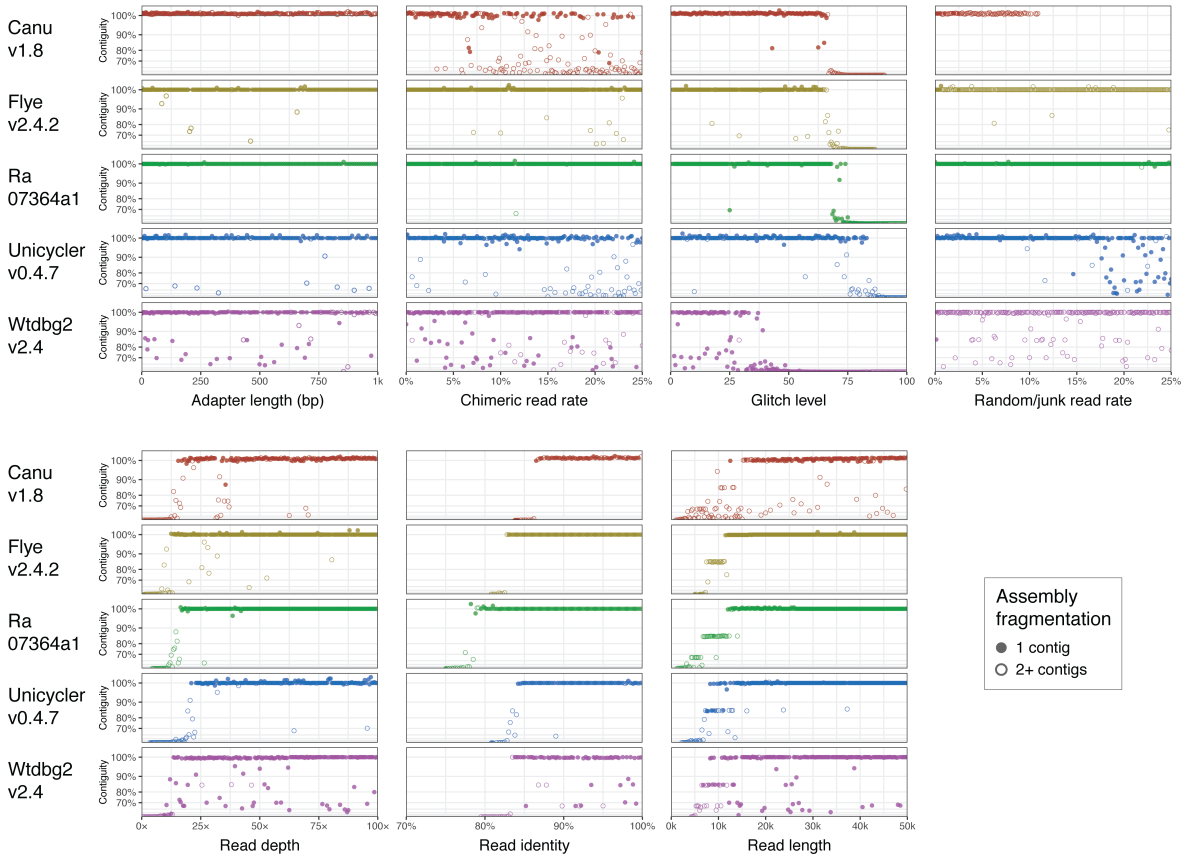


Figure 4.12: Effect of different reads property on assembly contiguity (number of contigs expect and map correctly on reference genome), of five assembly tools. `Unicycler` is an hybrid assembler (use second and thrid generation read). `Canu` is a long-read assembly pipeline they perform a self correction before construct assembly with a special `OLC` graph (more detail in Section 3.5). `Ra` perform a basic string graph assembly on raw reads with a correction of contigs after assembly (more detail in Section 3.6). `wtdbg2` and `Flye` use a `DBG` like approach to perform assembly on raw reads (more detail in Section 3.7). This figure is a reproduction of figure from [109].

- the minimal coverage is around 20x, but this study didn't analyze the error rate of assembly, we can suspect a high error rate in assembled contigs
- chimeric reads have an important impact on assembly contiguity but at level generally not observed in real data

We can observe an important variability of result (in `Canu`, `wtdbg2` and `Unicycler`). An assembly can fail for many reasons: a chimeric read in a repetition, a drop of coverage, a missing overlap, or a inappropriate set of parameters.

Analysis and understanding of the data produced by assembly tools help to check if assembly result didn't produce false result or to understand, and sometimes solve, assembly trouble. Some tools use remapping of reads against assembled contigs to found misassembly by detecting incongruity's in read coverage, mate pairs mapping, read mapping clipping. Some tools or assembly tools were developed in order to analyse assembly graphs to understand what is happening during assembly like `Bandage` [110], a tool to visualize assembly graph.

We developed `KNOT` a tool to simplify analysis of assembly tools results and help users to make choices improving assembly quality. This tool is based on the observation that the graph of raw reads is generally connected (we can reach any node from any node), while the graph of contigs does not. Therefore the idea of `KNOT` is to use the graph of raw reads to find the (potentially missed) links between contigs.

The Figure 4.13 present the main idea of `KNOT`, to combine information of assembly (the read coloration) with pieces of information that can be extracted from reads (the `OLC` graph build from `Minimap2` but another overlapping tools can be used). The contigs information helps us to ignore some already solved problem (red circle), unsolvable trouble (greed circle) and to focus on strange situations (blue circle). Figure 4.13 show a very simple example on a real case. The `OLC` graph can be very hard to read and understand for a human, analysing an `OLC` graph by hand is almost impossible. For these reasons and to run analysis without an human intervention we also automatised the idea of `KNOT`.

The paper was publish originally publish in Bioinformatics (https://doi.org/10.1093/bioinformatics/btz219), we reformat the paper in the style of this current document for reasons of readability.

4.4 Graph analysis of fragmented long-read bacterial genome assemblies

Originaly publish in Oxford Bioinformatics : https://doi.org/10.1093/bioinformatics/btz219

Author: Pierre Marijon, Rayan Chikhi, and Jean-Stéphane Varré

4.4.1 Abstract

Motivation: Long-read genome assembly tools are expected to reconstruct bacterial genomes nearly perfectly, however they still produce fragmented assemblies in some cases. It would be beneficial to understand whether these cases are intrinsically impossible to resolve, or if assemblers are at fault,

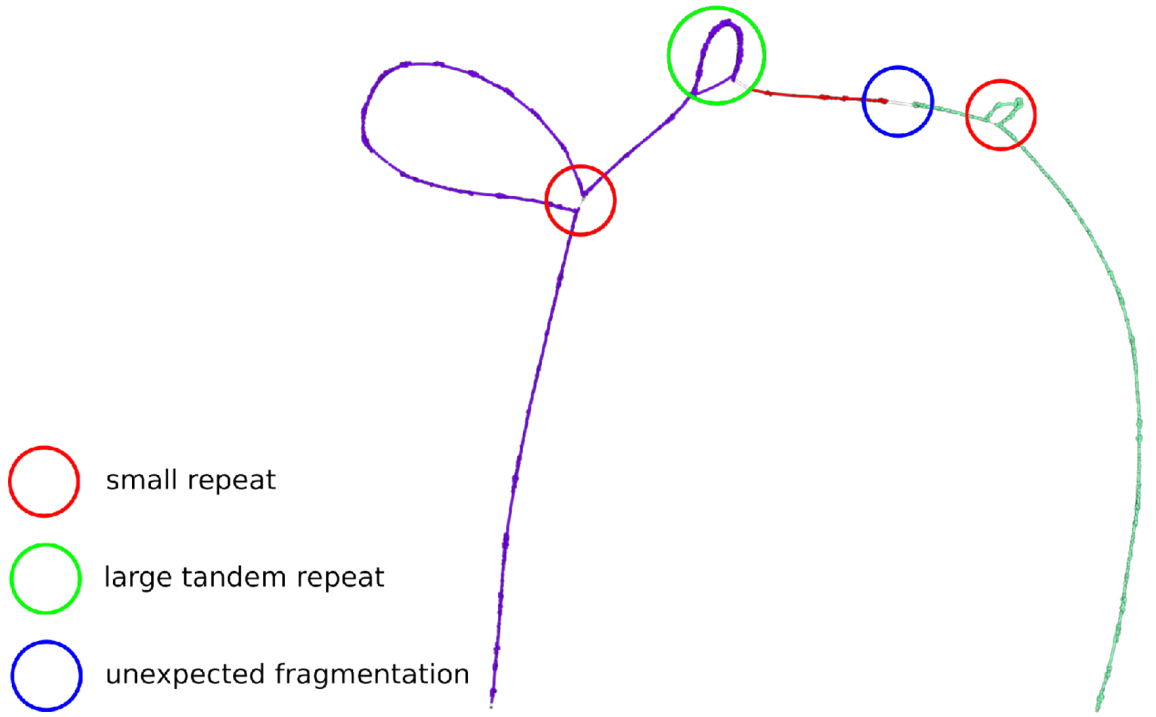


Figure 4.13: This graph is the overlap graph (computed by `Minimap2`), reads used by `Canu` to build its contigs are colored with same color. We can thus distinguish the three contigs computed by `Canu`. We can observe two fragmentation points, one can be explained by a repetition (green circle). We can observe that some repetitions are solved by `Canu`. But the fragmentation between green and red contigs (blue circle) can't be explained by a repetition.

implying that genomes could be refined or even finished with little to no additional experimental cost. **Results:** We propose a set of computational techniques to assist inspection of fragmented bacterial genome assemblies, through careful analysis of assembly graphs. By finding paths of overlapping raw reads between pairs of contigs, we recover potential short-range connections between contigs that were lost during the assembly process. We show that our procedure recovers 45% of missing contig adjacencies in fragmented `Canu` assemblies, on samples from the NCTC bacterial sequencing project. We also observe that a simple procedure based on enumerating weighted Hamiltonian cycles can suggest likely contig orderings. In our tests, the correct contig order is ranked first in half of the cases and within the top-3 predictions in nearly all evaluated cases, providing a direction for finishing fragmented long-read assemblies.

Availability: https://gitlab.inria.fr/pmarijon/knot

4.4.2 Introduction

Third-generation DNA sequencing using PacBio and Oxford Nanopore instruments is increasingly becoming a go-to technology for constructing reference genomes of non-model prokaryotes and eukaryotes. Longer sequencing reads allow in principle to overcome the reconstruction problems posed by genomic repetitions [13]. Direct assembly of second-generation (Illumina) sequencing data typically also results in high consensus accuracy yet generally more fragmented bacterial assemblies [7]. The large-scale ongoing NCTC project aims to assemble and make publicly available 3,000 bacterial strains sequenced using PacBio[1].

Recent works have demonstrated single-contig long-read assemblies of bacterial chromosomes [42, 59]. Therefore, it is natural to ask whether genome assembly is now a solved problem with long reads[2], at minimum for smaller genomes such as bacteria. It turns out that in several cases, bacterial assemblies remain fragmented into a handful of contigs, even with long-read sequencing and recent assembly techniques. Deciding whether an assembly instance is resolved is not always clear due to the presence of plasmids, contaminants and unplaced low-quality reads. In this work, an assembly is considered to be *resolved* if the number of contigs classified as chromosomal is equal to the expected number of chromosomes (generally just one, in the bacterial case).

To date, the NCTC project contains 1,735 samples for which 1,136 have been assembled by the consortium, and among these, 599 (34%) are unresolved according to the criteria above (as in Feb 2019). Later in this article, we will see that even when using multiple recent tools, many assemblies remain fragmented. Therefore there is a clear and unmet need for an investigation that determines whether those samples are intrinsically impossible to resolve, or whether current assembly methods are imperfect.

In this article we have selected a subset of NCTC samples (see Results section) and considered the outputs of three recent assemblers: `Canu`, `Miniasm`, and `HINGE`. We observe that instances where the assembly is fragmented can be challenging to further manually elucidate. In general, assemblers produce an assembly graph where nodes are contigs and edges reflect local sequence proximity in the genome (*adjacency*). In fragmented instances, the final assembly graph is sometimes uninformative due to the absence of edges between contigs, hindering further assembly finishing steps. In such cases, it would be tempting to conclude that the assembly is fragmented due to regions of insufficient sequencing coverage, with no way to determine a likely contig order. However, in a number of cases we found that a lack of connectivity can be due to reads that were discarded early in the assembly pipeline. Here we will show that contig adjacency information can be computationally recovered from the raw data.

To automatically investigate unresolved assemblies and propose directions for refinement, we introduce a set of *in silico* forensics operations for long-read assemblies, and we built a software framework. Our analyses are based solely on information present in the raw sequencing data in addition to the contigs produced by a given assembly tool, and are not biased by any other source, e.g. a closely related reference genome. For validation purposes only and to explain some of our observations, we will align contigs to a ground truth reference when one is available. Our framework is first tested on synthetic data to illustrate a simple case of fragmentation due to heuristics in the `Canu`

[1] https://www.sanger.ac.uk/resources/downloads/bacteria/nctc/
[2] See e.g. https://huit.re/PJMMA_uF

assembler. We then show on real data that our method helps recover useful adjacency information between contigs.

Going further, we demonstrate how to use this recovered information to provide likely assembly hypotheses using Hamiltonian paths, through a ranked list of contigs orderings. Obtaining a small set of possible orderings between contigs, knowing that the true genome order is likely one of them, can be instrumental to guide further genome finishing steps.

4.4.3 Related works

Assembly forensics date back to the Sanger era, e.g. with the AMOSvalidate software [84], which detects mis-assemblies within contigs using multiple sources of information (e.g. read coverage, properly mapped pairs, clipping). Other tools have been introduced for mis-assembly detection in Illumina data (REAPR [32], FRCbam [105], Pilon [106]) and for PacBio data (VALET [78]) using similar principles. Completeness of an assembly can be estimated without any reference, using core genes as a proxy metric, e.g. with BUSCO [97] or CheckM [81] software. Finally, assembly likelihood metrics have been introduced to assess the fit of an assembly to a probabilistic model of sequencing, via re-mapping reads to the assembly [23, 28, 85]. For a more complete exposition, refer to a recent survey on metagenomics assembly validation [79], that also largely applies to isolates.

For bacterial genomes specifically, several pipelines for *assembly finishing* have been developed [11]. They usually take as input an assembly obtained with short-read data and align it to one or multiple close reference genomes, in order to find a contig ordering [46]. Recent work has examined the cause of assembly fragmentation for seven bacterial genomes sequenced using PacBio sequencing, and rejected the hypothesis that gaps were caused by strong secondary DNA structure [102]. Instead, low coverage and repetitions appear to be the two main factors for contig termination.

To the best of our knowledge, little work has been carried to investigate assemblies based on the graph of assembled contigs or the initial string graph. Noteworthy exceptions are the Bandage software (an assembly graph visualization tool) [110], and the `HINGE` assembler that implements automated repeat handling based on the assembly graph [36]. We use Bandage extensively in the present work, and will consider datasets where even `HINGE` failed to produce a single-contig assembly.

4.4.3.1 Long-read assemblers

Several genome assemblers have been developed to process third-generation sequencing data, either stand-alone [36, 44, 54, 58] or in combination with Illumina data [6, 111, 114, 115]. In this work we will focus on three recent stand-alone assemblers, chosen because of their widespread usage (`Canu`), automated graph analysis algorithms (`HINGE`), and speed/modularity (`Miniasm`). However the techniques are likely to be applicable to a broader set of assemblers.

4.4.3.2 Description of `Canu`, `Miniasm`, and `HINGE`

The `Canu` [44] assembler consists of three major steps: correction, trimming and contig creation. The first two steps should not be regarded as innocuous pre-processing steps, as they significantly impact the rest of the assembly process. The correction step uses MHAP to perform all-against-all

read mapping then generates consensus reads with the falcon_sense tool [21]. `Canu` then performs overlapping of error-corrected reads with a legacy algorithm from the Celera assembler, named *ovl*. The trimming step detects hairpins, chimeric reads, and low-support regions and subsequently cuts reads. A 'unitigging' step is performed using `bogart`, a modified version of CABOG [67], to produce a graph that records only the longest overlaps between corrected reads (termed BOG for 'Best Overlap Graph'). `Canu` generates contigs from this graph and improves their consensus accuracy by re-mapping all reads.

The `Miniasm` pipeline consists of two separate tools: `Minimap2` and `Miniasm` [54]. `Minimap2` finds overlaps between raw reads and outputs alignments. `Miniasm` trims low-coverage regions of reads, then constructs a string graph from `Minimap2` alignments that are suffix-prefix overlaps. `Miniasm` performs simplification on the graph inspired by short-read assembly: transitive reduction, tip removal, bubble popping, and short overlaps removal based on a relative length threshold. After simplifications, non-branching paths are returned as contigs.

The `HINGE` [36] assembler uses raw uncorrected reads (similarly to `Miniasm`) to construct an overlap graph similar to the BOG of `Canu`. `HINGE` attempts to output finished bacterial assemblies through improved repeat-resolution. In cases where there subsist repetitions that are not spanned by reads, `HINGE` provides a visualization of the resulting assembly graph for manual inspection.

4.4.3.3 Assembly graphs

Short-read and long-read assemblers output final assembly sequences in FASTA format, and an increasing number of tools also output an assembly graph in Graphical Fragment Assembly (GFA) format[3]. A final long-read **assembly graph** typically consists of all contig sequences as nodes, and a set of overlaps between contigs as edges. Assembly graphs

Most long-read assemblers start by constructing then analyzing a *string graph* (**SG**) of the reads [72], where each read is a node, and overlaps between reads are represented by edges to which additional information is attached (e.g. overlap length, overlap error rate). In addition, transitive reduction is performed on the edges and reads that are fully contained in others are discarded.

4.4.4 Methods

We hypothesized that the final contig graph produced by assemblers does not always reflect all the information present in the raw data, and may be missing overlaps or even genomic regions. We built a novel algorithmic framework to recover some of the 'missing' information and further analyze it. The main steps are presented in Fig. 4.14, and the next sections describe them in more details.

4.4.4.1 Raw string graph

First, we eliminate chimeric reads from the raw data based on overlaps found by `Minimap2` using a custom tool[4] (manuscript in preparation [63], see Supplemental Fig. A.6). A string graph (SG) is then constructed using overlaps between chimera-removed reads (here, overlaps found by `Minimap2`).

[3] https://github.com/GFA-spec/GFA-spec
[4] https://gitlab.inria.fr/pmarijon/yacrd

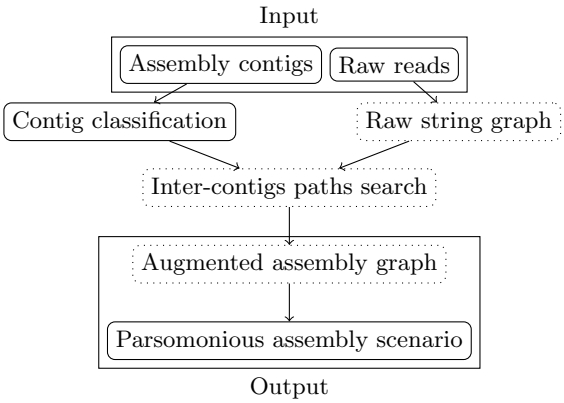


Figure 4.14: The proposed framework takes as input raw long-read sequencing data and the output of an assembler. The (optional) contig classification step removes non-chromosomal contigs. A string graph of raw reads is constructed, in which paths are searched between extremities of contigs, then are converted into links between contigs in an augmented assembly graph. When such a graph is connected, putative contig orderings are reported. Dotted nodes represent elements that are automatically visualized in the HTML report.

A stand-alone script was created to convert overlaps from the PAF format (defined in [54]) to a graph in the GFA format[5]. Transitive reduction over the edges of this SG is performed using Myers' algorithm [72].

4.4.4.2 Contigs classification

In order to simplify analyses and focus on chromosomal contigs, we filter out contigs of plasmid origin and contigs of unknown taxonomic status (see Supplemental Methods A.1). Contigs that were not marked as chromosomal are discarded. Note however that this contig classification step can be skipped in order to perform analysis of complete, unfiltered sets of contigs.

4.4.4.3 Computation of paths between contigs

An essential algorithmic component of our framework is the search for paths in the SG that uncover new connections between contigs. First, one read per contig extremity is identified among reads included in the SG: a read is selected such that both its incoming and outgoing neighbors also map at the same contig extremity (in order to avoid selecting dead-end nodes in the SG).

Then for each pair of contigs, shortest paths between reads at both extremities of each contig are computed in the SG using Dijkstra's algorithm. The length of a path is computed in nucleotides as follows: the sum of all reads lengths involved in the path minus all the overlaps between reads, as well as minus the overlaps between reads and contig extremities. If contigs overlap, the path length is reported as zero. Since we perform path search starting from each contig extremity, we may obtain two shortest paths for each pair of contigs, and only the shortest of those two is kept.

[5] https://gitlab.inria.fr/pmarijon/fpa

4.4.4.4 Augmented assembly graph

We transform a contig graph into a novel object, the *augmented assembly graph* (**AAG**), as follows. Nodes of the AAG are contig extremities. An edge is inserted between two nodes if a path has been found by the procedure in Section 4.4.4.3 between the two contig extremities. Each edge is weighted by the corresponding path length. Additionally, zero-weight edges are created between both extremities of each contig.

Such a graph allows to explore adjacencies between contigs, beyond those present in the original contig graph, in order to formulate hypotheses regarding the ordering of contigs. At a certain contig extremity, and in absence of genomic repeats, low-weight edges likely reflect adjacent contigs, while high-weight edges likely correspond to SG paths that pass through other contig(s) (i.e. transitively redundant edges in the AAG). In the presence of repeats, low-weight edges do not necessarily show true adjacencies between contigs, as the true path may be longer. Yet one can observe that a path longer than the longest repeat in the genome necessarily reveals a distant link between two contigs (i.e. necessarily contigs which are truly non-adjacent on the genome), and also such path may go through another contig.

According to [101] most repetitions in bacteria are shorter than 10kbp. We thus categorize edges of the AAG into 3 groups according to their weight. Consider the path in the SG that led to the creation of the edge e in the AAG between extremities of two different contigs a and b. If the path is longer than 10kbp, and/or it contains at least one read that was involved in the construction of another contig c, the edge e is named *distant*. Otherwise the edge e is considered to reflect an adjacency between a and b. If there is more than one edge outgoing from the extremity of a or of b, the edge e is named a *multiple adjacency* (likely revealing a putative repeat). Otherwise it is named a *single adjacency*.

4.4.4.5 Searching for parsimonious assembly scenarios

We sought to determine whether contigs could possibly be ordered directly using the AAG. In principle, we anticipate to recover a large number of distant edges in the AAG, therefore it would be non-trivial to determine a contig order by direct inspection of the graph layout (e.g. see Fig 4.16). Given a connected AAG, our working hypothesis is that a minimum-weight Hamiltonian cycle may correspond to the correct contig order (note that having a connected AAG is a necessary condition for such a cycle to exist, but not a sufficient one). This is guided by the intuition that edges in the AAG with high weight are more likely to correspond to false connections due to repetitions or true paths between distant contigs. For simplicity, we search for Hamiltonian cycles and not paths, under the assumption that the genome is circular. We further require that any Hamiltonian cycle traverses all zero-weight edges corresponding to both extremities of each contig. Moreover, contigs mapping inside another one are not considered.

We designed an automated procedure to test this hypothesis, based on computing and sorting Hamiltonian cycles according to their total edge weights. In practice some of the AAGs that we obtain are too complex, due to the presence of short contigs (see the Discussion section for more details). Our pipeline excluded contigs shorter than 100kbp from the AAG before listing all Hamiltonian cycles. For validation purposes, when a reference genome is available, we mapped all chromosomal contigs

against this reference to determine the true contig order. We then recovered the position of the true contig order within the list of orders given by Hamiltonian cycles.

4.4.4.6 Assembly report generation

We implemented a Snakemake [45] pipeline that takes as input raw reads, contigs produced by an assembler, and optionally a contig graph. The pipeline follows steps described Fig.4.14, then generates an HTML report for easy inspection. Companion tools to compute AAG edge classification and to perform Hamiltonian path search are also provided.

4.4.5 Results

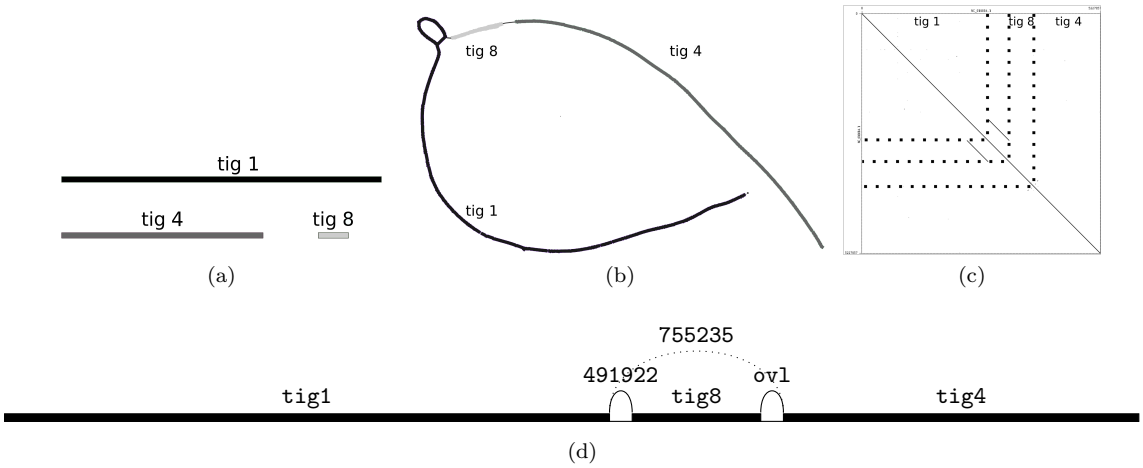


Figure 4.15: Graph analysis of a synthetic dataset (*T. roseus*). (a) Contig graph produced by `Canu` (visualized using Bandage): 3 contigs, no edge. (b) SG built from `Minimap2` overlaps, on which connected components of the `Canu` BOG are colored. (c) Dot-plot of the *T. roseus* genome (NC_018014.1) aligned against itself, showing a long tandem repeat. (d) The AAG with `Canu` contigs ordered according to their position on the *T. roseus* reference. If two contigs overlap, no length is given and instead the link is labeled 'ovl'.

4.4.5.1 Datasets

In order to illustrate our methods using a simple yet non-trivial case of assembly graph analysis, we simulated long reads from a linearized reference genome of *Terriglobulus roseus* (NC_018014.1, 5.2 Mbp). This genome contains an unusual 460kbp repeat that is challenging for assembly tools. We used LongISLND [49], with 20x sequencing coverage and 9kbp mean read length (Supplemental Table A.9).

To investigate real datasets, we mined the NCTC project which consists of 1735 bacterial strains (as of Feb 2019) sequenced using PacBio technology. For each dataset, the NCTC consortium had built an assembly using HGAP and Circlator [33] followed by a manual correction step. We estimate, based on visual inspection of 159 NCTC fragmented `HINGE` assemblies[6] out of 997, that assembly

[6] https://web.stanford.edu/~gkamath/NCTC/report.html

graphs are missing contig adjacency information in 69% of the fragmented assemblies of `HINGE` and `Miniasm`, i.e. around 13% of all NCTC datasets (including those that assemble perfectly). Among datasets for which both `Canu` and `HINGE` failed to produce a single contig per chromosome, we selected 19 datasets where the assembly made by NCTC contains as many chromosomal contigs as the number of expected chromosomes (i.e. is resolved), 24 datasets where the NCTC assembly is unresolved, and finally 2 datasets that were not yet assembled by NCTC. See Supplemental Table A.2 for a complete list of the 45 datasets. All datasets were assembled with `Canu` version 1.7 and `Miniasm` version 0.2.

`Canu` contigs were classified according to Section 4.4.4.2. On average for each dataset, 10.2% (resp. 6.4%) of the `Canu` (resp. `Miniasm`) contigs are marked as plasmid, 13.7% (resp. 12.2%) do not match any bacteria in the Blast database and are therefore marked as of undefined origin, and the remaining 76.0% (resp. 81.3%) of contigs are classified as chromosomal and are further considered for analysis. Full classification results are presented in Supplemental Table A.6 and A.7.

We further investigated whether the assemblies could somehow be combined, e.g. by improving `Canu` assemblies using `Miniasm` contigs. We have performed a simple test to evaluate this possibility (see Supplemental section A.2) and could not straightforwardly improve assemblies this way.

4.4.5.2 Assembly graph analysis of a synthetic low-coverage dataset

This section gives an introductory overview of the analyses that our method performs on the *T.roseus* simple synthetic dataset described above. `Canu` produced 3 contigs of total length 4.7 Mbp. A ≈500kbp region is missing from the assembly. `Miniasm` produced 7 contigs and the `HINGE` assembler (commit 8613194) was not able to produce an assembly, likely because of the low coverage (20x).

Since the SG has a single connected component (Fig. 4.15b) but both the BOG and the contig graph of `Canu` have multiple connected components (Fig. 4.15a), assembly fragmentation can be explained by reads that have been discarded at the BOG construction stage of `Canu`. The coloring of the SG using the connected components of `Canu` BOG (Fig. 4.15b) further suggests an ordering of contigs. Note that the `Canu` contig graph is uninformative on this dataset, as it contains no edges between contigs.

We performed path analysis as per Section 4.4.4.3. Fig. 4.15d shows the length of paths in SG found between reads at `Canu` contigs extremities. Since a reference genome is available, the true order of contigs is reported on the Figure but note that path analysis does not need this information. We find that the `Canu` contigs named tig8 and tig4 overlap in the SG. tig1 and tig8 are linked by a long path involving 491922bp. This long path can be explained by looking at how tig1 has been built by `Canu`: the path goes through a large 'loop' (see Supplemental Fig. A.2) which corresponds to a repeat in the reference (Fig. 4.15c). The repeat (of length 460kbp) was not resolved by `Canu`, leading to a region of about 440kbp missing from the assembly between tig1 and tig8, which explains why the shortest path between both contigs contains as many as 491922bp. We further checked that the path of length 755235bp between tig1 and tig4 indeed contains reads from tig8, and is therefore redundant. By aligning raw reads and `Canu` corrected reads to the reference genome, we observe a drop of raw reads coverage (around 8x) in the region between tig8 and tig4. This likely explains why `Canu` failed to connect both contigs.

As a side note, a `Canu` assembly of the same dataset with twice higher read coverage (40x)

yielded a two-contig assembly, also with same pattern as in between tig8 and tig4. An older version of `Canu` (1.6) fully resolved the 40x dataset into a single contig, likely due to changes in how reads are corrected and trimmed between version 1.6 and 1.7.

4.4.5.3 Investigation of 45 unresolved NCTC assemblies

We performed the same type of analysis on the 45 NCTC samples. A `Minimap2` AAG was constructed for each dataset using SG and `Canu` contig extremities. Assembly and AAG statistics are presented in Table 4.2 for an excerpt of the dataset. Full statistics and more details are given in Supplemental

	NCTC contigs			`Canu` contigs			# nodes	# dead-ends in		# edges in AAG		
NCTC ID	chr	pls	und	chr	pls	und	in AAG	contig graph	AAG	total	single adjacency	multiple adjacency
NCTC10006	1	0	0	3	0	0	4	2	2	4	2	0
NCTC10332	1	0	0	12	0	0	8	8	4	24	0	3
NCTC10444	1	0	0	7	0	0	8	3	0	24	0	6
NCTC10702	1	1	1	3	3	0	4	4	4	4	0	0
NCTC12123	2	3	0	5	4	1	6	4	1	12	1	2
NCTC12132	1	0	0	2	0	2	4	4	2	4	1	0
NCTC13125	1	2	4	6	3	1	6	0	0	12	0	4
NCTC13463	1	1	4	5	2	2	4	0	0	3	2	0
NCTC5050	2	3	0	4	2	3	6	6	0	12	3	0

Table 4.2: Assemblies and contig graphs statistics for an excerpt of 9 NCTC datasets (full tables in Supplemental Table A.2 and A.3), consisting of 8 datasets where Hamiltonian cycle search succeeded, and the NCTC5050 dataset discussed in the Results section. AAGs are constructed using a SG built from `Minimap2` overlaps and `Canu` contig extremities. The 'contig graph' column corresponds to the final assembly graph produced by `Canu`; 'chr': number of chromosomal contigs; 'pld': number of plasmid contigs; 'und': number of other contigs. Note that some of `Canu` 'chr' contigs may be contained in others, therefore the '# nodes in AAG' column corresponds to twice the number of non-contained contigs.

Tables A.2, A.6 and A.7. There we observe that the number of contigs in `Canu` and `Miniasm` assemblies is generally higher than in the assemblies made by NCTC. Nevertheless the sum of lengths of chromosomal contigs is about the same in all assemblies (Supplemental Table A.8).

Case study of two NCTC datasets We closely examine two NCTC datasets that contain interesting patterns, through the lens of a ground truth obtained by remapping `Canu` contigs against respective NCTC assemblies using BWA-mem [53].

NCTC12123 This dataset was assembled into 5 chromosomal contigs by `Canu`, including 2 contigs that are contained in others and are automatically discarded by our pipeline (see Fig. 4.16).

The assembly is made of 2 large contigs (tig1 and tig2) and a shorter one (tig9) totaling 4.78 Mbp. `Miniasm` produces also 5 chromosomal contigs, including 3 small ones. Both `Canu` and `Miniasm` contig graphs are made of two components. `HINGE` produces a single-component assembly graph but does not resolve it (because it detects multiple possible traversals). Finally, the NCTC assembly consists of 2 chromosomal contigs: one being 4.69Mbp long and the other 21kbp long. Contigs tig1 and tig2 both map over the large NCTC contig, while tig9 maps to both NCTC contigs. Using the AAG on `Canu` contigs (see Fig. 4.16), one can observe that a number of scaffolding scenarios could be made following this graph. Interestingly, based on the mapping of the 3 contigs on the larger contig of the NCTC assembly, edges of smaller weight (i.e. shortest paths) tend to be associated with true

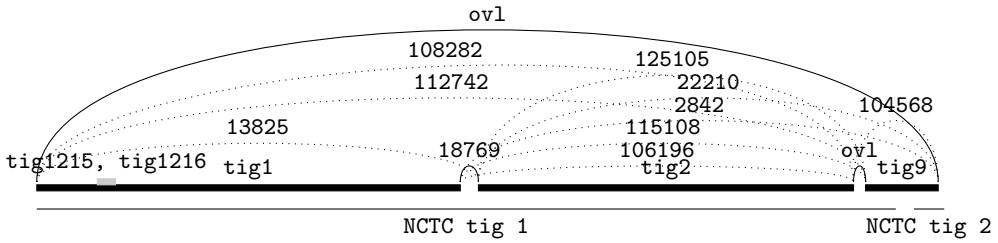


Figure 4.16: Mapping of `Canu` contigs (bold horizontal lines) against NCTC12123 assembly (the two thin horizontal lines). Links between contigs give the length (in bp) of the shortest path in SG between reads at extremities. If two contigs overlap, no length is given and instead the link is labeled with 'ovl'. Plain links are paths that are compatible with the sequential order of contigs given by mapping to the NCTC assembly, and dotted links are all other paths.

contig adjacencies. In this example, low-weight Hamiltonian cycles (Section 4.4.4.5) yield two likely contig orders (see Supplemental Fig. A.3). This SG analysis thus enabled to retrieve an adjacency that was missed by `Canu`. It also confirms the multiple traversals prediction of `HINGE`, further reducing the number of putative contig orders to only two.

NCTC5050 This dataset is assembled into 4 chromosomal contigs by `Canu`, including one that is contained in another. The `Canu` contig graph is 'fully' fragmented as each contig is its own connected component. There is no reference genome for this strain, and we chose as ground truth the NCTC assembly consisting of 2 contigs. One is entirely covered by a `Canu` contig, and the other contains the 3 remaining contigs (see Supplemental Fig. A.4). In the following, x_s and x_e denote left (resp. right) extremities of a contig x. We found *single* (i.e. non-repeat) adjacencies between $\text{tig1}_s/\text{tig23}_s$, $\text{tig1}_e/\text{tig10}_s$, $\text{tig10}_e/\text{tig23}_s$ that were confirmed by mapping to the longest contig from the NCTC assembly. Together, these single adjacencies suggest a putative scaffolding scenario: tig1 – tig10 – tig23(reversed). This scenario is also the top-ranked one proposed by our Hamiltonian path search procedure (see below).

We also mapped corrected and raw reads to the junction for validation (see Supplemental Fig. A.5). We observe a drop of coverage at this location (see reads mapping in Supplemental Fig. A.5) that is likely the cause of assembly fragmentation. Therefore, again in this dataset the path search operation enabled to recover a link between contigs that was discarded by the assembler due to a drop in sequencing coverage.

Path search enables to recover adjacency between contigs Table 4.2 reports statistics of paths found between `Canu` contigs by our method for a subset of 9 NCTC datasets (for the full dataset, see Supplemental Table A.3). We first focus on unambiguous contig adjacencies recovered by our pipeline. Single adjacency edges are only found in 6 out of 9 datasets, yet across the entire dataset of 45 samples, 60.4% of all single adjacency edges (43 in total) are found in samples that have a sequencing coverage below 38x, and only 17 single adjacency edges are found in datasets with coverage above 38x. This is likely due to the error-correction step in assemblers that is less effective in low-coverage datasets (even when the true sequencing coverage is given to the assembler as a

Mean number of	
`Canu` contigs	4.32
Edges in AAG	32.67
Theoretical max. edges in AAG	41.83
Distant edges	28.64
All adjacency edges	4.02
Single adjacency edges	1.16
Multiple adjacency edges	2.86
Dead-ends in `Canu` contigs	4.94
Dead-ends in AAG, adjacency edges	2.70

Table 4.3: Average statistics of augmented assembly graphs using a SG built from `Minimap2` overlaps on `Canu` contigs across the 38 NCTC datasets with two or more contigs, after size and classification filters. All rows are as per definitions in Section 4.4.4.4. 'Theoretical max. edges': number of possible edges in each AAG. 'Dead-ends in AAG, adjacency edges': number of dead-ends in the AAG when only adjacency edges are considered, i.e. distant edges are deleted.

parameter), which in turn causes assembly fragmentation. Our method therefore enables to recover single adjacency edges between contigs that were fragmented due to this effect.

To measure whether the `Canu` contig graphs could be used as-is to recover contig order, we counted the number of contig extremities that are not linked to any other extremity (i.e. *dead-ends*). Those are contigs for which no chromosomal order can be reliably inferred. In 35 out of the 45 datasets (7 out of 9 in Table 4.2), the `Canu` contig graph has some dead-end extremities (between 1 and 23). In principle dead-ends extremities should not exist in circular bacterial assembly graphs, except for linear chromosomes. Assemblers, here `Miniasm` and `Canu`, do not report all true contig adjacencies. In contrast, our method enables to recover some of these adjacencies and lower the number of dead-ends in 23 out the 37 datasets (and all but one dataset in Table 4.2).

Table 4.3 summarizes average AAG statistics over all 38 datasets on `Canu` contigs (per-dataset results in Supplemental Table A.3). Results for `Miniasm` contigs are shown in Supplementary Tables A.4 and A.3. On average, `Canu` contig graphs contain 4.32 nodes (5.86 extremities), among which 4.94 extremities are dead-ends. The AAG enables to reduce the number of dead-end extremities to 2.7 (45% lower), through the discovery of 1.16 single adjacency edges and 2.86 multiple adjacency edges in the AAG per dataset on average. The reduction is also significant for `Miniasm` contigs but not as high (31%, Supp. Table A.4). Note that these adjacencies are 'real' in the sense that they are all supported by paths of overlapping reads of total nucleotide length less than 10kbp, yet a number of them may be caused by repetitions. An upper bound on the ability to mine paths in the SG is given by the theoretical maximal number of edges in the AAG (41.83 edges). Our method is on average 78% close to this bound for `Canu` contigs (resp. 90.1% for `Miniasm`) as it discovered 32.67 edges per dataset (resp. 85.1). We note that large fraction (87%) of discovered edges were classified as distant edges, yet the remaining adjacency edges are informative as they significantly contribute to removing dead-ends in the contig graph.

Contig order search retrieves parsimonious assembly scenarios While the work done in the previous section helps to recover contig adjacencies, the presence of multiple adjacency edges due to repetitions often prevents us from unambiguously inferring a contig order. We applied the Hamiltonian

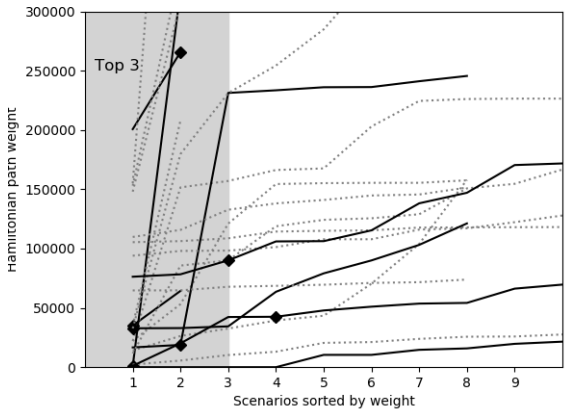


Figure 4.17: Weights of scenarios in AAGs. Each curve correspond to the sorted list of Hamiltonian cycles, sorted by weight. If a ground truth is known, a diamond symbol marks the correct assembly scenario. Extended Figure available in Supplementary material A.1

cycle procedure presented in Section 4.4.4.5 to determine likely contig orderings. Fig. 4.17 shows orderings sorted by weight across 23 datasets on which the method could successfully be executed (connected AAG, low number of edges).

A ground truth is known in only 8 of those datasets. Among them, the lowest-weight scenario is ranked first in 3 datasets, 2nd in 2 datasets, 3rd in 1, 4th in 1 and 38th in the last one.

These results suggest that the correct assembly scenario is likely to be one of the top predictions made by our parsimonious Hamiltonian cycle procedure. However finding many fragmented datasets that also have a ground truth is inherently difficult, thus further work is needed to confirm this hypothesis. Also, datasets where several scenarios have similar weights (i.e. curves that 'plateau' in Fig 4.17) will possibly be more challenging to resolve using this method. Yet for many samples with fragmented assemblies, parsimonious assembly scenarios are a promising approach to explore a limited number of hypotheses that could further be validated using long-range PCR to finish the genome.

4.4.6 Discussion

We presented a set of concepts to provide novel insights on fragmented long-read bacterial genome assemblies.

By searching for paths of overlapping raw reads between extremities of contigs, we construct an *augmented assembly graph* that recovers unreported adjacencies between contigs. We demonstrate several usages of this graph: to provide a more informative representation of fragmented assemblies, to examine repeat structures, and to propose likely contig orderings. In our tests, the AAGs of NCTC datasets recover edges for nearly half (45%) of the dead-end nodes in `Canu` contig graphs, on average. We further show a link between the lowest-weight Hamiltonian cycles in the AAG and the true contig order. We highlight that our method solely relies on the raw data and information produced by assemblers at various stages of their pipelines and, when our contig classification step is skipped, no reference genome nor external information (e.g. genome map, BLAST database) are used.

Our method hinges on directly constructing a string graph on the raw reads, after a relatively conservative chimera removal step. Doing so avoid biases that may be introduced in the read trimming and error-correction steps of an assembler. Indeed, overlaps between reads may become shorter or even absent after error-correction. For instance on the 45 NCTC datasets that we analyzed, the number of edges in SGs built from `Canu` error-corrected reads is reduced by 41.4% compared to the SGs of raw reads. We have classified edges in the AAG, by considering their underlying nucleotide lengths and whether they contain reads that belong to other contigs. To go further, one could define confidence metrics, e.g. based on local graph structures.

Due to a combination of engineering choices and the inherent difficulty of visualizing large assembly graphs, our software has only been tested on bacterial genomes and is unlikely to readily run on larger genomes. However, the techniques presented here (AAG, path search between contig extremities, weighted Hamiltonian cycles) are not specific to bacterial assembly, and should in principle be applicable to small and large eukaryotes. However more work would be needed e.g. to scale path search to thousands of contigs, refine thresholds (contig filter, adjacency edges), handle inter-chromosomal repeats, and an evaluation of the relevance of Hamiltonian cycles for larger genomes.

We stress that our techniques currently do not aim at detecting misassemblies within contigs. We also did not focus on the difficulty of running multiple assembly programs, but we note that the process has previously been reported to be challenging [48]. Our work is also orthogonal to assembly reconciliation [2], which consists of constructing a higher-contiguity assembly by merging the results of multiple assemblers.

No attempt was made to optimize the detection of overlaps between reads though this could be a direction for improvement. Finally, automatic post-assembly improvements based on the AAG would be a natural extension of this work. One could use the AAG to design an oracle that suggests a limited number of (long-range) PCR experiments for resolving individual repeats.

Acknowledgements

This work was supported by an Inria doctoral grant and the INCEPTION project (PIA/ANR-16-CONV-0005). The authors are grateful to Samarth Rangavittal, Monika Cechova, Jason Chin, Jason Hill and Christopher W. Wheat for discussions that led to this project, Gautam Kamath for guidance on reproducing NCTC analyses with `HINGE`, Antoine Limasset and anonymous reviewers for helpful comments on the manuscript.

4.5 Conclusion

In this section we studied how we can evaluate an assembly, and detailed some issues when we use a reference genome to evaluate a de novo long-read assemblies.

With `KNOT` we present the interest to go back to raw read information, and how it can solve bacterial assembly issues. To use `KNOT` on more complex datasets needs to improve some parts of `KNOT`, especially the graph construction, its representation in memory and the search of paths between contigs extremities. These improvements required some development, but the original idea of going

back to raw reads information can be used for more genome assembly improvements.

Chapter 5

Other contribution

This chapter have not any link to other ones they presents about my contribution on some projects where I spent some time during my PhD without sufficiently large contribution to have a specific chapter.

5.1 Labsquare

Labsquare is a community for genomics software, this community was create by me and some other bioinformaticien's friend I participate in developpement of two tools.

`FastQt` is a rewrite of `FastQC` in C++ with the framework `Qt`. `FastQt` is a tool to check the quality of sequencing data by providing some statistics, GC% distribution, read length distribution, error rate repartition along the length of reads. At the moment `FastQt` development was stopped.

CuteVariant is a tool to visualize and analyze `VCF` (for Variant Call Format) files, these file store variants found between an individual or a dataset of reads against a reference genome. `CuteVariant` allows selecting annotation, genotype, filter variant, sort and group variants, set operation between VCF file. To perform all this operation a query language was create the `VQL` (for Variant Query Language), `CuteVariant` is still in development and it was the subject of a poster during the conference Jobim 2019.

5.2 CAMI challenge 2

`CAMI` challenge is a metagenomics assembly challenge, I participate in the second edition of CAMI challenge with Camille Marchet, Antoine Limaset, Pierre Peterlongo, Claire Lemaitre and Rayan Chikhi. We tried different strategies to perform an assembly of metagenomics datasets.

5.2.1 Dataset description

In this challenge, we have two datasets with similar reads composition. A set of Illumina Hiseq like reads and a set of Pacbio like reads, simulated by `CAMISIM` [27].

We have two datasets, Marine Dataset built to correspond to the composition of a metagenomics sequencing of seawater, and Strain Madness Dataset with very important strain-level variation.

5.2.2 Assembly strategy

On the first hand we perform a short reads assembly with `gatb-pipeline` [1], that is a multi-kmer size short read assembly based on `Bloocoo` [8] for read correction, `Minia 3` [16, 17] for contig assembly and `BESST` for scaffolding [88–90].

On the second hand, my main contribution in this challenge, we build a long-read assembly pipeline with `fmlrc` [107] a hybrid long-read corrector, and assembly of corrected long read with a pipeline `Minimap2`, `fpa` (no internal match and overlap lower than 2000 bases), `Miniasm` or a `wtdbg2` assembly. We try to perform read classification before correction and assembly with `centrifuge` [40] to avoid the complexity of metagenomics dataset because our correction assembly tools are not built to use this type of data, but we did not use this strategy due to lack of time.

Finally, we submit a reconciliation of short reads assembly made by `gatb-pipeline` and `wtdbg2` assembly of corrected long-read, if a short reads contig maps in a long reads contig the short reads contigs is discarded. This strategy should have allowed us to have good quality contigs (from long reads assemblies) on the most present strains without losing the information of the least present strains (contained in the short reads contigs). Indeed, long reads sequencing technologies have lower sequencing depths which cannot allow the detection and assembly of the least present strains.

When writing this document, we do not have yet the result of other teams or an idea of quality of our assembly.

5.3 10X linked-read deconvolution

10X linked-read sequencing is a sequencing technique developed by 10X genomics. Figure 5.1 presents the main idea of 10X linked-read sequencing. After purificationn DNA is fragmented into large molecules (≈100 kb length). By microfluidic method each large molecule is separated into an individuals bubble. Each bubble is associated to a barcode. In a bubble DNA is fragmented into shorter fragments (compatible with Illumina sequencing method) and a barcode is added to the extremity of each fragment. After a classic short-read sequencing, we can use barcode information to determinate if read comes from the same large fragment or not.

Unfortunately, there is not a barcode for each large DNA molecule and therefore several fragments will share the barcode. The task of assigning each reads its original molecule is called **deconvolution**. Knowing exactly the original molecule of each reads is useful to:

- assembly and scaffolding, by allowing to solve repetitions that are spanned by large molecules,

[1] https://github.com/GATB/gatb-minia-pipeline

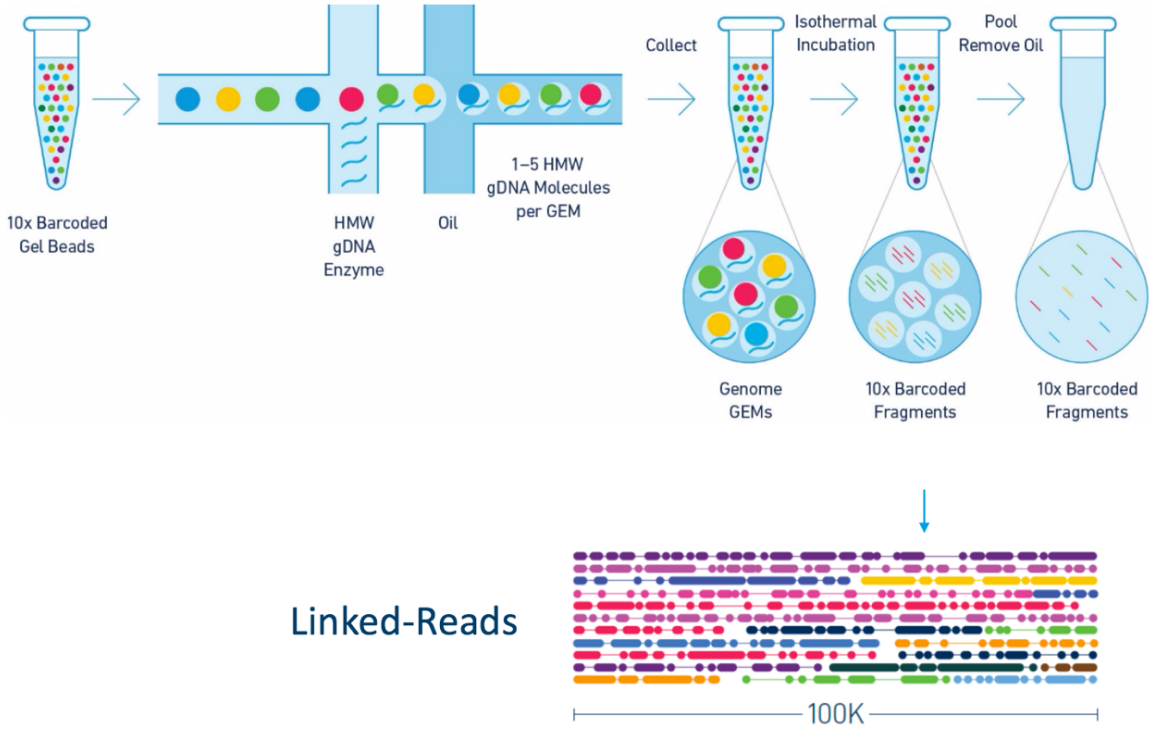


Figure 5.1: 10X linked-read sequencing idea. Source: https://ucdavis-bioinformatics-training.github.io/2018-Dec-Genome-Assembly/10x-supernova/10x-supernova.html

- variant phasing; all reads coming from same molecule necessarily come from the same haplotype.

If we have access to a good genome reference, deconvolution is easy: by mapping reads against the reference, we can look for reads of same barcode that map within approximately a 100 kb range on the reference genome. Those reads then likely come from same molecule. This general idea used by `ema` [95] and `Lariat` [10] to assign a read to a molecule.

But when we don't have a good reference genome we cannot use this method. I proposed a method to perform deconvolution based on assembly graph analysis. After a classic `DBG` assembly (using `bcalm`), we remap reads against contigs with the `ema` software. Reads with same barcode and map on same contig are assigned an identical premolecule identifier by `ema`. We attempted to glue clusters of reads by analyzing the contigs graph. For all clusters of premolecules with the same barcode, we searched for the shortest path in the contigs graph between clusters. If the length (in number of bases) of a path is shorter than a threshold, we merge both premolecule clusters.

At this stage, we did not perform a complete evaluation of the method, thus this is still work in progress.

Chapter 6

Conclusion

In this work, we aimed to improve the process of long-read genome assembly, without creating a new assembly tool. We have designed tools that work before and after assembly. These tools can be easily integrated into a workflow. The underlying idea is to improve assembly pipelines one tool at a time.

Building pipelines with a collection of tools that perform simple tasks, makes it easier to provide independent improvements to each task separately. It enhances the re-usability of each component, and the flexibility of the pipeline usage. Many assembly pipelines are a set of difficultly configurable black boxes, which does not help the user to adapt assembly tools to their own problem. Applying UNIX philosophy "Doing only one thing, and doing it well" on genome assembly could save the time of the community and improve results, as shown in the Hackseq 2018 Genome Assembler Components project[1]. Modular assembly should be the route to design versatile tools, able to be easily tuned to specific tasks, while understanding and keeping under control each step.

`fpa` was created after a reflection on information generated by overlapping tools and its impact on disk space. Many overlaps are not useful for all analysis, for example `Miniasm` keeps only end-to-end overlaps, thus storing all overlaps found by `Minimap2` on disk is a waste of disk space. Moreover, writing and reading these overlaps takes times. `fpa` not only filters overlaps, but also can rename reads in overlap (to reduce disk memory impact of overlaps), generates a `GFA1` overlap graph, – or index the position of overlap in output file. This functionality was used by `CONSENT` [68]. `fpa` was used to avoid the necessity of writing one's own filters, Erik Garrison uses it to simplify his work on `seqwish`[2], a tools to create pangenome graph.

`yacrd` uses coverage information as a proxy of reads region quality, it's a simple idea already present in correction tools. However, `yacrd` extracts this functionally out of correction tools, increasing the modularity of pipelines. This helps to improve each step of the pipeline separately, to choose the relevant tools for specific data and analysis. A pre-publication version of `yacrd`, with chimeric detection only, was used in a long read microbiota profiling pipeline to clean chimeric reads [24] and to improve some `Flye` assemblies.

Some improvements can be made on `yacrd` pipeline. To detect bad quality regions, `yacrd` uses `Minimap2` with a specific parameter, to avoid the creation of a bridge between two good quality

[1] https://github.com/hackseq/modular-assembly-hs18
[2] https://github.com/ekg/seqwish

			In KNOT graph		KNOT graph
	# bases	# reads	# Nodes	# Edges	construction time
E. coli	1621000527	158590	24966	158590	2 hours
D. melanogaster	9064470438	1327569	234253	956929	3 days
ratio	5.59	8.37	9.38	6.03	36

Table 6.1: A comparison of two Nanopore datasets. The ratio was computed by dividing *D. melanogaster* value by *E. coli* value. The size of data increases by less than an order of magnitude but the construction time increases more than 2 orders of magnitude.

regions over the bad quality regions. A solution like `miniscrub` was to use the seed position as a proxy of a quality region instead of the overlap, to directly avoid this trouble. Another solution was to replace minimizers by seed with error to find a similar region between reads over sequencing errors. Replacing `Minimap2` by tools using seeds with error to estimate the coverage of reads regions, was probably improved by these tools. Some overlapping tools use this idea, like `GroupK` [25].

`yacrd` takes a very global point of view on the composition in bases and the quality of the reads, avoiding the problems of masking heterozygosity that can still be observed today in correctors. But the problem of the accentuation of coverage gaps by which we have been able to observe and solve with to `KNOT` is potentially always present in `yacrd`. Indeed, if we follow the recommended parameters and a region of the genome is sequenced at a depth of less than 3 `yacrd` will create a coverage hole. If we want to avoid this problem we would need to have a broader analysis of the problem, not this focus on a single read at a time, potentially through the construction of local overlap graphs around the reads. This work can be apply to scrubbing and correction tools, but this change in perspective will probably take time and some many development to have equivalent performance of actual tools.

`KNOT` is a tool to retrieve missing connections between contigs. `KNOT` uses `Minimap2` to find overlap between reads and between contigs, `yacrd` to remove low quality reads from raw reads dataset and `fpa` to filter overlaps and generate overlaps graph; then a script in `KNOT` performs path search within this graph. The main idea behind `KNOT` is that sometimes we have to consider all the available data to solve a problem. At the moment, assembly pipelines try to keep only the minimum amount of information to solve the assembly problem (cf Chapter 3) and this is a very good approach that allows to accelerate the assembly in a very important way. But sometimes, this reduction of information goes too far and important information is lost. `KNOT`, by going back to the original information and focusing only on unresolved points, tries to correct these errors.

This idea to go back to the total information can however become a trouble for `KNOT`. The size of `KNOT` overlap graph is very important for example in Table 6.1. We can see two Nanopore datasets, one from *E. coli* and one from *D. melanogaster*. *D. melanogaster* dataset is larger than *E. coli* dataset, less than 10 times. But the computation time to build `KNOT` overlap graph from the overlaps found by `Minimap2` was increased by 30 times.

To use `KNOT` on a large datasets, we need to change how we use this graph. Currently `KNOT` loads all graphs in memory, however we don't need all this information to be permanently loaded into memory. We could load only one part of the graph at a time. Another trouble with larger datasets concerns genome with more than one chromosomes. At this time we did not try to prevent the creation of false links between contigs for different chromosomes. To adapt `KNOT` to larger genomes, we have

to solve a technical problem on how to represent these very large graphs in memory. We must also tackle a more theoretical problem of how to ensure that we do not create links between contigs of different chromosomes.

If `KNOT` needs to be updated to be run easily on larger datasets, ideas behind`KNOT` can also lead to more features. We use `KNOT` overlap graph to refund the lost link between contigs, we focus our analysis of graph on contigs extremities. But by performing some graph analysis along of the contigs, we can maybe detect misassemblies. To do this, we can draw inspiration from `Canu` repetition detection module (see 3.7) or something not to far to `tigmint` [34].

The current version of `KNOT` has total confidence in the contigs given as input, while the future evolution could may be mark some spurious region or break contigs. Analysis of all reads information can lead to another extension. I think we can convert contigs and `KNOT` overlap graph information to a genome graph. A genome graph is a new type of genome representation, which replaces a linear representation of genome by a graph where each nucleotide is a node, and an edge is created if nodes follow in the genome. This type of structure could be useful to solve the limitations of reference genome approaches. For example `WhatsHap` [64], a tool to phase variant, perform a mapping of read against the reference. When `WhatsHap` found a mismatch in mapping, he need to build a small new version of genome according variant database. `WhatsHap` perform remapping of read against this new reference to confirm the read dataset contains effectively a known variation of this genome. A similar structure was used to perform genome comparison `Cactus` [82].

This type of structure seems promising for future bioinformatics analysis, variant detection and phasing, genome comparison, genomics evolution, and variation analysis [5]. But some trouble still needs to be addressed: how to build this type of graph, and how to map reads against them to construct an efficient coordinate system. Here are some blog post was you can read some blog post about part of this trouble [3] [4] [5]. Another challenge that interests me a lot would be to be able to build a graph genome during assembly.

By using the contigs generated by assembly tools as scaffolds of a genome graph and `KNOT` overlap graph information, I think we can generate directly the genome graph from the reads. If reads come from a single homozygous individual, this genome graph does not contain variant information, in theory. But for a heterozygous individual or a set of divergent cells like cancer cell or a bacterial population, this genome graph representation can help to have a better understanding of the sequenced genome.

Summary of perspectives

In the previous section we have summarized a number of elements of the thesis and detailed several improvements of our work. Here we would like to provide a more synthetic summary of the research perspectives opened by this work.

[3] http://ekg.github.io/2019/07/09/Untangling-graphical-pangenomics
[4] https://lh3.github.io/2019/07/08/on-a-reference-pan-genome-model
[5] https://lh3.github.io/2019/07/12/on-a-reference-pan-genome-model-part-ii

Overlaping consensus: Overlapping search was a hard task, and perform it in reasonable time and memory usage was harder while many overlaps were missed. Combining information of different overlapping tools could be use full to improve downstream analysis.

To create this overlapping consensus tool we need to solve a technical problem: what is the best method to store and request this information. And even more theoretically, first, how to determine if we can merge these overlaps, and second how to assess the confidence we can have in resulting overlaps.

Scrub and correct reads without creating coverage gap: Our work on `KNOT` shows that sometimes the cleaning of reads create coverage gaps in reads. These gaps reduce the contiguity of assembly and reduce our confidence in contigs generated by assembly. At this moment, all trimming, scrubbing and correction tools work like a greedy algorithm, they focus on one read at time.

A read with high error rate and without support from other reads is probably not useful, but sometimes it can solve an assembly trouble. Spending time to find how to change the paradigm of these reads cleaning tools seems useful to me to maximize the usage of the data provided by the sequencing technology.

Find variant at assembly time: We have indicated that the only long read corrector that tried to keep the heterozygosity of the reads during correction was `falcon_sense`. For a de novo assembly, we generally sequence individuals with as smallest heterozygosity as possible or a colony of the same cell, to facilitate our work during assembly.

Consequently, we build assembly tools that do not manage high heterozygosity or sets of cells with variants, like cancer cells and metagenomics datasets. Rewriting a complete assembly tools to manage data with variants seems very hard. The `KNOT` strategy uses classic assembly tools to assemble simple parts of the genome, but going back to original information to find variants and heterozygosity seems a good way to find variant at assembly time.

Bibliography

[1] The schistosoma japonicum genome reveals features of host–parasite interplay. *Nature*, 460 (7253):345–351, jul 2009. doi: 10.1038/nature08140. URL https://doi.org/10.1038%2Fnature08140.

[2] Hind Alhakami, Hamid Mirebrahim, and Stefano Lonardi. A comparative evaluation of genome assembly reconciliation tools. *Genome biology*, 18(1):93, 2017.

[3] S. Altschul. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Research*, 25(17):3389–3402, sep 1997. doi: 10.1093/nar/25.17.3389. URL https://doi.org/10.1093%2Fnar%2F25.17.3389.

[4] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, oct 1990. doi: 10.1016/s0022-2836(05)80360-2. URL https://doi.org/10.1016%2Fs0022-2836%2805%2980360-2.

[5] Adam Ameur. Goodbye reference, hello genome graphs. *Nature Biotechnology*, 37(8): 866–868, aug 2019. doi: 10.1038/s41587-019-0199-7. URL https://doi.org/10.1038%2Fs41587-019-0199-7.

[6] Dmitry Antipov et al. hybridSPAdes: an algorithm for hybrid assembly of short and long reads. *Bioinformatics*, 32(7):1009–1015, nov 2015. doi: 10.1093/bioinformatics/btv688. URL https://doi.org/10.1093/bioinformatics/btv688.

[7] Anton Bankevich et al. SPAdes: A new genome assembly algorithm and its applications to single-cell sequencing. *Journal of Computational Biology*, 19(5):455–477, may 2012. doi: 10.1089/cmb.2012.0021. URL https://doi.org/10.1089/cmb.2012.0021.

[8] Gaëtan Benoit, Dominique Lavenier, Claire Lemaitre, and Guillaume Rizk. Bloocoo, a memory efficient read corrector. European Conference on Computational Biology (ECCB), September 2014. URL https://hal.inria.fr/hal-01092960. Poster.

[9] Konstantin Berlin, Sergey Koren, Chen-Shan Chin, James P Drake, Jane M Landolin, and Adam M Phillippy. Assembling large genomes with single-molecule sequencing and locality-sensitive hashing. *Nature Biotechnology*, 33(6):623–630, may 2015. doi: 10.1038/nbt.3238. URL https://doi.org/10.1038/nbt.3238.

[10] Alex Bishara, Yuling Liu, Ziming Weng, Dorna Kashef-Haghighi, Daniel E. Newburger, Robert West, Arend Sidow, and Serafim Batzoglou. Read clouds uncover variation in complex regions of the human genome. *Genome Research*, 25(10):1570–1580, aug 2015. doi: 10.1101/gr.191189.115. URL https://doi.org/10.1101%2Fgr.191189.115.

[11] Emanuele Bosi et al. MeDuSa: a multi-draft based scaffolder. *Bioinformatics*, 31(15):2443–2451, 2015.

[12] Guy Bresler, Ma'ayan Bresler, and David Tse. Optimal assembly for high throughput shotgun sequencing. *BMC Bioinformatics*, 14(S5), apr 2013. doi: 10.1186/1471-2105-14-s5-s18. URL https://doi.org/10.1186%2F1471-2105-14-s5-s18.

[13] Guy Bresler, Ma'ayan Bresler, and David Tse. Optimal assembly for high throughput shotgun sequencing. *BMC Bioinformatics*, 14, 07 2013.

[14] Christina J. Castro and Terry Fei Fan Ng. U50: A new metric for measuring assembly output based on non-overlapping, target-specific contigs. *Journal of Computational Biology*, 24(11): 1071–1080, nov 2017. doi: 10.1089/cmb.2017.0013. URL https://doi.org/10.1089%2Fcmb.2017.0013.

[15] Mark J Chaisson and Glenn Tesler. Mapping single molecule sequencing reads using basic local alignment with successive refinement (BLASR): application and theory. *BMC Bioinformatics*, 13(1), sep 2012. doi: 10.1186/1471-2105-13-238. URL https://doi.org/10.1186%2F1471-2105-13-238.

[16] Rayan Chikhi and Guillaume Rizk. Space-efficient and exact de bruijn graph representation based on a bloom filter. *Algorithms for Molecular Biology*, 8(1), jan 2013. doi: 10.1186/1748-7188-8-22. URL https://doi.org/10.1186%2F1748-7188-8-22.

[17] Rayan Chikhi, Antoine Limasset, and Paul Medvedev. Compacting de bruijn graphs from sequencing data quickly and in low memory. *Bioinformatics*, 32(12):i201–i208, jun 2016. doi: 10.1093/bioinformatics/btw279. URL https://doi.org/10.1093%2Fbioinformatics%2Fbtw279.

[18] Chen-Shan Chin and Asif Khalak. Human genome assembly in 100 minutes. jul 2019. doi: 10.1101/705616. URL https://doi.org/10.1101%2F705616.

[19] Chen-Shan Chin, David H Alexander, Patrick Marks, Aaron A Klammer, James Drake, Cheryl Heiner, Alicia Clum, Alex Copeland, John Huddleston, Evan E Eichler, Stephen W Turner, and Jonas Korlach. Nonhybrid, finished microbial genome assemblies from long-read SMRT sequencing data. *Nature Methods*, 10(6):563–569, May 2013. doi: 10.1038/nmeth.2474. URL https://doi.org/10.1038/nmeth.2474.

[20] Chen-Shan Chin, Paul Peluso, Fritz J Sedlazeck, Maria Nattestad, Gregory T Concepcion, Alicia Clum, Christopher Dunn, Ronan O'Malley, Rosa Figueroa-Balderas, Abraham Morales-Cruz, Grant R Cramer, Massimo Delledonne, Chongyuan Luo, Joseph R Ecker, Dario Cantu, David R Rank, and Michael C Schatz. Phased diploid genome assembly with single-molecule real-time sequencing. *Nature Methods*, 13(12):1050–1054, October 2016. doi: 10.1038/nmeth.4035. URL https://doi.org/10.1038/nmeth.4035.

[21] Chen-Shan Chin et al. Phased diploid genome assembly with single-molecule real-time sequencing. *Nature Methods*, 13(12):1050–1054, oct 2016. doi: 10.1038/nmeth.4035. URL https://doi.org/10.1038/nmeth.4035.

[22] Justin Chu, Hamid Mohamadi, René L. Warren, Chen Yang, and Inanç Birol. Innovations and challenges in detecting long read overlaps: an evaluation of the state-of-the-art. *Bioinformatics*, page btw811, December 2016. doi: 10.1093/bioinformatics/btw811. URL https://doi.org/10.1093/bioinformatics/btw811.

[23] Scott C. Clark, Rob Egan, Peter I. Frazier, and Zhong Wang. ALE: a generic assembly likelihood evaluation framework for assessing the accuracy of genome and metagenome assemblies. *Bioinformatics*, 29(4):435–443, jan 2013. doi: 10.1093/bioinformatics/bts723. URL https://doi.org/10.1093%2Fbioinformatics%2Fbts723.

[24] Anna Cuscó, Carlotta Catozzi, Joaquim Viñes, Armand Sanchez, and Olga Francino. Microbiota profiling with long amplicons using nanopore sequencing: full-length 16s rRNA gene and whole rrn operon. *F1000Research*, 7:1755, nov 2018. doi: 10.12688/f1000research.16817.1. URL https://doi.org/10.12688%2Ff1000research.16817.1.

[25] Nan Du, Jiao Chen, and Yanni Sun. Improving the sensitivity of long read overlap detection using grouped short k-mer matches. *BMC Genomics*, 20(S2), apr 2019. doi: 10.1186/s12864-019-5475-x. URL https://doi.org/10.1186%2Fs12864-019-5475-x.

[26] Paolo Ferragina and Giovanni Manzini. An experimental study of an opportunistic index. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 269–278. Society for Industrial and Applied Mathematics, 2001.

[27] Adrian Fritz, Peter Hofmann, Stephan Majda, Eik Dahms, Johannes Dröge, Jessika Fiedler, Till R. Lesker, Peter Belmann, Matthew Z. DeMaere, Aaron E. Darling, Alexander Sczyrba, Andreas Bremges, and Alice C. McHardy. CAMISIM: simulating metagenomes and microbial communities. *Microbiome*, 7(1), feb 2019. doi: 10.1186/s40168-019-0633-6. URL https://doi.org/10.1186%2Fs40168-019-0633-6.

[28] Mohammadreza Ghodsi, Christopher M Hill, Irina Astrovskaya, Henry Lin, Dan D Sommer, Sergey Koren, and Mihai Pop. De novo likelihood-based measures for comparing genome assemblies. *BMC research notes*, 6(1):334, 2013.

[29] D. Gordon, J. Huddleston, M. J. P. Chaisson, C. M. Hill, Z. N. Kronenberg, K. M. Munson, M. Malig, A. Raja, I. Fiddes, L. W. Hillier, C. Dunn, C. Baker, J. Armstrong, M. Diekhans, B. Paten, J. Shendure, R. K. Wilson, D. Haussler, C.-S. Chin, and E. E. Eichler. Long-read sequence assembly of the gorilla genome. *Science*, 352(6281):aae0344–aae0344, March 2016. doi: 10.1126/science.aae0344. URL https://doi.org/10.1126/science.aae0344.

[30] Alexey Gurevich, Vladislav Saveliev, Nikolay Vyahhi, and Glenn Tesler. QUAST: quality assessment tool for genome assemblies. *Bioinformatics*, 29(8):1072–1075, February 2013. doi: 10.1093/bioinformatics/btt086. URL https://doi.org/10.1093/bioinformatics/btt086.

[31] Simon A. Hardwick, Anoushka Joglekar, Paul Flicek, Adam Frankish, and Hagen U. Tilgner. Getting the entire message: Progress in isoform sequencing. *Frontiers in Genetics*, 10, aug 2019. doi: 10.3389/fgene.2019.00709. URL https://doi.org/10.3389%2Ffgene.2019.00709.

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

[33] Martin Hunt, Nishadi De Silva, Thomas D. Otto, Julian Parkhill, Jacqueline A. Keane, and Simon R. Harris. Circlator: automated circularization of genome assemblies using long sequencing reads. *Genome Biology*, 16(1), dec 2015. doi: 10.1186/s13059-015-0849-0. URL https://doi.org/10.1186/s13059-015-0849-0.

[34] Shaun D Jackman, Lauren Coombe, Justin Chu, Rene L Warren, Benjamin P Vandervalk, Sarah Yeo, Zhuyi Xue, Hamid Mohamadi, Joerg Bohlmann, Steven JM Jones, et al. Tigmint: Correcting assembly errors using linked reads from large molecules. *bioRxiv*, page 304253, 2018.

[35] Miten Jain, Sergey Koren, Karen H Miga, Josh Quick, Arthur C Rand, Thomas A Sasani, John R Tyson, Andrew D Beggs, Alexander T Dilthey, Ian T Fiddes, Sunir Malla, Hannah Marriott, Tom Nieto, Justin O'Grady, Hugh E Olsen, Brent S Pedersen, Arang Rhie, Hollian Richardson, Aaron R Quinlan, Terrance P Snutch, Louise Tee, Benedict Paten, Adam M Phillippy, Jared T Simpson, Nicholas J Loman, and Matthew Loose. Nanopore sequencing and assembly of a human genome with ultra-long reads. *Nature Biotechnology*, 36(4):338–345, January 2018. doi: 10.1038/nbt.4060. URL https://doi.org/10.1038/nbt.4060.

[36] Govinda M. Kamath, Ilan Shomorony, Fei Xia, Thomas A. Courtade, and David N. Tse. HINGE: long-read assembly achieves optimal repeat resolution. *Genome Research*, 27(5):747–756, mar 2017. doi: 10.1101/gr.216465.116. URL https://doi.org/10.1101%2Fgr.216465.116.

[37] Melissa C. Keinath, Vladimir A. Timoshevskiy, Nataliya Y. Timoshevskaya, Panagiotis A. Tsonis, S. Randal Voss, and Jeramiah J. Smith. Initial characterization of the large genome of the salamander ambystoma mexicanum using shotgun and laser capture chromosome sequencing. *Scientific Reports*, 5(1), nov 2015. doi: 10.1038/srep16413. URL https://doi.org/10.1038%2Fsrep16413.

[38] W. J. Kent. Assembly of the working draft of the human genome with GigAssembler. *Genome Research*, 11(9):1541–1548, aug 2001. doi: 10.1101/gr.183201. URL https://doi.org/10.1101%2Fgr.183201.

[39] Nilesh Khiste and Lucian Ilie. HISEA: HIerarchical SEed aligner for PacBio data. *BMC Bioinformatics*, 18(1), dec 2017. doi: 10.1186/s12859-017-1953-9. URL https://doi.org/10.1186/s12859-017-1953-9.

[40] Daehwan Kim, Li Song, Florian P. Breitwieser, and Steven L. Salzberg. Centrifuge: rapid and sensitive classification of metagenomic sequences. *Genome Research*, 26(12):1721–1729, oct 2016. doi: 10.1101/gr.210641.116. URL https://doi.org/10.1101%2Fgr.210641.116.

[41] Mikhail Kolmogorov, Jeffrey Yuan, Yu Lin, and Pavel A. Pevzner. Assembly of long, error-prone reads using repeat graphs. *Nature Biotechnology*, 37(5):540–546, apr 2019. doi: 10.1038/s41587-019-0072-8. URL https://doi.org/10.1038%2Fs41587-019-0072-8.

[42] Sergey Koren and Adam M Phillippy. One chromosome, one contig: Complete microbial genomes from long-read sequencing and assembly. *Current Opinion in Microbiology*, 23:110–120, 2015. doi: 10.1016/j.mib.2014.11.014. URL https://doi.org/10.1016/j.mib.2014.11.014.

[43] Sergey Koren and Adam M Phillippy. One chromosome, one contig: complete microbial genomes from long-read sequencing and assembly. *Current Opinion in Microbiology*, 23:110–120, February 2015. doi: 10.1016/j.mib.2014.11.014. URL https://doi.org/10.1016/j.mib.2014.11.014.

[44] Sergey Koren, Brian P. Walenz, Konstantin Berlin, Jason R. Miller, Nicholas H. Bergman, and Adam M. Phillippy. Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation. *Genome Research*, 27(5):722–736, mar 2017. doi: 10.1101/gr.215087.116. URL https://doi.org/10.1101/gr.215087.116.

[45] J. Koster and S. Rahmann. Snakemake–a scalable bioinformatics workflow engine. *Bioinformatics*, 28(19):2520–2522, aug 2012. doi: 10.1093/bioinformatics/bts480. URL https://doi.org/10.1093/bioinformatics/bts480.

[46] Frederico Schmitt Kremer et al. Approaches for in silico finishing of microbial genome sequences. *Genetics and molecular biology*, 40(3):553–576, 2017.

[47] Nathan LaPierre, Rob Egan, Wei Wang, and Zhong Wang. MiniScrub: de novo long read scrubbing using approximate alignment and deep learning. *bioRxiv*, oct 2018. doi: 10.1101/433573. URL https://doi.org/10.1101/433573.

[48] Delphine Lariviere, Han Mei, Mallory Freeberg, James Taylor, and Anton Nekrutenko. Understanding trivial challenges of microbial genomics: An assembly example. *bioRxiv*, 2018. doi: 10.1101/347625. URL https://www.biorxiv.org/content/early/2018/06/14/347625.

[49] Bayo Lau et al. LongISLND:in silicosequencing of lengthy and noisy datatypes. *Bioinformatics*, 32(24):3829–3832, sep 2016. doi: 10.1093/bioinformatics/btw602. URL https://doi.org/10.1093/bioinformatics/btw602.

[50] C. Lee, C. Grasso, and M. F. Sharlow. Multiple sequence alignment using partial order graphs. *Bioinformatics*, 18(3):452–464, March 2002. doi: 10.1093/bioinformatics/18.3.452. URL https://doi.org/10.1093/bioinformatics/18.3.452.

[51] Dinghua Li, Chi-Man Liu, Ruibang Luo, Kunihiko Sadakane, and Tak-Wah Lam. MEGAHIT: an ultra-fast single-node solution for large and complex metagenomics assembly via succinct de bruijn graph. *Bioinformatics*, 31(10):1674–1676, jan 2015. doi: 10.1093/bioinformatics/btv033. URL https://doi.org/10.1093%2Fbioinformatics%2Fbtv033.

[52] Heng Li. Aligning sequence reads, clone sequences and assembly contigs with bwa-mem, 2013.

[53] Heng Li. Aligning sequence reads, clone sequences and assembly contigs with bwa-mem, 2013.

[54] Heng Li. `Minimap2` and `Miniasm`: Fast mapping and *de novo* assembly for noisy long sequences. *Bioinformatics*, 32(14):2103–2110, 2016. doi: 10.1093/bioinformatics/btw152. URL `https://doi.org/10.1093/bioinformatics/btw152`.

[55] Heng Li. Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences. *Bioinformatics*, 32(14):2103–2110, March 2016. doi: 10.1093/bioinformatics/btw152. URL `https://doi.org/10.1093/bioinformatics/btw152`.

[56] Heng Li. Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences. *Bioinformatics*, 32(14):2103–2110, mar 2016. doi: 10.1093/bioinformatics/btw152. URL `https://doi.org/10.1093/bioinformatics/btw152`.

[57] Heng Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34(18): 3094–3100, May 2018. doi: 10.1093/bioinformatics/bty191. URL `https://doi.org/10.1093/bioinformatics/bty191`.

[58] Yu Lin, Jeffrey Yuan, Mikhail Kolmogorov, Max W Shen, Mark Chaisson, and Pavel A Pevzner. Assembly of long error-prone reads using de Bruijn graphs. *Proceedings of the National Academy of Sciences*, 113(52):E8396–E8405, 2016.

[59] Nicholas J Loman, Joshua Quick, and Jared T Simpson. A complete bacterial genome assembled de novo using only nanopore sequencing data. *Nature Methods*, 12(8):733–735, jun 2015. doi: 10.1038/nmeth.3444. URL `https://doi.org/10.1038/nmeth.3444`.

[60] Fang Luo, Mingbo Yin, Xiaojin Mo, Chengsong Sun, Qunfeng Wu, Bingkuan Zhu, Manyu Xiang, Jipeng Wang, Yi Wang, Jian Li, Ting Zhang, Bin Xu, Huajun Zheng, Zheng Feng, and Wei Hu. An improved genome assembly of the fluke schistosoma japonicum. *PLOS Neglected Tropical Diseases*, 13(8):e0007612, aug 2019. doi: 10.1371/journal.pntd.0007612. URL `https://doi.org/10.1371%2Fjournal.pntd.0007612`.

[61] Nicola De Maio, Liam P. Shaw, Alasdair Hubbard, Sophie George, Nick Sanderson, Jeremy Swann, Ryan Wick, Manal AbuOun, Emma Stubberfield, Sarah J. Hoosdally, Derrick W. Crook, Timothy E. A. Peto, Anna E. Sheppard, Mark J. Bailey, Daniel S. Read, Muna F. Anjum, A. Sarah Walker, and Nicole Stoesser and. Comparison of long-read sequencing technologies in the hybrid assembly of complex bacterial genomes. *bioRxiv*, jan 2019. doi: 10.1101/530824. URL `https://doi.org/10.1101/530824`.

[62] Pierre Marijon, Rayan Chikhi, and Jean-Stéphane Varré. Graph analysis of fragmented long-read bacterial genome assemblies. *Bioinformatics*, mar 2019. doi: 10.1093/bioinformatics/btz219. URL `https://doi.org/10.1093%2Fbioinformatics%2Fbtz219`.

[63] Pierre Marijon, Rayan Chikhi, and Jean-Stéphane Varré. yacrd and fpa: upstream tools for long-read genome assembly. *bioRxiv*, jun 2019. doi: 10.1101/674036. URL `https://doi.org/10.1101%2F674036`.

[64] Marcel Martin, Murray Patterson, Shilpa Garg, Sarah O Fischer, Nadia Pisanti, Gunnar W Klau, Alexander Schöenhuth, and Tobias Marschall. WhatsHap: fast and accurate read-based phasing. nov 2016. doi: 10.1101/085050. URL `https://doi.org/10.1101%2F085050`.

[65] Karen H. Miga, Sergey Koren, Arang Rhie, Mitchell R. Vollger, Ariel Gershman, Andrey Bzikadze, Shelise Brooks, Edmund Howe, David Porubsky, Glennis A. Logsdon, Valerie A. Schneider, Tamara Potapova, Jonathan Wood, William Chow, Joel Armstrong, Jeanne Fredrickson, Evgenia Pak, Kristof Tigyi, Milinn Kremitzki, Christopher Markovic, Valerie Maduro, Amalia Dutra, Gerard G. Bouffard, Alexander M. Chang, Nancy F. Hansen, Françoisen Thibaud-Nissen, Anthony D. Schmitt, Jon-Matthew Belton, Siddarth Selvaraj, Megan Y. Dennis, Daniela C. Soto, Ruta Sahasrabudhe, Gulhan Kaya, Josh Quick, Nicholas J. Loman, Nadine Holmes, Matthew Loose, Urvashi Surti, Rosa ana Risques, Tina A. Graves Lindsay, Robert Fulton, Ira Hall, Benedict Paten, Kerstin Howe, Winston Timp, Alice Young, James C. Mullikin, Pavel A. Pevzner, Jennifer L. Gerton, Beth A. Sullivan, Evan E. Eichler, and Adam M. Phillippy. Telomere-to-telomere assembly of a complete human x chromosome. *bioRxiv*, aug 2019. doi: 10.1101/735928. URL `https://doi.org/10.1101%2F735928`.

[66] Jason R. Miller, Arthur L. Delcher, Sergey Koren, Eli Venter, Brian P. Walenz, Anushka Brownley, Justin Johnson, Kelvin Li, Clark Mobarry, and Granger Sutton. Aggressive assembly of pyrosequencing reads with mates. *Bioinformatics*, 24(24):2818–2824, October 2008. doi: 10.1093/bioinformatics/btn548. URL `https://doi.org/10.1093/bioinformatics/btn548`.

[67] Jason R. Miller et al. Aggressive assembly of pyrosequencing reads with mates. *Bioinformatics*, 24(24):2818–2824, oct 2008. doi: 10.1093/bioinformatics/btn548. URL `https://doi.org/10.1093/bioinformatics/btn548`.

[68] Pierre Morisse, Camille Marchet, Antoine Limasset, Thierry Lecroq, and Arnaud Lefebvre. CONSENT: Scalable self-correction of long reads with multiple sequence alignment. *bioRxiv*, feb 2019. doi: 10.1101/546630. URL `https://doi.org/10.1101/546630`.

[69] E. W. Myers. A whole-genome assembly of drosophila. *Science*, 287(5461):2196–2204, March 2000. doi: 10.1126/science.287.5461.2196. URL `https://doi.org/10.1126/science.287.5461.2196`.

[70] E. W. Myers. The fragment assembly string graph. *Bioinformatics*, 21(Suppl 2):ii79–ii85, September 2005. doi: 10.1093/bioinformatics/bti1114. URL `https://doi.org/10.1093/bioinformatics/bti1114`.

[71] EUGENE W. MYERS. Toward simplifying and accurately formulating fragment assembly. *Journal of Computational Biology*, 2(2):275–290, jan 1995. doi: 10.1089/cmb.1995.2.275. URL `https://doi.org/10.1089%2Fcmb.1995.2.275`.

[72] Gene Myers. The fragment assembly string graph. *Bioinformatics*, 21(suppl_2):ii79–ii85, 2005.

[73] Gene Myers. Daligner: Fast and sensitive detection of all pairwise local alignments. `https://dazzlerblog.wordpress.com/2014/07/10/dalign-fast-and-sensitive-detection-of-all-pairwise-local-alignments/`, 2014.

[74] Gene Myers. Intrinsic quality values. `https://dazzlerblog.wordpress.com/2015/11/06/intrinsic-quality-values/`, 2015.

[75] Gene Myers. Scrubbing reads for better assembly. `https://dazzlerblog.wordpress.com/2017/04/22/1344/`, 2017.

[76] Niranjan Nagarajan and Mihai Pop. Sequence assembly demystified. *Nature Reviews Genetics*, 14(3):157–167, January 2013. doi: 10.1038/nrg3367. URL https://doi.org/10.1038/nrg3367.

[77] Sergej Nowoshilow, Siegfried Schloissnig, Ji-Feng Fei, Andreas Dahl, Andy W. C. Pang, Martin Pippel, Sylke Winkler, Alex R. Hastie, George Young, Juliana G. Roscito, Francisco Falcon, Dunja Knapp, Sean Powell, Alfredo Cruz, Han Cao, Bianca Habermann, Michael Hiller, Elly M. Tanaka, and Eugene W. Myers. The axolotl genome and the evolution of key tissue formation regulators. *Nature*, 554(7690):50–55, jan 2018. doi: 10.1038/nature25458. URL https://doi.org/10.1038%2Fnature25458.

[78] Nathan D. Olson et al. Metagenomic assembly through the lens of validation: recent advances in assessing and improving the quality of genomes assembled from metagenomes. *Briefings in Bioinformatics*, aug 2017. doi: 10.1093/bib/bbx098. URL https://doi.org/10.1093/bib/bbx098.

[79] Nathan D. Olson et al. Metagenomic assembly through the lens of validation: recent advances in assessing and improving the quality of genomes assembled from metagenomes. *Briefings in Bioinformatics*, aug 2017. doi: 10.1093/bib/bbx098. URL https://doi.org/10.1093/bib/bbx098.

[80] Brian D. Ondov, Todd J. Treangen, Páll Melsted, Adam B. Mallonee, Nicholas H. Bergman, Sergey Koren, and Adam M. Phillippy. Mash: fast genome and metagenome distance estimation using MinHash. *Genome Biology*, 17(1), June 2016. doi: 10.1186/s13059-016-0997-x. URL https://doi.org/10.1186/s13059-016-0997-x.

[81] Donovan H Parks et al. CheckM: assessing the quality of microbial genomes recovered from isolates, single cells, and metagenomes. *Genome research*, pages gr–186072, 2015.

[82] Benedict Paten, Mark Diekhans, Dent Earl, John St. John, Jian Ma, Bernard Suh, and David Haussler. Cactus graphs for genome comparisons. *Journal of Computational Biology*, 18(3):469–481, mar 2011. doi: 10.1089/cmb.2010.0252. URL https://doi.org/10.1089%2Fcmb.2010.0252.

[83] P. A. Pevzner, H. Tang, and M. S. Waterman. An eulerian path approach to DNA fragment assembly. *Proceedings of the National Academy of Sciences*, 98(17):9748–9753, August 2001. doi: 10.1073/pnas.171285098. URL https://doi.org/10.1073/pnas.171285098.

[84] Adam M Phillippy, Michael C Schatz, and Mihai Pop. Genome assembly forensics: finding the elusive mis-assembly. *Genome Biology*, 9(3):R55, 2008. doi: 10.1186/gb-2008-9-3-r55. URL https://doi.org/10.1186/gb-2008-9-3-r55.

[85] Atif Rahman and Lior Pachter. CGAL: computing genome assembly likelihoods. *Genome biology*, 14(1):R8, 2013.

[86] Franka J. Rang, Wigard P. Kloosterman, and Jeroen de Ridder. From squiggle to basepair: computational approaches for improving nanopore sequencing read accuracy. *Genome Biology*, 19(1), July 2018. doi: 10.1186/s13059-018-1462-9. URL https://doi.org/10.1186/s13059-018-1462-9.

[87] Jue Ruan and Heng Li. Fast and accurate long-read assembly with wtdbg2. *bioRxiv*, January 2019. doi: 10.1101/530972. URL `https://doi.org/10.1101/530972`.

[88] K. Sahlin, N. Street, J. Lundeberg, and L. Arvestad. Improved gap size estimation for scaffolding algorithms. *Bioinformatics*, 28(17):2215–2222, Sep 2012. doi: 10.1093/bioinformatics/bts441.

[89] K. Sahlin, F. Vezzi, B. Nystedt, J. Lundeberg, and L. Arvestad. BESST–efficient scaffolding of large fragmented assemblies. *BMC Bioinformatics*, 15:281, 2014. doi: 10.1186/1471-2105-15-281.

[90] Kristoffer Sahlin, Rayan Chikhi, and Lars Arvestad. Assembly scaffolding with pe-contaminated mate-pair libraries. *Bioinformatics*, 2016. doi: 10.1093/bioinformatics/btw064. URL `http://bioinformatics.oxfordjournals.org/content/early/2016/03/09/bioinformatics.btw064.abstract`.

[91] F. Sanger, G. M. Air, B. G. Barrell, N. L. Brown, A. R. Coulson, C. A. Fiddes, C. A. Hutchison, P. M. Slocombe, and M. Smith. Nucleotide sequence of bacteriophage phi X174 DNA. *Nature*, 265(5596):687–695, Feb 1977.

[92] Aylwyn Scally, Julien Y. Dutheil, LaDeana W. Hillier, Gregory E. Jordan, Ian Goodhead, Javier Herrero, Asger Hobolth, Tuuli Lappalainen, Thomas Mailund, Tomas Marques-Bonet, Shane McCarthy, Stephen H. Montgomery, Petra C. Schwalie, Y. Amy Tang, Michelle C. Ward, Yali Xue, Bryndis Yngvadottir, Can Alkan, Lars N. Andersen, Qasim Ayub, Edward V. Ball, Kathryn Beal, Brenda J. Bradley, Yuan Chen, Chris M. Clee, Stephen Fitzgerald, Tina A. Graves, Yong Gu, Paul Heath, Andreas Heger, Emre Karakoc, Anja Kolb-Kokocinski, Gavin K. Laird, Gerton Lunter, Stephen Meader, Matthew Mort, James C. Mullikin, Kasper Munch, Timothy D. O'Connor, Andrew D. Phillips, Javier Prado-Martinez, Anthony S. Rogers, Saba Sajjadian, Dominic Schmidt, Katy Shaw, Jared T. Simpson, Peter D. Stenson, Daniel J. Turner, Linda Vigilant, Albert J. Vilella, Weldon Whitener, Baoli Zhu, David N. Cooper, Pieter de Jong, Emmanouil T. Dermitzakis, Evan E. Eichler, Paul Flicek, Nick Goldman, Nicholas I. Mundy, Zemin Ning, Duncan T. Odom, Chris P. Ponting, Michael A. Quail, Oliver A. Ryder, Stephen M. Searle, Wesley C. Warren, Richard K. Wilson, Mikkel H. Schierup, Jane Rogers, Chris Tyler-Smith, and Richard Durbin. Insights into hominid evolution from the gorilla genome sequence. *Nature*, 483(7388):169–175, mar 2012. doi: 10.1038/nature10842. URL `https://doi.org/10.1038/nature10842`.

[93] Fritz J. Sedlazeck, Hayan Lee, Charlotte A. Darby, and Michael C. Schatz. Piercing the dark matter: bioinformatics of long-range sequencing and mapping. *Nature Reviews Genetics*, 19 (6):329–346, March 2018. doi: 10.1038/s41576-018-0003-4. URL `https://doi.org/10.1038/s41576-018-0003-4`.

[94] Kishwar Shafin, Trevor Pesout, Ryan Lorig-Roach, Marina Haukness, Hugh E. Olsen, Colleen Bosworth, Joel Armstrong, Kristof Tigyi, Nicholas Maurer, Sergey Koren, Fritz J. Sedlazeck, Tobias Marschall, Simon Mayes, Vania Costa, Justin M. Zook, Kelvin J. Liu, Duncan Kilburn, Melanie Sorensen, Katy M. Munson, Mitchell R. Vollger, Evan E. Eichler, Sofie Salama, David Haussler, Richard E. Green, Mark Akeson, Adam Phillippy, Karen H. Miga, Paolo Carnevali, Miten Jain, and Benedict Paten. Efficient de novo assembly of eleven human genomes using

PromethION sequencing and a novel nanopore toolkit. jul 2019. doi: 10.1101/715722. URL https://doi.org/10.1101%2F715722.

[95] A. Shajii, I. Numanagi?, and B. Berger. Latent Variable Model for Aligning Barcoded Short-Reads Improves Downstream Analyses. *Res Comput Mol Biol*, 10812:280–282, Apr 2018.

[96] Ilan Shomorony, Thomas A. Courtade, and David Tse. Fundamental limits of genome assembly under an adversarial erasure model. *IEEE Transactions on Molecular, Biological and Multi-Scale Communications*, 2(2):199–208, dec 2016. doi: 10.1109/tmbmc.2016.2641440. URL https://doi.org/10.1109%2Ftmbmc.2016.2641440.

[97] Felipe A. Simão, Robert M. Waterhouse, Panagiotis Ioannidis, Evgenia V. Kriventseva, and Evgeny M. Zdobnov. BUSCO: assessing genome assembly and annotation completeness with single-copy orthologs. *Bioinformatics*, 31(19):3210–3212, jun 2015. doi: 10.1093/bioinformatics/btv351. URL https://doi.org/10.1093/bioinformatics/btv351.

[98] J. T. Simpson and R. Durbin. Efficient de novo assembly of large genomes using compressed data structures. *Genome Research*, 22(3):549–556, dec 2011. doi: 10.1101/gr.126953.111. URL https://doi.org/10.1101%2Fgr.126953.111.

[99] Jeramiah J. Smith, Nataliya Timoshevskaya, Vladimir A. Timoshevskiy, Melissa C. Keinath, Drew Hardy, and S. Randal Voss. A chromosome-scale assembly of the axolotl genome. *Genome Research*, 29(2):317–324, jan 2019. doi: 10.1101/gr.241901.118. URL https://doi.org/10.1101%2Fgr.241901.118.

[100] Ivan Sović, Mile Šikić, Andreas Wilm, Shannon Nicole Fenlon, Swaine Chen, and Niranjan Nagarajan. Fast and sensitive mapping of nanopore sequencing reads with GraphMap. *Nature Communications*, 7:11307, apr 2016. doi: 10.1038/ncomms11307. URL https://doi.org/10.1038/ncomms11307.

[101] Todd J. Treangen, Anne-Laure Abraham, Marie Touchon, and Eduardo P.C. Rocha. Genesis, effects and fates of repeats in prokaryotic genomes. *FEMS Microbiology Reviews*, 33(3):539–571, may 2009. doi: 10.1111/j.1574-6976.2009.00169.x. URL https://doi.org/10.1111/j.1574-6976.2009.00169.x.

[102] Sagar M Utturkar et al. A case study into microbial genome assembly gap sequences and finishing strategies. *Frontiers in microbiology*, 8:1272, 2017.

[103] Robert Vaser and Mile Šikić. Yet another de novo genome assembler. *bioRxiv*, may 2019. doi: 10.1101/656306. URL https://doi.org/10.1101%2F656306.

[104] Robert Vaser, Ivan Sović, Niranjan Nagarajan, and Mile Šikić. Fast and accurate de novo genome assembly from long uncorrected reads. *Genome Research*, 27(5):737–746, jan 2017. doi: 10.1101/gr.214270.116. URL https://doi.org/10.1101/gr.214270.116.

[105] Francesco Vezzi, Giuseppe Narzisi, and Bud Mishra. Reevaluating assembly evaluations with feature response curves: GAGE and assemblathons. *PloS one*, 7(12):e52210, 2012.

[106] Bruce J Walker et al. Pilon: an integrated tool for comprehensive microbial variant detection and genome assembly improvement. *PloS one*, 9(11):e112963, 2014.

[107] Jeremy R. Wang, James Holt, Leonard McMillan, and Corbin D. Jones. FMLRC: Hybrid long read error correction using an FM-index. *BMC Bioinformatics*, 19(1), feb 2018. doi: 10.1186/s12859-018-2051-3. URL https://doi.org/10.1186%2Fs12859-018-2051-3.

[108] Ryan Wick. Porechop: adapter trimmer for oxford nanopore reads, jan 2018. URL https://github.com/rrwick/Porechop.

[109] Ryan Wick and Kathryn E. Holt. rrwick/Long-read-assembler-comparison: Initial release, May 2019. URL https://doi.org/10.5281/zenodo.2702443.

[110] Ryan R. Wick, Mark B. Schultz, Justin Zobel, and Kathryn E. Holt. Bandage: interactive visualization ofde novogenome assemblies: Fig. 1. *Bioinformatics*, 31(20):3350–3352, jun 2015. doi: 10.1093/bioinformatics/btv383. URL https://doi.org/10.1093%2Fbioinformatics%2Fbtv383.

[111] Ryan R. Wick et al. Unicycler: Resolving bacterial genome assemblies from short and long sequencing reads. *PLOS Computational Biology*, 13(6):e1005595, jun 2017. doi: 10.1371/journal.pcbi.1005595. URL https://doi.org/10.1371/journal.pcbi.1005595.

[112] Chuan-Le Xiao, Ying Chen, Shang-Qian Xie, Kai-Ning Chen, Yan Wang, Yue Han, Feng Luo, and Zhi Xie. MECAT: fast mapping, error correction, and de novo assembly for single-molecule sequencing reads. *Nature Methods*, 14(11):1072–1074, sep 2017. doi: 10.1038/nmeth.4432. URL https://doi.org/10.1038/nmeth.4432.

[113] Gokhan Yavas, Huixiao Hong, and Wenming Xiao. dnAQET: a framework to compute a consolidated metric for benchmarking quality of de novo assemblies. *BMC Genomics*, 20(1), sep 2019. doi: 10.1186/s12864-019-6070-x. URL https://doi.org/10.1186%2Fs12864-019-6070-x.

[114] Chengxi Ye, Christopher M. Hill, Shigang Wu, Jue Ruan, and Zhanshan Ma. DBG2olc: Efficient assembly of large genomes using long erroneous reads of the third generation sequencing technologies. *Scientific Reports*, 6(1), aug 2016. doi: 10.1038/srep31900. URL https://doi.org/10.1038/srep31900.

[115] Aleksey V Zimin, Guillaume Marçais, Daniela Puiu, Michael Roberts, Steven L Salzberg, and James A Yorke. The MaSuRCA genome assembler. *Bioinformatics*, 29(21):2669–2677, 2013.

Appendix A

KNOT

A.1 Contig classification

In order to have a better understanding of the contig graph produced by a given assembler, we wish to filter out contigs that are not of chromosomal origin. We compare each contig against the *nr* database using Megablast (Morgulis et al., 2008), and classify a contig as chromosomal if its length is greater than 1 Mb, or is such that 80% of the first 50 Megablast hits map to a complete bacterial genome. We use the same second criterion to classify whether a contig is of plasmid origin, regardless of its size. Remaining unclassified contigs are classified as of undefined origin. In addition, we flag as *containment* contigs those which map (using `Minimap2`) over at least 75% of their length to another contig.

A.2 On whether `Canu` contig fragmentation can be solved using `Miniasm` contigs

To check if `Miniasm` contigs could possibly enable to order and fill gaps between `Canu` contigs, we performed an assembly using the `Minimap2` and `Miniasm` pipeline using both the `Canu` contigs and the `Miniasm` contigs as input (to be clear: no reads were used as input to this assembly, only two contig sets). To allow `Minimap2` to find shorter matches, mapping of `Miniasm` contigs against `Canu` contigs was performed with the following parameters: `-x map-pb -m 25 -n 2`. To avoid `Miniasm` filtering overlaps, we ran it with the following parameters: `-1 -2 -s 1000 -c 0`.

We ran this pipeline on all datasets, and counted the number of times that a `Miniasm` contig overlaps with two `Canu` contigs. We also counted the number of contigs generated by `Miniasm` using the overlap created at the previous step. Results are summarized in Supplementary Table A.1.

NCTC ID	number of genomic contig Canu	Miniasm	number of Miniasm contigs that overlap two Canu contigs	number of merged contigs contigs from Miniasm/Canu overlaps
NCTC10006	3	7	0	0
NCTC10332	12	22	1	0
NCTC10444	7	5	0	0
NCTC10702	3	2	0	0
NCTC10766	13	7	1	0
NCTC10794	7	5	0	0
NCTC10988	10	9	0	0
NCTC11126	7	15	5	0
NCTC11343	12	10	2	1
NCTC11360	26	25	1	0
NCTC11435	8	6	2	0
NCTC11800	7	3	1	0
NCTC11872	7	13	3	0
NCTC12123	5	3	2	0
NCTC12126	13	15	4	0
NCTC12131	16	77	3	0
NCTC12132	2	4	1	0
NCTC12146	3	1	0	0
NCTC12694	21	123	1	0
NCTC12841	16	1	0	0
NCTC12993	5	2	0	0
NCTC12998	3	4	0	0
NCTC13095	3	2	0	0
NCTC13125	6	7	0	0
NCTC13348	25	17	3	1
NCTC13463	5	4	1	0
NCTC13543	3	3	0	0
NCTC4672	68	16	5	1
NCTC5050	4	4	0	0
NCTC5053	8	11	2	1
NCTC5055	143	20	0	0
NCTC7922	13	9	4	1
NCTC8179	15	15	1	0
NCTC8500	3	1	0	1
NCTC8684	5	2	0	0
NCTC9075	7	3	2	0
NCTC9078	4	2	0	0
NCTC9098	8	6	3	0
NCTC9111	9	13	0	0
NCTC9112	7	15	10	0
NCTC9184	141	17	0	0
NCTC9645	31	76	9	3
NCTC9646	8	9	3	1
NCTC9695	2	1	0	0

Table A.1: The pipeline described section A.2 found more than one overlap between Canu contigs with Miniasm contig for 24 over 45 datasets. When these overlaps are re-assembled using Miniasm, one or more merged contigs are produced in only 8 out of 45 datasets.

A.3 Assembly summary

Tables A.2 and A.3 report our complete results for the 45 NCTC datasets.

NCTC ID	species	cov	NCTC contigs chr	pld	und	HINGE status	Canu contigs chr	pld	und	Miniasm contigs chr	pld	und
NCTC10006	*E. aerogenes*	56	1	0	0	MAF	3	0	0	7	0	2
NCTC10332	*P. aeruginosa*	36	1	0	0	MAF	12	0	0	22	0	1
NCTC10444	*E. coli*	61	1	0	0	MAF	7	0	0	5	0	1
NCTC10702	*S. aureus*	24	1	1	1	MAF	3	3	0	2	1	2
NCTC10766	*E. alkalescens*	37	0	0	11	MA	13	7	3	7	2	5
NCTC10794	*H. parahaemolyticus*	26	0	0	3	MAF	7	0	2	5	0	2
NCTC10988	*S. aureus*	87	0	0	13	MA	10	0	26	9	0	4
NCTC11126	*E. coli*	50	2	0	0	FALC	7	0	2	15	0	18
NCTC11343	*S. multivorum*	22	0	0	11	MAF	12	0	0	10	0	0
NCTC11360	*S. agalactiae*	3	0	0	3	MAF	26	0	17	25	0	1
NCTC11435	*V. mimicus*	60	0	0	3	MA*	8	0	0	6	0	2
NCTC11800	*P. stuartii*	32	0	0	4	MA	7	0	0	3	0	0
NCTC11872	*H. influenzae*	27	0	0	11	MAF	7	0	3	13	0	1
NCTC12123	*E. asburiae*	64	2	3	0	FAMT	5	4	1	3	1	1
NCTC12126	*E.rcancerogenus*	42	6	1	0	MAF	13	1	4	15	0	10
NCTC12131	*Y. regensburgei*	41	3	0	0	MAF	16	0	0	77	0	0
NCTC12132	*M. wisconsensis*	86	1	0	0	MAF	2	0	2	4	0	0
NCTC12146	*Klebsiella terrigena*	11	0	0	2	MAF	3	0	1	1	0	2
NCTC12694	*S. enterica*	19	0	0	121	MAF	21	3	0	123	2	0
NCTC12841	*S. pyogenes*	75	*	*	*	MA	16	0	0	1	0	2
NCTC12993	*K. cryocrescens*	46	5	1	0	FCA	5	4	0	2	3	0
NCTC12998	*R. planticola*	41	1	1	0	MAF	3	2	4	4	1	2
NCTC13095	*K. planticola*	38	1	1	3	FCA	3	3	0	2	0	1
NCTC13125	*E. coli*	49	1	2	4	MAF	6	3	1	7	2	1
NCTC13348	*S. enterica*	41	4	2	0	MAF	25	1	1	17	1	2
NCTC13463	*E. coli*	62	1	1	4	MAF	5	2	2	4	1	3
NCTC13543	*R. radiobacter*	31	0	0	12	MA	3	2	2	3	3	0
NCTC4672	*S. uberis*	10	0	0	3	MAF	68	0	8	16	0	1
NCTC5050	*K. pneumoniae*	54	2	3	0	MAF	4	2	3	4	3	2
NCTC5053	*K. pneumoniae*	28	0	0	7	MAF	8	5	1	11	5	1
NCTC5055	*K. pneumoniae*	69	1	0	2	MAF	143	8	3	20	3	1
NCTC7152	*E. coli*	49	1	0	4	MAF	2	3	5	1	1	3
NCTC7922	*E. coli*	26	0	0	6	MAF	13	3	4	9	2	1
NCTC8179	*E. coli*	36	1	1	3	MAF	15	4	4	15	2	0
NCTC8500	*E. coli*	29	1	1	0	MAF	3	1	1	1	1	5
NCTC8684	*C. violaceum*	36	0	0	3	MAF	5	0	0	2	0	1
NCTC9075	*E. coli*	35	1	0	3	MAF	7	0	14	3	0	1
NCTC9078	*E. coli*	55	1	2	2	MA	4	3	1	2	1	2
NCTC9098	*E. coli*	56	1	1	2	MAF	8	0	1	6	2	2
NCTC9111	*E. coli*	62	1	1	9	MAF	9	6	2	13	3	1
NCTC9112	*E. coli*	69	1	0	0	MAF	7	0	5	15	0	1
NCTC9184	*Klebsiella sp.*	6	0	0	179	MAF	141	5	0	17	0	4
NCTC9645	*K. pneumoniae*	17	0	0	16	MAF	31	10	1	76	4	2
NCTC9646	*K. aerogenes*	24	*	*	*	MAF	8	3	3	9	5	1
NCTC9695	*C. violaceum*	34	1	0	0	MAF	2	9	3	1	0	1
	T. roseus	20	*	*	*	*	3	0	0	6	0	0

Table A.2: Datasets from the NCTC project chosen for analysis (the last row corresponds to our simulated dataset). For each sample, the coverage (cov) is given as well as the number of contigs and their assignment; chr: number of chromosomal contigs, pld: number of plasmid contigs, und: number of other contigs. For two datasets (NCTC12841 and NCTC9646) the NCTC project does not yet provide an assembly ("Pending"). For `Canu` and `Miniasm`, a classification similar to the one of NCTC is given (see text). We reported `HINGE` classification; FALC: Finished assembly (lacking circularization), FA: Finished assembly, MA: Mis-assembly, MA*: labeled as misassembled but actually correctly solved as 2 chromosomes, FCA: Finished circular assembly, MAF: Mis-assembly/Fragmented, FAMT: Finished assembly with multiple traversals.

NCTC ID	Canu			Edges in the AAG					
	contigs	dead-ends	dead-ends with adj. edge	total AAG	theoretical max. edges	distant edges	adjacency edges total	adjacency edges single	adjacency edges multiple
NCTC10006	2	2	2	4	4	2	2	2	0
NCTC10332	4	8	4	24	24	21	3	0	3
NCTC10444	4	3	3	24	24	18	6	0	6
NCTC10702	2	4	0	4	4	4	0	0	0
NCTC10766	4	6	2	24	24	22	2	2	0
NCTC10794	3	5	0	12	12	12	0	0	0
NCTC10988	1	0	0	0	0	0	0	0	0
NCTC11126	4	4	3	20	24	15	5	0	5
NCTC11343	7	6	3	72	84	66	6	1	5
NCTC11360	3	6	0	12	12	12	0	0	0
NCTC11435	5	4	4	40	40	35	5	2	3
NCTC11800	2	0	0	3	4	1	2	2	0
NCTC11872	5	6	4	40	40	36	4	4	0
NCTC12123	3	4	3	12	12	9	3	1	2
NCTC12126	6	7	6	36	60	26	10	0	10
NCTC12131	8	6	6	83	112	60	23	0	23
NCTC12132	2	4	2	4	4	3	1	1	0
NCTC12146	2	4	0	4	4	4	0	0	0
NCTC12694	10	20	6	61	180	58	3	3	0
NCTC12841	1	0	0	0	0	0	0	0	0
NCTC12993	2	4	2	4	4	3	1	1	0
NCTC12998	1	2	0	0	0	0	0	0	0
NCTC13095	2	2	0	4	4	3	1	1	0
NCTC13125	3	0	0	12	12	8	4	0	4
NCTC13348	7	7	0	75	84	68	7	0	7
NCTC13463	2	0	0	3	4	1	2	2	0
NCTC13543	2	2	0	4	4	4	0	0	0
NCTC4672	5	8	4	32	40	28	4	0	4
NCTC5050	3	6	6	12	12	9	3	3	0
NCTC5053	5	6	2	32	40	28	4	1	3
NCTC5055	1	2	0	0	0	0	0	0	0
NCTC7152	1	0	0	0	0	0	0	0	0
NCTC7922	6	3	2	60	60	56	4	2	2
NCTC8179	7	4	0	84	84	79	5	3	2
NCTC8500	1	2	0	0	0	0	0	0	0
NCTC8684	1	2	0	0	0	0	0	0	0
NCTC9075	6	8	7	60	60	54	6	4	2
NCTC9078	2	0	0	2	4	1	1	1	0
NCTC9098	4	1	1	24	24	16	8	0	8
NCTC9111	3	2	2	12	12	8	4	0	4
NCTC9112	4	0	0	24	24	14	10	0	10
NCTC9184	0	0	0	0	0	0	0	0	0
NCTC9645	14	23	5	244	364	238	6	3	3
NCTC9646	5	8	4	40	40	37	3	3	0
NCTC9695	2	0	0	2	4	1	1	1	0
Summary	3.71	4.24	1.84	26.86	34.4	23.55	3.31	0.95	2.35

Table A.3: Assembly graph statistics for a selection of 45 fragmented assemblies from the NCTC project. `Canu` assembly graph statistics: number of contigs, number of dead-end extremities. AAG statistics: theoretical maximal number number of edges. Note that for some of the most fragmented datasets (e.g. NCTC9184), none of the contigs pass the 100 Kbp length threshold, hence the AAG is empty.

Mean number of	
`Miniasm` contigs	5.8
Edges in AAG	85.1
Theoretical max. edges in AAG	94.4
Distant edges	83.12
All adjacency edges	1.98
Single adjacency edges	1.51
Multiple adjacency edges	0.46
Dead-ends in `Miniasm` contigs	11.61
Dead-ends in AAG, adjacency edges	7.95

Table A.4: Average statistics of augmented assembly graphs using a SG built from `Minimap2` overlaps on `Miniasm` contigs across the 37 NCTC datasets with two or more contigs, after size and classification filters. All rows are as per definitions in Section 4.4.4.4. 'Theoretical max. edges': number of possible edges in each AAG. 'Dead-ends in AAG, adjacency edges': number of dead-ends in the AAG when only adjacency edges are considered, i.e. distant edges are deleted.

A.4

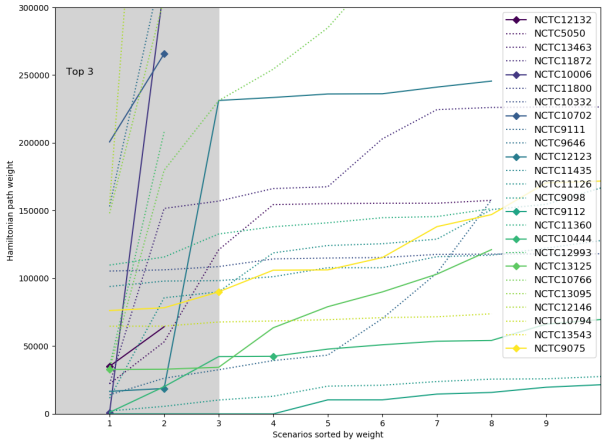


Figure A.1: Weights of scenarios in AAGs. Each curve correspond to the sorted list of Hamiltonian cycles, sorted by weight. If a ground truth is known, a diamond symbol marks the correct assembly scenario

NCTC ID	Miniasm		Edges in the AAG					
	contigs	dead-ends	dead-ends with adj. edge	total AAG	theoretical max. edges	distant edges	adjacency edges total	single	multiple
NCTC10006	7	14	11	84	84	78	6	4	2
NCTC10332	14	28	12	364	364	358	6	6	0
NCTC10444	5	10	2	40	40	39	1	1	0
NCTC10702	1	2	0	0	0	0	0	0	0
NCTC10766	5	10	8	40	40	35	5	2	3
NCTC10794	3	6	0	12	12	12	0	0	0
NCTC10988	3	6	2	12	12	11	1	1	0
NCTC11126	11	22	8	200	220	196	4	4	0
NCTC11343	3	6	0	8	12	8	0	0	0
NCTC11360	7	14	2	84	84	83	1	1	0
NCTC11435	6	12	4	33	60	31	2	2	0
NCTC11800	3	6	2	8	12	7	1	1	0
NCTC11872	9	18	10	113	144	108	5	5	0
NCTC12123	2	4	4	4	4	1	3	0	3
NCTC12126	11	22	10	146	220	141	5	5	0
NCTC12131	17	34	2	512	544	511	1	1	0
NCTC12132	2	4	0	4	4	4	0	0	0
NCTC12146	1	2	0	0	0	0	0	0	0
NCTC12694	0	0	0	0	0	0	0	0	0
NCTC12841	1	2	0	0	0	0	0	0	0
NCTC12993	2	4	2	4	4	3	1	1	0
NCTC12998	4	8	2	24	24	23	1	1	0
NCTC13095	1	2	0	0	0	0	0	0	0
NCTC13125	5	10	4	32	40	30	2	2	0
NCTC13348	15	30	19	420	420	410	10	8	2
NCTC13463	4	8	2	18	24	17	1	1	0
NCTC13543	2	4	0	4	4	4	0	0	0
NCTC4672	8	16	*	*	*	*	*	*	*
NCTC5050	4	8	5	24	24	20	4	0	4
NCTC5053	9	18	8	113	144	108	5	2	3
NCTC5055	1	2	0	0	0	0	0	0	0
NCTC7152	2	4	0	4	4	4	0	0	0
NCTC7922	6	12	2	50	60	49	1	1	0
NCTC8179	11	22	4	220	220	218	2	2	0
NCTC8500	1	2	0	0	0	0	0	0	0
NCTC8684	2	4	0	2	4	2	0	0	0
NCTC9075	2	4	2	4	4	3	1	1	0
NCTC9078	2	4	0	4	4	4	0	0	0
NCTC9098	5	10	3	32	40	30	2	0	2
NCTC9111	11	22	2	220	220	219	1	1	0
NCTC9112	10	20	4	180	180	178	2	2	0
NCTC9184	0	0	0	0	0	0	0	0	0
NCTC9645	16	32	2	366	480	365	1	1	0
NCTC9646	7	14	6	72	84	69	3	3	0
NCTC9695	1	2	0	0	0	0	0	0	0
Summary	5.32	10.6	3.27	78.6	87.3	76.8	1.77	1.34	0.432

Table A.5: Assembly graph statistics for a selection of 45 fragmented assemblies from the NCTC project. `Miniasm` assembly graph statistics: number of contigs, number of dead-end extremities. AAG statistics: theoretical maximal number number of edges. Note that for some of the most fragmented datasets (e.g. NCTC9184), none of the contigs pass the 100 Kbp length threshold, hence the AAG is empty. '*' denotes dataset for which the result is not available.

A.5 Contigs length and classification

Canu

Dataset	Contig name	Classification	Length
NCTC10006	tig00000055	chromosomal	635691
	tig00001802	chromosomal	4649423
	tig00001803	chromosomal	11996
	total chromosomal length		5297110
NCTC10332	tig00000001	chromosomal	3474338
	tig00000002	chromosomal	30165
	tig00000049	chromosomal	477163
	tig00000076	chromosomal	781581
	tig00000121	chromosomal	2609
	tig00000123	chromosomal	2461
	tig00000125	chromosomal	2452
	tig00009835	chromosomal	2395
	tig00009836	chromosomal	1564849
	tig00009837	chromosomal	11088
	tig00009838	chromosomal	2340
	tig00009839	chromosomal	2302
	total chromosomal length		6353743
NCTC10444	tig00000085	chromosomal	16691
	tig00000105	chromosomal	989155
	tig00000671	chromosomal	2391267
	tig00000672	chromosomal	14372
	tig00000673	chromosomal	1333749
	tig00000674	chromosomal	9774
	tig00000675	chromosomal	603044
	total chromosomal length		5358052
NCTC10702	tig00000001	chromosomal	1882575
	tig00000002	chromosomal	1048854
	tig00000080	chromosomal	70302
	total chromosomal length		3001731
	tig00000200	plasmidic	49994
	tig00001328	plasmidic	28893
	tig00001329	plasmidic	7575
	tig00000084	undefined	30012
	tig00000087	undefined	2442
	tig00000199	undefined	20259
NCTC10766	tig00000009	chromosomal	35058
	tig00000021	chromosomal	331047
	tig00001907	chromosomal	4740
	tig00001908	chromosomal	3602512
	tig00001909	chromosomal	15279
	tig00001910	chromosomal	700851
	tig00001911	chromosomal	14965
	tig00001912	chromosomal	10378
	tig00001913	chromosomal	20674
	tig00001915	chromosomal	10586
	tig00001916	chromosomal	9467
	tig00001921	chromosomal	7453
	tig00001922	chromosomal	710378
	total chromosomal length		5473388
	tig00000032	plasmidic	91068
	tig00000038	plasmidic	6441
	tig00000057	plasmidic	49757
	tig00001917	plasmidic	30098
	tig00001918	plasmidic	12494
	tig00001919	plasmidic	8557
	tig00001920	plasmidic	22116
	tig00000035	undefined	7262
	tig00000036	undefined	3368
NCTC10794	tig00000006	chromosomal	54912
	tig00000027	chromosomal	3322
	tig00000081	chromosomal	92448
	tig00000108	chromosomal	3328
	tig00000189	chromosomal	480759
	tig00000190	chromosomal	290320
	tig00004951	chromosomal	591102
	total chromosomal length		1516191
	tig00000003	undefined	186664
	tig00000014	undefined	105799
	tig00000039	undefined	180586
	tig00000040	undefined	18002
	tig00000042	undefined	42405
	tig00000047	undefined	41214
	tig00000067	undefined	3608
	tig00000072	undefined	3493
	tig00000098	undefined	3650
	tig00000100	undefined	3534
	tig00000102	undefined	3498
	tig00000110	undefined	1327
	tig00000114	undefined	3322
	tig00000116	undefined	3800
	tig00000187	undefined	16904
	tig00000188	undefined	12596
	tig00000191	undefined	12836
	tig00000192	undefined	12673
	tig00004950	undefined	4641
	tig00004952	undefined	105999
	tig00004953	undefined	11722
	tig00004954	undefined	6412
	tig00004955	undefined	9623
	tig00000096	none	5730
	tig00000112	none	1434
	tig00000186	none	5922
NCTC10988	tig00000006	chromosomal	25636
	tig00000279	chromosomal	3040963
	tig00000896	chromosomal	22144
	tig00000897	chromosomal	19058
	tig00000898	chromosomal	14604
	tig00000899	chromosomal	14371
	tig00000900	chromosomal	13453
	tig00000901	chromosomal	19223
	tig00000902	chromosomal	14801
	tig00000903	chromosomal	17641
	total chromosomal length		3201894
	tig00000088	undefined	4456
	tig00000105	undefined	36448
NCTC11126	tig00000037	chromosomal	577906
	tig00000074	chromosomal	666697

	tig00000192	chromosomal	1514971
	tig00000193	chromosomal	6302
	tig00000194	chromosomal	2066502
	tig00003788	chromosomal	8944
	tig00003789	chromosomal	46666
	total chromosomal length		4887988
NCTC11343	tig00000004	chromosomal	11272
	tig00000067	chromosomal	117209
	tig00000083	chromosomal	258828
	tig00000095	chromosomal	5614
	tig00000272	chromosomal	249754
	tig00000291	chromosomal	226277
	tig00000726	chromosomal	2208641
	tig00005693	chromosomal	12001
	tig00005694	chromosomal	876928
	tig00005696	chromosomal	3878
	tig00005698	chromosomal	8934
	tig00005699	chromosomal	214339
	total chromosomal length		4193675
	tig00000047	undefined	470559
	tig00000158	undefined	3400
	tig00000261	undefined	3105
	tig00000357	undefined	55998
	tig00000360	undefined	46002
	tig00000381	undefined	1809
	tig00000727	undefined	136723
	tig00000728	undefined	6803
	tig00000729	undefined	19083
	tig00000730	undefined	6003
	tig00005695	undefined	783131
	tig00005697	undefined	20459
	tig00005700	undefined	12630
	tig00005701	undefined	4319
	tig00005702	undefined	4350
	tig00005703	undefined	325592
	tig00005704	undefined	8727
NCTC11360	tig00000001	chromosomal	856759
	tig00000002	chromosomal	167905
	tig00000023	chromosomal	2941
	tig00000024	chromosomal	74373
	tig00000036	chromosomal	2775
	tig00000039	chromosomal	4498
	tig00000040	chromosomal	90923
	tig00000044	chromosomal	93819
	tig00000059	chromosomal	3339
	tig00000061	chromosomal	5726
	tig00000067	chromosomal	69601
	tig00000084	chromosomal	2711
	tig00000115	chromosomal	85300
	tig00000116	chromosomal	6833
	tig00000117	chromosomal	89165
	tig00000118	chromosomal	7948
	tig00000119	chromosomal	73822
	tig00000121	chromosomal	61437
	tig00000122	chromosomal	11175
	tig00000123	chromosomal	7718
	tig00002040	chromosomal	3708
	tig00002041	chromosomal	446407
	tig00002042	chromosomal	2667
	tig00002043	chromosomal	3997
	tig00002044	chromosomal	6676
	tig00002045	chromosomal	6999
	total chromosomal length		2189222
NCTC11435	tig00000001	chromosomal	1450647
	tig00000002	chromosomal	1627001
	tig00000267	chromosomal	3308
	tig00001171	chromosomal	213960
	tig00001172	chromosomal	11255
	tig00001173	chromosomal	260680
	tig00001174	chromosomal	13563
	tig00001175	chromosomal	871963
	total chromosomal length		4452377
NCTC11800	tig00000003	chromosomal	2775
	tig00000100	chromosomal	2617
	tig00000108	chromosomal	2163
	tig00000110	chromosomal	2403
	tig00000228	chromosomal	797974
	tig00000229	chromosomal	7782
	tig00003669	chromosomal	3650645
	total chromosomal length		4466359
	tig00000002	undefined	2722
	tig00000104	undefined	2273
	tig00003670	undefined	10818
NCTC11872	tig00000016	chromosomal	486488
	tig00000035	chromosomal	414114
	tig00000200	chromosomal	305814
	tig00000201	chromosomal	6170
	tig00000202	chromosomal	554139
	tig00000203	chromosomal	6679
	tig00000204	chromosomal	106287
	total chromosomal length		1879691
	tig00000072	undefined	1688
NCTC12123	tig00000001	chromosomal	2025792
	tig00000002	chromosomal	2402021
	tig00000009	chromosomal	319720
	tig00001215	chromosomal	17954
	tig00001216	chromosomal	19022
	total chromosomal length		4784509
	tig00000045	plasmidic	7248
	tig00001219	plasmidic	14495
	tig00001220	plasmidic	11169
	tig00001221	plasmidic	12838
	tig00000003	undefined	7552
	tig00000036	undefined	2048
	tig00001217	undefined	44732
	tig00001218	undefined	4652
NCTC12126	tig00000002	chromosomal	2504
	tig00000003	chromosomal	6347
	tig00000005	chromosomal	312284
	tig00000018	chromosomal	697355
	tig00000041	chromosomal	180413
	tig00000088	chromosomal	980155
	tig00000103	chromosomal	2869
	tig00000144	chromosomal	58545

	tig00000151	chromosomal	9045
	tig00000154	chromosomal	4231
	tig00000255	chromosomal	1991519
	tig00000256	chromosomal	3006
	tig00000257	chromosomal	620710
	total chromosomal length		4868983
	tig00000119	plasmidic	168880
NCTC12131	tig00000004	chromosomal	585202
	tig00000022	chromosomal	2720
	tig00000052	chromosomal	782381
	tig00000129	chromosomal	3215
	tig00000133	chromosomal	2760
	tig00000260	chromosomal	277244
	tig00000261	chromosomal	6991
	tig00000262	chromosomal	654135
	tig00000263	chromosomal	7044
	tig00000264	chromosomal	44446
	tig00000265	chromosomal	6070
	tig00000266	chromosomal	658311
	tig00000267	chromosomal	173281
	tig00000268	chromosomal	6652
	tig00000269	chromosomal	735339
	tig00000271	chromosomal	839355
	total chromosomal length		4785146
	tig00000272	undefined	11590
	tig00000273	undefined	3170
NCTC12132	tig00000001	chromosomal	2583454
	tig00000002	chromosomal	756442
	total chromosomal length		3339896
	tig00000004	undefined	20873
NCTC12146	tig00000001	chromosomal	4385596
	tig00001748	chromosomal	15170
	tig00001749	chromosomal	1248170
	total chromosomal length		5648936
NCTC12694	tig00000001	chromosomal	1305929
	tig00000004	chromosomal	723799
	tig00000010	chromosomal	270213
	tig00000013	chromosomal	244711
	tig00000015	chromosomal	205059
	tig00000017	chromosomal	163002
	tig00000019	chromosomal	200318
	tig00000021	chromosomal	138348
	tig00000028	chromosomal	101438
	tig00000031	chromosomal	87449
	tig00000032	chromosomal	63734
	tig00000035	chromosomal	90673
	tig00000038	chromosomal	41457
	tig00000040	chromosomal	64898
	tig00000042	chromosomal	69114
	tig00000045	chromosomal	37727
	tig00000047	chromosomal	27321
	tig00000052	chromosomal	2331
	tig00000091	chromosomal	753465
	tig00000092	chromosomal	2291
	tig00000093	chromosomal	2280
	total chromosomal length		4595557
	tig00000050	plasmidic	1930
	tig00006898	plasmidic	5801
	tig00006899	plasmidic	63765
NCTC12841	tig00000004	chromosomal	12036
	tig00000005	chromosomal	1851
	tig00000007	chromosomal	2368
	tig00000047	chromosomal	1630
	tig00000050	chromosomal	1416
	tig00000052	chromosomal	2797
	tig00000054	chromosomal	2185
	tig00000058	chromosomal	1348
	tig00000060	chromosomal	1588
	tig00000066	chromosomal	2323
	tig00000257	chromosomal	1926784
	tig00000258	chromosomal	11427
	tig00032866	chromosomal	17087
	tig00032867	chromosomal	11198
	tig00032868	chromosomal	1405
	tig00032869	chromosomal	1416
	total chromosomal length		1998859
NCTC12993	tig00000002	chromosomal	2655515
	tig00002251	chromosomal	2377976
	tig00002252	chromosomal	8006
	tig00002253	chromosomal	9235
	tig00002254	chromosomal	11903
	total chromosomal length		5062635
	tig00000055	plasmidic	12328
	tig00000063	plasmidic	5676
	tig00000064	plasmidic	2730
	tig00000113	plasmidic	5891
	tig00000052	undefined	222246
	tig00000114	undefined	4385
	tig00002255	undefined	9923
	tig00002256	undefined	13795
NCTC12998	tig00000002	chromosomal	2569
	tig00002880	chromosomal	5608109
	tig00002881	chromosomal	9135
	total chromosomal length		5619813
	tig00002882	plasmidic	126740
	tig00002883	plasmidic	7454
NCTC13095	tig00000036	chromosomal	2168596
	tig00000037	chromosomal	8008
	tig00000038	chromosomal	3511453
	total chromosomal length		5688057
	tig00000015	plasmidic	166342
	tig00001684	plasmidic	124320
	tig00001685	plasmidic	18225
	tig00000003	none	21738
NCTC13125	tig00000001	chromosomal	4777685
	tig00000003	chromosomal	461931
	tig00000408	chromosomal	263450
	tig00000409	chromosomal	19433
	tig00001778	chromosomal	18427
	tig00001779	chromosomal	24134
	total chromosomal length		5565060
	tig00000080	plasmidic	105599
	tig00000081	plasmidic	120877
	tig00000083	plasmidic	18752

	tig00000084	undefined	56975
	tig00000099	undefined	1213
NCTC13348	tig00000012	chromosomal	2875
	tig00000029	chromosomal	163558
	tig00000045	chromosomal	2613
	tig00000114	chromosomal	3490
	tig00000124	chromosomal	2641
	tig00000162	chromosomal	87300
	tig00000171	chromosomal	2696
	tig00000186	chromosomal	2783
	tig00000348	chromosomal	742809
	tig00000349	chromosomal	7200
	tig00000350	chromosomal	898431
	tig00000351	chromosomal	4061
	tig00000352	chromosomal	224612
	tig00000353	chromosomal	201081
	tig00000356	chromosomal	2525
	tig00005291	chromosomal	1458800
	tig00005292	chromosomal	3871
	tig00005293	chromosomal	1173550
	tig00005294	chromosomal	6497
	tig00005295	chromosomal	8532
	tig00005296	chromosomal	9588
	tig00005297	chromosomal	3796
	tig00005298	chromosomal	3770
	tig00005299	chromosomal	6030
	tig00005300	chromosomal	4132
	total chromosomal length		5027241
	tig00000183	plasmidic	99046
	tig00000196	undefined	4009
	tig00000355	undefined	2810
NCTC13463	tig00000066	chromosomal	4612761
	tig00000067	chromosomal	15891
	tig00000068	chromosomal	473422
	tig00000070	chromosomal	11585
	tig00000071	chromosomal	9027
	total chromosomal length		5122686
	tig00000024	plasmidic	99437
	tig00000028	plasmidic	3907
	tig00000026	undefined	9287
	tig00000069	undefined	63008
NCTC13543	tig00000001	chromosomal	2912152
	tig00000044	chromosomal	20274
	tig00000092	chromosomal	1100488
	total chromosomal length		4032914
	tig00000037	plasmidic	71251
	tig00000039	plasmidic	27238
	tig00000024	undefined	489748
	tig00000034	undefined	174464
	tig00000042	undefined	31750
	tig00000047	undefined	3595
	tig00000049	undefined	6247
	tig00000057	undefined	3104
	tig00000093	undefined	8383
	tig00000094	undefined	985883
NCTC4672	tig00000005	chromosomal	234563
	tig00000013	chromosomal	183355
	tig00000048	chromosomal	1061
	tig00000049	chromosomal	2728
	tig00000057	chromosomal	1112
	tig00000065	chromosomal	1150
	tig00000084	chromosomal	1399
	tig00000092	chromosomal	3917
	tig00000095	chromosomal	1292
	tig00000124	chromosomal	3541
	tig00000128	chromosomal	1599
	tig00000139	chromosomal	1711
	tig00000144	chromosomal	4980
	tig00000159	chromosomal	3137
	tig00000198	chromosomal	1889
	tig00000233	chromosomal	1995
	tig00000242	chromosomal	7213
	tig00000258	chromosomal	1676
	tig00000262	chromosomal	2808
	tig00000265	chromosomal	1307
	tig00000266	chromosomal	1405
	tig00000269	chromosomal	3265
	tig00000275	chromosomal	1624
	tig00000277	chromosomal	1819
	tig00000278	chromosomal	1609
	tig00000280	chromosomal	1497
	tig00000283	chromosomal	1155
	tig00000288	chromosomal	1859
	tig00000290	chromosomal	1529
	tig00000296	chromosomal	4494
	tig00000297	chromosomal	2018
	tig00000300	chromosomal	1525
	tig00000304	chromosomal	1433
	tig00000306	chromosomal	1315
	tig00000309	chromosomal	1535
	tig00000320	chromosomal	1446
	tig00000323	chromosomal	1479
	tig00000330	chromosomal	1947
	tig00000334	chromosomal	3660
	tig00000338	chromosomal	1749
	tig00000345	chromosomal	1368
	tig00000347	chromosomal	1669
	tig00000349	chromosomal	1420
	tig00000358	chromosomal	1659
	tig00000367	chromosomal	1044
	tig00000380	chromosomal	1237
	tig00000886	chromosomal	59611
	tig00000887	chromosomal	11888
	tig00000888	chromosomal	778559
	tig00000889	chromosomal	130493
	tig00000890	chromosomal	5574
	tig00000891	chromosomal	34698
	tig00000892	chromosomal	1261
	tig00000893	chromosomal	4516
	tig00012913	chromosomal	528826
	tig00012914	chromosomal	3329
	tig00012915	chromosomal	8743
	tig00012916	chromosomal	6640
	tig00012917	chromosomal	8651

	tig00012918	chromosomal	1378
	tig00012919	chromosomal	1701
	tig00012920	chromosomal	1535
	tig00012921	chromosomal	1435
	tig00012922	chromosomal	1444
	tig00012923	chromosomal	1179
	tig00012924	chromosomal	1180
	tig00012925	chromosomal	3460
	tig00012926	chromosomal	3441
	total chromosomal length		2108735
	tig00000046	undefined	3142
	tig00012927	undefined	1024
	tig00012928	undefined	1023
NCTC5050	tig00000001	chromosomal	3626030
	tig00000010	chromosomal	1250471
	tig00000023	chromosomal	227716
	tig00000041	chromosomal	3864
	total chromosomal length		5108081
	tig00000038	plasmidic	82367
	tig00000039	plasmidic	52025
	tig00000037	undefined	117821
NCTC5053	tig00000133	chromosomal	198522
	tig00000255	chromosomal	920215
	tig00000256	chromosomal	5841
	tig00000257	chromosomal	1006535
	tig00000258	chromosomal	6903
	tig00000259	chromosomal	2186965
	tig00003210	chromosomal	6218
	tig00003211	chromosomal	930059
	total chromosomal length		5261258
	tig00000136	plasmidic	112876
	tig00000143	plasmidic	105258
	tig00000146	plasmidic	13447
	tig00000160	plasmidic	9791
	tig00000261	plasmidic	209198
	tig00000260	undefined	9413
	tig00003209	undefined	107411
	tig00003212	undefined	10219
NCTC5055	tig00000055	chromosomal	11815
	tig00000057	chromosomal	12732
	tig00000059	chromosomal	6105
	tig00000060	chromosomal	4943
	tig00000064	chromosomal	5662
	tig00000065	chromosomal	15192
	tig00000070	chromosomal	3156
	tig00000074	chromosomal	4830
	tig00000076	chromosomal	4460
	tig00000077	chromosomal	7113
	tig00000078	chromosomal	5247
	tig00000080	chromosomal	4590
	tig00000081	chromosomal	7472
	tig00000082	chromosomal	2196
	tig00000084	chromosomal	9133
	tig00000094	chromosomal	5354
	tig00000095	chromosomal	3374
	tig00000096	chromosomal	4914
	tig00000097	chromosomal	9470
	tig00000098	chromosomal	6023
	tig00000100	chromosomal	2873
	tig00000101	chromosomal	3992
	tig00000102	chromosomal	3774
	tig00000106	chromosomal	5073
	tig00000107	chromosomal	8708
	tig00000109	chromosomal	6353
	tig00000110	chromosomal	3657
	tig00000112	chromosomal	2278
	tig00000116	chromosomal	3106
	tig00000117	chromosomal	2467
	tig00000119	chromosomal	4337
	tig00000122	chromosomal	3239
	tig00000127	chromosomal	3330
	tig00000129	chromosomal	3810
	tig00000130	chromosomal	8852
	tig00000133	chromosomal	4009
	tig00000151	chromosomal	1816
	tig00000153	chromosomal	4264
	tig00000154	chromosomal	9420
	tig00000155	chromosomal	3231
	tig00000156	chromosomal	3481
	tig00000158	chromosomal	2227
	tig00000159	chromosomal	5958
	tig00000160	chromosomal	3393
	tig00000161	chromosomal	2176
	tig00000162	chromosomal	2694
	tig00000163	chromosomal	2441
	tig00000165	chromosomal	1982
	tig00000167	chromosomal	8049
	tig00000168	chromosomal	3057
	tig00000169	chromosomal	4639
	tig00000171	chromosomal	5174
	tig00000172	chromosomal	4436
	tig00000176	chromosomal	2044
	tig00000177	chromosomal	3065
	tig00000179	chromosomal	5480
	tig00000180	chromosomal	5299
	tig00000181	chromosomal	7740
	tig00000182	chromosomal	3451
	tig00000183	chromosomal	3189
	tig00000184	chromosomal	1334
	tig00000186	chromosomal	3107
	tig00000187	chromosomal	2091
	tig00000189	chromosomal	2580
	tig00000190	chromosomal	1472
	tig00000191	chromosomal	8189
	tig00000192	chromosomal	5362
	tig00000193	chromosomal	3042
	tig00000194	chromosomal	6645
	tig00000195	chromosomal	1695
	tig00000196	chromosomal	1678
	tig00000202	chromosomal	2682
	tig00000203	chromosomal	9552
	tig00000204	chromosomal	3295
	tig00000209	chromosomal	6262
	tig00000212	chromosomal	6643

	tig00000220	chromosomal	3460
	tig00000225	chromosomal	2926
	tig00000229	chromosomal	5309
	tig00000256	chromosomal	3504
	tig00000263	chromosomal	1548
	tig00000264	chromosomal	2071
	tig00000266	chromosomal	7550
	tig00000267	chromosomal	1633
	tig00000269	chromosomal	2507
	tig00000273	chromosomal	3348
	tig00000274	chromosomal	3548
	tig00000275	chromosomal	4156
	tig00000277	chromosomal	3110
	tig00000279	chromosomal	4417
	tig00000281	chromosomal	3851
	tig00000288	chromosomal	5472
	tig00000289	chromosomal	4032
	tig00000291	chromosomal	3818
	tig00000292	chromosomal	4370
	tig00000293	chromosomal	3129
	tig00000294	chromosomal	2304
	tig00000296	chromosomal	3225
	tig00000301	chromosomal	7281
	tig00000305	chromosomal	8491
	tig00000314	chromosomal	5433
	tig00000317	chromosomal	3678
	tig00000325	chromosomal	1863
	tig00000327	chromosomal	3222
	tig00000328	chromosomal	5106
	tig00000333	chromosomal	3256
	tig00000341	chromosomal	1291
	tig00000342	chromosomal	2493
	tig00000346	chromosomal	1815
	tig00000353	chromosomal	2918
	tig00000355	chromosomal	4982
	tig00000357	chromosomal	2946
	tig00000358	chromosomal	1834
	tig00000360	chromosomal	2630
	tig00000364	chromosomal	3574
	tig00000370	chromosomal	2820
	tig00000378	chromosomal	8735
	tig00000381	chromosomal	3848
	tig00000382	chromosomal	2055
	tig00000386	chromosomal	2616
	tig00000387	chromosomal	1427
	tig00000389	chromosomal	1736
	tig00000397	chromosomal	5670
	tig00000401	chromosomal	2024
	tig00000407	chromosomal	3932
	tig00000409	chromosomal	4037
	tig00000426	chromosomal	1317
	tig00000429	chromosomal	5444
	tig00000430	chromosomal	3589
	tig00001790	chromosomal	22546
	tig00001791	chromosomal	4656080
	tig00008453	chromosomal	2243
	tig00008454	chromosomal	3013
	tig00008455	chromosomal	2506
	tig00008456	chromosomal	2503
	tig00008457	chromosomal	2363
	tig00008458	chromosomal	1846
	tig00008459	chromosomal	8988
	tig00008460	chromosomal	7418
	tig00008461	chromosomal	2516
	tig00008462	chromosomal	1286
	tig00008463	chromosomal	1256
	tig00008464	chromosomal	1283
	total chromosomal length		5275172
	tig00000062	plasmidic	18081
	tig00000105	plasmidic	10705
	tig00000121	plasmidic	7705
	tig00000157	plasmidic	2638
	tig00000228	plasmidic	5859
	tig00000336	plasmidic	1653
	tig00000366	plasmidic	1797
	tig00001789	plasmidic	274671
	tig00000173	undefined	4757
	tig00000270	undefined	5189
	tig00000282	undefined	2754
	tig00000306	undefined	2137
	tig00000308	undefined	5179
NCTC7152	tig00001521	chromosomal	4895392
	tig00001522	chromosomal	11663
	total chromosomal length		4907055
	tig00000020	plasmidic	140100
	tig00000021	plasmidic	22029
	tig00000023	plasmidic	17571
	tig00000004	undefined	12499
	tig00001524	undefined	9161
	tig00000002	none	14822
	tig00001523	none	13250
NCTC7922	tig00000005	chromosomal	30266
	tig00000010	chromosomal	231607
	tig00000015	chromosomal	8910
	tig00000061	chromosomal	624029
	tig00000089	chromosomal	224263
	tig00000120	chromosomal	118779
	tig00000357	chromosomal	3437368
	tig00000358	chromosomal	27476
	tig00000359	chromosomal	62893
	tig00000360	chromosomal	14447
	tig00000361	chromosomal	517854
	tig00004505	chromosomal	2628
	tig00004506	chromosomal	7127
	total chromosomal length		5307647
	tig00000123	plasmidic	92363
	tig00000136	plasmidic	68892
	tig00000137	plasmidic	2971
	tig00000138	undefined	2688
	tig00000140	undefined	8279
	tig00000143	undefined	5528
	tig00000356	undefined	9292
NCTC8179	tig00000002	chromosomal	32726
	tig00000005	chromosomal	34757

	tig00000006	chromosomal	156816
	tig00000012	chromosomal	932548
	tig00000140	chromosomal	32623
	tig00000141	chromosomal	1989140
	tig00000143	chromosomal	297068
	tig00000144	chromosomal	33325
	tig00000145	chromosomal	260864
	tig00000146	chromosomal	22495
	tig00000147	chromosomal	1150072
	tig00001520	chromosomal	24836
	tig00001521	chromosomal	21995
	tig00001522	chromosomal	17732
	tig00001523	chromosomal	732378
	total chromosomal length		5739375
	tig00000063	plasmidic	127915
	tig00000065	plasmidic	85310
	tig00000066	plasmidic	5132
	tig00000069	plasmidic	3833
	tig00000142	none	18135
NCTC8500	tig00000001	chromosomal	4654897
	tig00000069	chromosomal	14271
	tig00000172	chromosomal	2477
	total chromosomal length		4671645
	tig00000166	plasmidic	61752
NCTC8684	tig00000042	chromosomal	2510
	tig00000044	chromosomal	2653
	tig00000096	chromosomal	1675130
	tig00000100	chromosomal	9818
	tig00005015	chromosomal	2002
	total chromosomal length		1692113
	tig00000019	undefined	90777
	tig00000035	undefined	334610
	tig00000040	undefined	2954
	tig00000090	undefined	463211
	tig00000091	undefined	6937
	tig00000092	undefined	226539
	tig00000093	undefined	10565
	tig00000094	undefined	683840
	tig00000095	undefined	8091
	tig00000097	undefined	11829
	tig00000098	undefined	815147
	tig00000099	undefined	582249
	tig00005013	undefined	3845
	tig00005014	undefined	3834
NCTC9075	tig00000001	chromosomal	2771864
	tig00000014	chromosomal	707603
	tig00000055	chromosomal	975632
	tig00000129	chromosomal	250221
	tig00000196	chromosomal	115073
	tig00002929	chromosomal	6892
	tig00002930	chromosomal	441745
	total chromosomal length		5269030
	tig00000200	undefined	67419
NCTC9078	tig00000001	chromosomal	4157901
	tig00000006	chromosomal	11044
	tig00000036	chromosomal	1211
	tig00000051	chromosomal	1033327
	total chromosomal length		5203483
	tig00000025	plasmidic	84831
	tig00000052	plasmidic	15048
	tig00000053	plasmidic	141326
	tig00000050	undefined	13786
NCTC9098	tig00000001	chromosomal	3151410
	tig00000030	chromosomal	19458
	tig00000163	chromosomal	19823
	tig00000526	chromosomal	324234
	tig00000527	chromosomal	19807
	tig00000528	chromosomal	196308
	tig00000529	chromosomal	15991
	tig00000530	chromosomal	1487922
	total chromosomal length		5234953
	tig00000209	none	64136
	tig00000212	none	86222
NCTC9111	tig00000001	chromosomal	4605377
	tig00000032	chromosomal	15239
	tig00000054	chromosomal	151455
	tig00000063	chromosomal	586362
	tig00000064	chromosomal	28263
	tig00000186	chromosomal	5942
	tig00002643	chromosomal	30626
	tig00002644	chromosomal	14812
	tig00002645	chromosomal	16371
	total chromosomal length		5454447
	tig00000120	plasmidic	4002
	tig00000187	plasmidic	88084
	tig00002646	plasmidic	132127
	tig00002648	plasmidic	84308
	tig00002649	plasmidic	16303
	tig00002651	plasmidic	12651
	tig00000118	undefined	3898
	tig00000123	undefined	106160
	tig00002647	undefined	12391
	tig00002650	undefined	10115
	tig00002642	none	17615
NCTC9112	tig00000065	chromosomal	1280329
	tig00000084	chromosomal	1227588
	tig00000705	chromosomal	761814
	tig00000706	chromosomal	23527
	tig00000707	chromosomal	2213829
	tig00001864	chromosomal	23718
	tig00001865	chromosomal	26283
	total chromosomal length		5557088
NCTC9184	tig00000001	chromosomal	44206
	tig00000003	chromosomal	75248
	tig00000005	chromosomal	54514
	tig00000010	chromosomal	38058
	tig00000013	chromosomal	57613
	tig00000015	chromosomal	34569
	tig00000017	chromosomal	29672
	tig00000021	chromosomal	41507
	tig00000022	chromosomal	32208
	tig00000025	chromosomal	33831
	tig00000027	chromosomal	30328
	tig00000028	chromosomal	25544

tig00000029	chromosomal	31199
tig00000031	chromosomal	18451
tig00000032	chromosomal	26706
tig00000036	chromosomal	27618
tig00000039	chromosomal	27467
tig00000040	chromosomal	21778
tig00000042	chromosomal	28070
tig00000045	chromosomal	26501
tig00000046	chromosomal	23919
tig00000048	chromosomal	35573
tig00000049	chromosomal	18586
tig00000051	chromosomal	24356
tig00000055	chromosomal	35816
tig00000056	chromosomal	37501
tig00000057	chromosomal	9675
tig00000058	chromosomal	25028
tig00000059	chromosomal	21381
tig00000060	chromosomal	32086
tig00000061	chromosomal	22676
tig00000063	chromosomal	20847
tig00000065	chromosomal	16377
tig00000066	chromosomal	22324
tig00000069	chromosomal	23508
tig00000071	chromosomal	23542
tig00000072	chromosomal	24820
tig00000078	chromosomal	13426
tig00000082	chromosomal	23417
tig00000088	chromosomal	17003
tig00000089	chromosomal	15211
tig00000090	chromosomal	21564
tig00000091	chromosomal	10799
tig00000094	chromosomal	34765
tig00000095	chromosomal	16175
tig00000096	chromosomal	28943
tig00000099	chromosomal	2490
tig00000102	chromosomal	10959
tig00000104	chromosomal	15702
tig00000105	chromosomal	17032
tig00000113	chromosomal	17463
tig00000114	chromosomal	24382
tig00000115	chromosomal	6126
tig00000116	chromosomal	7311
tig00000117	chromosomal	6497
tig00000118	chromosomal	13154
tig00000119	chromosomal	19876
tig00000121	chromosomal	17839
tig00000122	chromosomal	10689
tig00000124	chromosomal	14467
tig00000128	chromosomal	16138
tig00000129	chromosomal	18515
tig00000134	chromosomal	15758
tig00000135	chromosomal	7877
tig00000139	chromosomal	12365
tig00000141	chromosomal	23830
tig00000145	chromosomal	15645
tig00000147	chromosomal	21070
tig00000148	chromosomal	31094
tig00000158	chromosomal	6592
tig00000159	chromosomal	12026
tig00000160	chromosomal	19542
tig00000161	chromosomal	16653
tig00000162	chromosomal	9525
tig00000163	chromosomal	3503
tig00000164	chromosomal	10038
tig00000168	chromosomal	22095
tig00000171	chromosomal	5815
tig00000172	chromosomal	3557
tig00000173	chromosomal	8034
tig00000174	chromosomal	13049
tig00000175	chromosomal	13166
tig00000176	chromosomal	4913
tig00000177	chromosomal	4186
tig00000182	chromosomal	16661
tig00000184	chromosomal	12911
tig00000187	chromosomal	10310
tig00000191	chromosomal	11302
tig00000193	chromosomal	10014
tig00000199	chromosomal	11611
tig00000201	chromosomal	14360
tig00000204	chromosomal	2382
tig00000210	chromosomal	10068
tig00000212	chromosomal	8977
tig00000223	chromosomal	17229
tig00000240	chromosomal	6878
tig00000241	chromosomal	15069
tig00000242	chromosomal	6620
tig00000245	chromosomal	15723
tig00000246	chromosomal	4700
tig00000248	chromosomal	5491
tig00000250	chromosomal	14542
tig00000252	chromosomal	20309
tig00000253	chromosomal	5109
tig00000254	chromosomal	6407
tig00000255	chromosomal	4126
tig00000263	chromosomal	18307
tig00000264	chromosomal	6065
tig00000269	chromosomal	2756
tig00000272	chromosomal	15386
tig00000273	chromosomal	10403
tig00000276	chromosomal	3194
tig00000280	chromosomal	10412
tig00000289	chromosomal	5925
tig00000297	chromosomal	2750
tig00000301	chromosomal	14266
tig00000305	chromosomal	6556
tig00000311	chromosomal	4992
tig00000315	chromosomal	5174
tig00000316	chromosomal	9510
tig00000318	chromosomal	3586
tig00003367	chromosomal	9616
tig00003368	chromosomal	55674
tig00003369	chromosomal	40990
tig00003370	chromosomal	4425
tig00003371	chromosomal	11626

	tig00003372	chromosomal	28757
	tig00003373	chromosomal	23908
	tig00003374	chromosomal	6632
	tig00003375	chromosomal	6460
	tig00003376	chromosomal	8901
	tig00003377	chromosomal	13617
	tig00003378	chromosomal	7801
	tig00003379	chromosomal	3818
	tig00003380	chromosomal	7550
	tig00003381	chromosomal	10743
	tig00003382	chromosomal	11567
	tig00003383	chromosomal	14003
	tig00003384	chromosomal	15516
	tig00003385	chromosomal	17588
	tig00003386	chromosomal	17512
	total chromosomal length		2470164
	tig00000107	plasmidic	15044
	tig00000166	plasmidic	10162
	tig00000186	plasmidic	12869
	tig00000188	plasmidic	18137
	tig00000299	plasmidic	2027
	tig00000180	undefined	13067
NCTC9645	tig00000007	chromosomal	2625
	tig00000011	chromosomal	607255
	tig00000013	chromosomal	599160
	tig00000021	chromosomal	40668
	tig00000024	chromosomal	405420
	tig00000026	chromosomal	317955
	tig00000036	chromosomal	103955
	tig00000037	chromosomal	234258
	tig00000042	chromosomal	220152
	tig00000047	chromosomal	201508
	tig00000052	chromosomal	207208
	tig00000058	chromosomal	2660
	tig00000061	chromosomal	135529
	tig00000094	chromosomal	27162
	tig00000096	chromosomal	18889
	tig00000098	chromosomal	20876
	tig00000101	chromosomal	5995
	tig00000102	chromosomal	1801
	tig00000105	chromosomal	2743
	tig00000109	chromosomal	1646
	tig00000113	chromosomal	1844
	tig00000206	chromosomal	382698
	tig00000207	chromosomal	7336
	tig00000208	chromosomal	1225204
	tig00000209	chromosomal	232251
	tig00000210	chromosomal	16029
	tig00000211	chromosomal	97614
	tig00000219	chromosomal	3443
	tig00000220	chromosomal	80500
	tig00012227	chromosomal	4751
	tig00012228	chromosomal	100031
	total chromosomal length		5309166
	tig00000072	plasmidic	10238
	tig00000086	plasmidic	8968
	tig00000212	plasmidic	82446
	tig00000213	plasmidic	14986
	tig00000214	plasmidic	32714
	tig00000215	plasmidic	99472
	tig00000217	plasmidic	11997
	tig00000218	plasmidic	87460
	tig00000221	plasmidic	2054
	tig00000222	plasmidic	13460
	tig00000035	undefined	168687
	tig00000088	undefined	7153
	tig00000069	none	81493
NCTC9646	tig00000001	chromosomal	3665711
	tig00000002	chromosomal	614927
	tig00000026	chromosomal	206992
	tig00000027	chromosomal	878265
	tig00000047	chromosomal	295064
	tig00000187	chromosomal	2534
	tig00003591	chromosomal	4056
	tig00003592	chromosomal	4764
	total chromosomal length		5672313
	tig00000022	plasmidic	148222
	tig00003589	plasmidic	8057
	tig00003590	plasmidic	6751
	tig00000021	undefined	36388
	tig00000063	undefined	1282
	tig00000065	undefined	1113
NCTC9695	tig00000074	chromosomal	1279605
	tig00000076	chromosomal	1894574
	total chromosomal length		3174179
	tig00000003	undefined	473776
	tig00000004	undefined	204759
	tig00000019	undefined	4937
	tig00000038	undefined	3861
	tig00000040	undefined	2672
	tig00000042	undefined	2170
	tig00000075	undefined	7627
	tig00000077	undefined	7294
	tig00000078	undefined	911580

Table A.6: Canu contigs classification per NCTC dataset. Total length of chromosomal contigs is given.

Miniasm

Dataset	Contig name	Classification	Length
NCTC10006	utg000001l	chromosomal	260336
	utg000002l	chromosomal	1081553
	utg000003l	chromosomal	1615186
	utg000004l	chromosomal	1435892
	utg000005l	chromosomal	629371
	utg000006l	chromosomal	301502
	utg000007l	chromosomal	263124
	total chromosomal length		5586964
NCTC10332	utg000002l	chromosomal	213696
	utg000003l	chromosomal	598526
	utg000004l	chromosomal	220355
	utg000005l	chromosomal	687999
	utg000006l	chromosomal	92810
	utg000007l	chromosomal	274321
	utg000008l	chromosomal	889152
	utg000009l	chromosomal	236367
	utg000010l	chromosomal	62450
	utg000011l	chromosomal	436257
	utg000012l	chromosomal	720033
	utg000013l	chromosomal	191547
	utg000014l	chromosomal	301467
	utg000015l	chromosomal	41317
	utg000016l	chromosomal	350649
	utg000017l	chromosomal	273059
	utg000018l	chromosomal	339294
	utg000019l	chromosomal	65630
	utg000020l	chromosomal	81777
	utg000022l	chromosomal	43390
	utg000023l	chromosomal	42183
	utg000024l	chromosomal	16950
	total chromosomal length		6179229
	utg000011l	none	274988
	utg000021l	none	57736
NCTC10444	utg000001l	chromosomal	2018895
	utg000002l	chromosomal	1852358
	utg000003l	chromosomal	240957
	utg000004l	chromosomal	1224505
	utg000005c	chromosomal	234694
	total chromosomal length		5571409
	utg000006c	none	4134
NCTC10702	utg000001c	chromosomal	3036414
	utg000004l	chromosomal	5937
	total chromosomal length		3042351
	utg000003l	plasmidic	36874
	utg000002c	undefined	34724
NCTC10766	utg000001l	chromosomal	424892
	utg000002l	chromosomal	360757
	utg000003l	chromosomal	3136691
	utg000004l	chromosomal	991822
	utg000005l	chromosomal	814775
	utg000007l	chromosomal	17423
	utg000010l	chromosomal	76701
	total chromosomal length		5823061
	utg000006c	plasmidic	56779
	utg000008c	plasmidic	88390

	utg0000091	undefined	7051
	utg000011c	undefined	6079
NCTC10794	utg0000011	chromosomal	686754
	utg0000031	chromosomal	73314
	utg0000041	chromosomal	198317
	utg0000051	chromosomal	344693
	utg0000081	chromosomal	84572
	total chromosomal length		1387650
	utg0000021	undefined	23304
	utg0000061	undefined	618933
	utg0000071	undefined	122221
	utg0000091	undefined	50190
	utg0000101	undefined	18666
NCTC10988	utg0000011	chromosomal	470745
	utg0000021	chromosomal	1143622
	utg0000031	chromosomal	39170
	utg0000041	chromosomal	1521633
	utg0000051	chromosomal	35669
	utg0000061	chromosomal	36182
	utg0000071	chromosomal	28025
	utg0000091	chromosomal	23011
	utg0000111	chromosomal	25778
	total chromosomal length		3323835
	utg000010c	undefined	27255
	utg000008c	none	1813
NCTC11126	utg0000011	chromosomal	799550
	utg0000031	chromosomal	39468
	utg0000051	chromosomal	199017
	utg0000061	chromosomal	654158
	utg0000071	chromosomal	801856
	utg0000081	chromosomal	150048
	utg0000091	chromosomal	615460
	utg0000101	chromosomal	446084
	utg0000121	chromosomal	187190
	utg0000131	chromosomal	24165
	utg0000151	chromosomal	124049
	utg0000161	chromosomal	115589
	utg0000171	chromosomal	214509
	utg0000181	chromosomal	36740
	utg0000191	chromosomal	18016
	total chromosomal length		4425899
	utg0000021	undefined	463729
	utg0000041	none	129544
	utg0000111	none	152040
	utg0000141	none	20861
NCTC11343	utg0000011	chromosomal	1137077
	utg0000081	chromosomal	387994
	utg0000091	chromosomal	303645
	utg0000131	chromosomal	73842
	utg0000171	chromosomal	44820
	utg0000211	chromosomal	71812
	utg0000231	chromosomal	82330
	utg0000261	chromosomal	37829
	utg0000271	chromosomal	20725
	utg0000281	chromosomal	10169
	total chromosomal length		2170243
	utg0000021	undefined	110639
	utg0000031	undefined	629323
	utg0000041	undefined	483485
	utg0000051	undefined	88713
	utg0000061	undefined	94951
	utg0000071	undefined	268271
	utg0000101	undefined	265843
	utg0000111	undefined	186796
	utg0000121	undefined	244669
	utg0000141	undefined	739847
	utg0000151	undefined	328310
	utg0000161	undefined	159212
	utg0000181	undefined	95506
	utg0000191	undefined	84352
	utg0000201	undefined	49752
	utg0000221	undefined	78366
	utg0000241	undefined	25887
	utg0000251	undefined	63812
NCTC11360	utg0000011	chromosomal	83040
	utg0000021	chromosomal	142687
	utg0000031	chromosomal	86224
	utg0000041	chromosomal	117971
	utg0000051	chromosomal	103040
	utg0000061	chromosomal	379750
	utg0000071	chromosomal	73713
	utg0000081	chromosomal	87575
	utg0000091	chromosomal	173405
	utg0000101	chromosomal	39660
	utg0000111	chromosomal	75956
	utg0000121	chromosomal	43377
	utg0000131	chromosomal	104822
	utg0000141	chromosomal	66226
	utg0000151	chromosomal	97577
	utg0000161	chromosomal	22672
	utg0000171	chromosomal	68226
	utg0000181	chromosomal	31089
	utg0000191	chromosomal	135026
	utg0000201	chromosomal	12813
	utg0000211	chromosomal	59788
	utg0000221	chromosomal	18821
	utg0000231	chromosomal	10147
	utg0000241	chromosomal	14492
	utg0000251	chromosomal	12493
	total chromosomal length		2060590
NCTC11435	utg0000011	chromosomal	348162
	utg000002c	chromosomal	1514816
	utg0000031	chromosomal	992029
	utg0000041	chromosomal	641762
	utg0000051	chromosomal	861119
	utg0000061	chromosomal	287740
	total chromosomal length		4645628
	utg000007c	none	1992
NCTC11800	utg0000011	chromosomal	3251923
	utg0000021	chromosomal	430158
	utg0000031	chromosomal	887997
	total chromosomal length		4570078
	utg0000041	undefined	190071
NCTC11872	utg0000011	chromosomal	171196

	utg000002l	chromosomal	82073
	utg000003l	chromosomal	411282
	utg000004l	chromosomal	188111
	utg000005l	chromosomal	116854
	utg000006l	chromosomal	68409
	utg000007l	chromosomal	132209
	utg000008l	chromosomal	135142
	utg000009l	chromosomal	198320
	utg000010l	chromosomal	170105
	utg000011l	chromosomal	62719
	utg000012l	chromosomal	193447
	utg000013l	chromosomal	33052
	total chromosomal length		1962919
NCTC12123	utg000001l	chromosomal	2853675
	utg000002l	chromosomal	2105813
	utg000003l	chromosomal	40089
	total chromosomal length		4999577
	utg000005c	plasmidic	10039
	utg000004l	undefined	31948
NCTC12126	utg000001l	chromosomal	319898
	utg000002l	chromosomal	731428
	utg000003l	chromosomal	2015098
	utg000004l	chromosomal	424548
	utg000005l	chromosomal	234649
	utg000006l	chromosomal	95263
	utg000007l	chromosomal	401716
	utg000008l	chromosomal	165317
	utg000009l	chromosomal	79334
	utg000010l	chromosomal	121403
	utg000011l	chromosomal	117059
	utg000013l	chromosomal	100109
	utg000014l	chromosomal	144200
	utg000015l	chromosomal	64609
	utg000016l	chromosomal	59101
	total chromosomal length		5073732
	utg000012l	undefined	169903
NCTC12131	utg000001l	chromosomal	111790
	utg000002l	chromosomal	92827
	utg000003l	chromosomal	97030
	utg000004l	chromosomal	124270
	utg000005l	chromosomal	69426
	utg000007l	chromosomal	26064
	utg000008l	chromosomal	39017
	utg000009l	chromosomal	161140
	utg000010l	chromosomal	67339
	utg000013l	chromosomal	30391
	utg000014l	chromosomal	100618
	utg000015l	chromosomal	125621
	utg000016l	chromosomal	60566
	utg000017l	chromosomal	16554
	utg000018l	chromosomal	195541
	utg000019l	chromosomal	101170
	utg000020l	chromosomal	130409
	utg000021l	chromosomal	49770
	utg000022l	chromosomal	84913
	utg000023l	chromosomal	78770
	utg000024l	chromosomal	257054
	utg000025l	chromosomal	154990
	utg000026l	chromosomal	38811
	utg000027l	chromosomal	68728
	utg000028l	chromosomal	46799
	utg000030l	chromosomal	161317
	utg000031l	chromosomal	71407
	utg000032l	chromosomal	77866
	utg000033l	chromosomal	86540
	utg000034l	chromosomal	114741
	utg000036l	chromosomal	60369
	utg000037l	chromosomal	28783
	utg000038l	chromosomal	71999
	utg000039l	chromosomal	64473
	utg000040l	chromosomal	72097
	utg000041l	chromosomal	12003
	utg000042l	chromosomal	19063
	utg000043l	chromosomal	29043
	utg000044l	chromosomal	36339
	utg000045l	chromosomal	134261
	utg000046l	chromosomal	56314
	utg000047l	chromosomal	101727
	utg000048l	chromosomal	16300
	utg000049l	chromosomal	101615
	utg000050l	chromosomal	25187
	utg000051l	chromosomal	23598
	utg000052l	chromosomal	17725
	utg000053l	chromosomal	106162
	utg000054l	chromosomal	18067
	utg000055l	chromosomal	87197
	utg000056l	chromosomal	38172
	utg000057l	chromosomal	68391
	utg000058l	chromosomal	38284
	utg000061l	chromosomal	73927
	utg000062l	chromosomal	34241
	utg000063l	chromosomal	30728
	utg000064l	chromosomal	22173
	utg000065l	chromosomal	15783
	utg000066l	chromosomal	25123
	utg000067l	chromosomal	49646
	utg000068l	chromosomal	15064
	utg000069l	chromosomal	38231
	utg000070l	chromosomal	25850
	utg000072l	chromosomal	103746
	utg000073l	chromosomal	32616
	utg000074l	chromosomal	20672
	utg000075l	chromosomal	20049
	utg000076l	chromosomal	16489
	utg000078l	chromosomal	7701
	utg000079l	chromosomal	17282
	utg000080l	chromosomal	25386
	utg000081l	chromosomal	13893
	utg000082l	chromosomal	21851
	utg000084l	chromosomal	6442
	utg000085l	chromosomal	4765
	utg000086l	chromosomal	8500
	utg000087l	chromosomal	5363
	total chromosomal length		4704169

	utg0000061	undefined	14352
	utg0000111	undefined	40323
	utg0000121	undefined	26155
	utg0000291	undefined	24896
	utg0000591	undefined	21237
	utg0000601	undefined	5107
	utg0000711	undefined	44778
	utg0000771	undefined	11705
	utg0000831	undefined	9380
	utg0000351	none	65396
NCTC12132	utg0000011	chromosomal	2895268
	utg000002c	chromosomal	536810
	utg0000031	chromosomal	69541
	utg0000041	chromosomal	19313
	total chromosomal length		3520932
NCTC12146	utg0000011	chromosomal	5930232
	total chromosomal length		5930232
NCTC12694	utg0000011	chromosomal	58159
	utg0000021	chromosomal	35524
	utg0000031	chromosomal	28409
	utg0000041	chromosomal	16449
	utg0000051	chromosomal	17394
	utg0000061	chromosomal	21530
	utg0000071	chromosomal	11853
	utg0000081	chromosomal	8442
	utg0000091	chromosomal	10039
	utg0000101	chromosomal	49546
	utg0000111	chromosomal	26813
	utg0000121	chromosomal	23265
	utg0000131	chromosomal	21153
	utg0000141	chromosomal	12511
	utg0000151	chromosomal	17131
	utg0000161	chromosomal	42769
	utg0000171	chromosomal	12766
	utg0000181	chromosomal	24545
	utg0000191	chromosomal	23162
	utg0000201	chromosomal	19756
	utg0000211	chromosomal	45548
	utg0000221	chromosomal	21115
	utg0000231	chromosomal	25591
	utg0000241	chromosomal	30239
	utg0000251	chromosomal	21095
	utg0000261	chromosomal	9863
	utg0000271	chromosomal	23159
	utg0000291	chromosomal	8102
	utg0000301	chromosomal	11082
	utg0000311	chromosomal	28054
	utg0000331	chromosomal	42654
	utg0000341	chromosomal	30933
	utg0000351	chromosomal	46346
	utg0000361	chromosomal	30903
	utg0000371	chromosomal	25189
	utg0000381	chromosomal	18193
	utg0000391	chromosomal	43343
	utg0000401	chromosomal	30397
	utg0000411	chromosomal	15935
	utg0000421	chromosomal	19588
	utg0000431	chromosomal	18820
	utg0000441	chromosomal	23764
	utg0000451	chromosomal	27579
	utg0000461	chromosomal	26394
	utg0000471	chromosomal	18649
	utg0000481	chromosomal	20699
	utg0000491	chromosomal	28188
	utg0000501	chromosomal	7493
	utg0000511	chromosomal	46315
	utg0000531	chromosomal	5107
	utg0000541	chromosomal	40004
	utg0000551	chromosomal	20811
	utg0000561	chromosomal	6201
	utg0000571	chromosomal	42503
	utg0000581	chromosomal	9998
	utg0000591	chromosomal	14321
	utg0000601	chromosomal	22892
	utg0000611	chromosomal	28867
	utg0000621	chromosomal	28898
	utg0000641	chromosomal	24612
	utg0000651	chromosomal	41956
	utg0000661	chromosomal	22542
	utg0000671	chromosomal	18013
	utg0000681	chromosomal	30693
	utg0000691	chromosomal	20647
	utg0000701	chromosomal	35946
	utg0000711	chromosomal	26657
	utg0000721	chromosomal	18854
	utg0000731	chromosomal	8301
	utg0000741	chromosomal	6505
	utg0000751	chromosomal	18941
	utg0000761	chromosomal	17232
	utg0000771	chromosomal	21805
	utg0000781	chromosomal	14634
	utg0000791	chromosomal	37447
	utg0000801	chromosomal	12665
	utg0000811	chromosomal	39494
	utg0000821	chromosomal	17950
	utg0000831	chromosomal	21934
	utg0000841	chromosomal	22518
	utg0000851	chromosomal	7041
	utg0000861	chromosomal	28654
	utg0000871	chromosomal	16884
	utg0000881	chromosomal	28723
	utg0000891	chromosomal	16734
	utg0000901	chromosomal	13996
	utg0000911	chromosomal	23988
	utg0000921	chromosomal	14611
	utg0000931	chromosomal	7501
	utg0000941	chromosomal	25459
	utg0000951	chromosomal	4562
	utg0000961	chromosomal	47136
	utg0000971	chromosomal	4814
	utg0000981	chromosomal	25935
	utg0000991	chromosomal	10951
	utg0001001	chromosomal	21525
	utg0001011	chromosomal	10732

	utg000102l	chromosomal	6168
	utg000103l	chromosomal	15062
	utg000104l	chromosomal	11927
	utg000105l	chromosomal	10715
	utg000106l	chromosomal	7331
	utg000107l	chromosomal	10881
	utg000108l	chromosomal	15574
	utg000109l	chromosomal	27587
	utg000110l	chromosomal	23469
	utg000111l	chromosomal	12398
	utg000112l	chromosomal	57852
	utg000113l	chromosomal	13429
	utg000114l	chromosomal	15167
	utg000115l	chromosomal	29785
	utg000116l	chromosomal	21769
	utg000117l	chromosomal	8477
	utg000118l	chromosomal	16251
	utg000119l	chromosomal	21816
	utg000120l	chromosomal	4937
	utg000121l	chromosomal	15924
	utg000122l	chromosomal	14350
	utg000123l	chromosomal	17284
	utg000124l	chromosomal	14773
	utg000125l	chromosomal	13950
	utg000126l	chromosomal	12821
	utg000127l	chromosomal	12776
	total chromosomal length		2667113
	utg000028l	plasmidic	39729
	utg000063l	plasmidic	24889
	utg000032l	none	10367
	utg000052l	none	44932
NCTC12841	utg0000011	chromosomal	2025517
	total chromosomal length		2025517
NCTC12993	utg0000011	chromosomal	4585710
	utg000002l	chromosomal	728592
	total chromosomal length		5314302
	utg000004l	plasmidic	83189
	utg000006l	plasmidic	25452
	utg000007l	plasmidic	6215
	utg000003l	undefined	109233
	utg000005l	undefined	74705
NCTC12998	utg0000011	chromosomal	669883
	utg000002l	chromosomal	3936078
	utg000003l	chromosomal	584057
	utg000004l	chromosomal	737303
	total chromosomal length		5927321
	utg000005c	plasmidic	125518
NCTC13095	utg000001c	chromosomal	5922307
	utg000003l	chromosomal	34891
	total chromosomal length		5957198
	utg000002c	undefined	117186
	utg000004c	undefined	164749
NCTC13125	utg0000011	chromosomal	4270665
	utg000002l	chromosomal	470580
	utg000003l	chromosomal	287719
	utg000004l	chromosomal	62610
	utg000005l	chromosomal	392905
	utg000006l	chromosomal	215108
	utg000009l	chromosomal	25575
	total chromosomal length		5725162
	utg000007c	plasmidic	105857
	utg000008c	plasmidic	93624
	utg000010c	undefined	41914
NCTC13348	utg0000011	chromosomal	297750
	utg000002l	chromosomal	152446
	utg000003l	chromosomal	219639
	utg000004l	chromosomal	237470
	utg000006l	chromosomal	471362
	utg000007l	chromosomal	399972
	utg000008l	chromosomal	226216
	utg000009l	chromosomal	979865
	utg000010l	chromosomal	154813
	utg000011l	chromosomal	97347
	utg000012l	chromosomal	283827
	utg000013l	chromosomal	408789
	utg000014l	chromosomal	469421
	utg000015l	chromosomal	431950
	utg000016l	chromosomal	157498
	utg000017l	chromosomal	129726
	utg000018l	chromosomal	18621
	total chromosomal length		5136712
	utg000005l	plasmidic	105592
NCTC13463	utg0000011	chromosomal	2602844
	utg000002l	chromosomal	476415
	utg000003l	chromosomal	2106813
	utg000006l	chromosomal	151619
	total chromosomal length		5337691
	utg000004c	plasmidic	89940
	utg000005c	undefined	50683
	utg000007l	undefined	2881
NCTC13543	utg000001c	chromosomal	3006943
	utg000002l	chromosomal	2166302
	utg000007l	chromosomal	54242
	total chromosomal length		5227487
	utg000005l	plasmidic	24866
	utg000006l	plasmidic	57848
	utg000008l	plasmidic	14511
	utg000004c	undefined	509202
	utg000009l	undefined	15019
	utg000003l	none	163283
NCTC4672	utg0000011	chromosomal	390339
	utg000002l	chromosomal	234263
	utg000003l	chromosomal	205142
	utg000004l	chromosomal	194700
	utg000005l	chromosomal	143175
	utg000006l	chromosomal	145320
	utg000007l	chromosomal	20463
	utg000008l	chromosomal	72927
	utg000009l	chromosomal	171587
	utg000010l	chromosomal	152272
	utg000011l	chromosomal	39361
	utg000012l	chromosomal	68994
	utg000013l	chromosomal	26376
	utg000014l	chromosomal	41255

	utg000015l	chromosomal	60948
	utg000016l	chromosomal	13117
	total chromosomal length		1980239
NCTC5050	utg000001l	chromosomal	1602894
	utg000002l	chromosomal	1353385
	utg000003l	chromosomal	1365254
	utg000004l	chromosomal	1153772
	total chromosomal length		5475305
	utg000005c	plasmidic	116760
	utg000006c	plasmidic	40870
	utg000007c	plasmidic	76667
	utg000008l	none	37817
NCTC5053	utg000001l	chromosomal	1215813
	utg000002l	chromosomal	334789
	utg000004l	chromosomal	1438231
	utg000006l	chromosomal	675061
	utg000007l	chromosomal	225750
	utg000008l	chromosomal	372066
	utg000009l	chromosomal	188575
	utg000010l	chromosomal	677706
	utg000011l	chromosomal	244959
	utg000012l	chromosomal	63154
	utg000013l	chromosomal	26449
	total chromosomal length		5462553
	utg000005c	plasmidic	208263
	utg000014c	plasmidic	102763
	utg000015c	plasmidic	104579
	utg000016c	plasmidic	12712
	utg000018c	plasmidic	5167
	utg000003c	undefined	105118
	utg000017c	none	2115
NCTC5055	utg000001l	chromosomal	5214489
	utg000002l	chromosomal	13329
	utg000004l	chromosomal	16397
	utg000006l	chromosomal	11993
	utg000007l	chromosomal	11530
	utg000008l	chromosomal	23643
	utg000010l	chromosomal	12125
	utg000011l	chromosomal	20225
	utg000012l	chromosomal	11167
	utg000013l	chromosomal	10465
	utg000014l	chromosomal	11390
	utg000015l	chromosomal	21429
	utg000016l	chromosomal	11492
	utg000017l	chromosomal	14583
	utg000018l	chromosomal	10242
	utg000019l	chromosomal	20037
	utg000020l	chromosomal	12298
	utg000022l	chromosomal	21719
	utg000023l	chromosomal	20549
	utg000024l	chromosomal	7808
	total chromosomal length		5496910
	utg000003l	plasmidic	19472
	utg000009l	plasmidic	34026
	utg000021l	plasmidic	17587
	utg000005l	undefined	10762
NCTC7152	utg000001l	chromosomal	4797250
	utg000003l	chromosomal	446534
	total chromosomal length		5243784
	utg000002c	plasmidic	140333
	utg000004l	plasmidic	45336
NCTC7922	utg000001l	chromosomal	223615
	utg000002l	chromosomal	3332503
	utg000003l	chromosomal	633327
	utg000004l	chromosomal	213700
	utg000005l	chromosomal	47119
	utg000006l	chromosomal	541108
	utg000007l	chromosomal	353127
	utg000010l	chromosomal	56267
	utg000011l	chromosomal	15478
	total chromosomal length		5416244
	utg000008c	plasmidic	84907
	utg000009c	plasmidic	59289
	utg000012c	undefined	5073
	utg000013c	undefined	5631
	utg000014c	none	1251
NCTC8179	utg000001l	chromosomal	865090
	utg000002l	chromosomal	1960752
	utg000003l	chromosomal	277217
	utg000004l	chromosomal	786279
	utg000005l	chromosomal	111098
	utg000006l	chromosomal	93197
	utg000007l	chromosomal	293360
	utg000008l	chromosomal	24186
	utg000010l	chromosomal	292276
	utg000012l	chromosomal	125358
	utg000013l	chromosomal	265142
	utg000014l	chromosomal	277553
	utg000015l	chromosomal	294414
	utg000016l	chromosomal	32286
	utg000017l	chromosomal	22000
	total chromosomal length		5720208
	utg000009c	plasmidic	118435
	utg000011c	plasmidic	71383
	utg000018c	none	3105
NCTC8500	utg000001c	chromosomal	4831304
	total chromosomal length		4831304
	utg000002c	plasmidic	55643
NCTC8684	utg000001l	chromosomal	1077377
	utg000006l	chromosomal	1059624
	total chromosomal length		2137001
	utg000002l	undefined	750049
	utg000003l	undefined	701800
	utg000004l	undefined	762169
	utg000005l	undefined	622781
	utg000007l	undefined	49586
NCTC9075	utg000001l	chromosomal	2460452
	utg000002l	chromosomal	3016645
	utg000004l	chromosomal	9967
	total chromosomal length		5487064
	utg000003c	undefined	52022
NCTC9078	utg000001c	chromosomal	4242489
	utg000002c	chromosomal	1146428
	total chromosomal length		5388917

	utg000004c	plasmidic	87254
	utg000003c	none	132921
NCTC9098	utg000001l	chromosomal	3054707
	utg000002l	chromosomal	1563634
	utg000003l	chromosomal	219418
	utg000005l	chromosomal	337874
	utg000007l	chromosomal	29947
	utg000008l	chromosomal	288339
	total chromosomal length		5493919
	utg000004c	plasmidic	52547
	utg000006c	plasmidic	88507
	utg000009c	none	3685
	utg000010c	none	2517
NCTC9111	utg000002l	chromosomal	382296
	utg000003l	chromosomal	258361
	utg000004l	chromosomal	1397867
	utg000005l	chromosomal	120406
	utg000006l	chromosomal	360884
	utg000007l	chromosomal	354979
	utg000008l	chromosomal	393118
	utg000009l	chromosomal	780946
	utg000010l	chromosomal	496328
	utg000011c	chromosomal	751321
	utg000013l	chromosomal	14110
	utg000014l	chromosomal	298974
	utg000018c	chromosomal	9151
	total chromosomal length		5618741
	utg000001c	plasmidic	126297
	utg000012c	plasmidic	92083
	utg000015c	plasmidic	71898
	utg000016c	undefined	5475
	utg000017c	undefined	97293
NCTC9112	utg000001l	chromosomal	267187
	utg000002l	chromosomal	38684
	utg000003l	chromosomal	799223
	utg000004l	chromosomal	226433
	utg000005l	chromosomal	1445001
	utg000006l	chromosomal	583111
	utg000007l	chromosomal	772601
	utg000008l	chromosomal	739996
	utg000009l	chromosomal	283544
	utg000010l	chromosomal	537734
	utg000011l	chromosomal	134545
	utg000012l	chromosomal	20182
	utg000013l	chromosomal	22078
	utg000014l	chromosomal	19079
	utg000015l	chromosomal	17055
	total chromosomal length		5906453
	utg000016c	none	786
NCTC9184	utg000001l	chromosomal	29829
	utg000002l	chromosomal	18494
	utg000004l	chromosomal	16159
	utg000005l	chromosomal	24250
	utg000006l	chromosomal	18494
	utg000007l	chromosomal	21484
	utg000008l	chromosomal	27067
	utg000009l	chromosomal	16993
	utg000010l	chromosomal	23540
	utg000011l	chromosomal	22732
	utg000012l	chromosomal	6381
	utg000013l	chromosomal	29998
	utg000014l	chromosomal	22501
	utg000015l	chromosomal	22166
	utg000016l	chromosomal	14255
	utg000017l	chromosomal	13452
	utg000018l	chromosomal	19121
	total chromosomal length		346916
	utg000003c	none	4242
NCTC9645	utg000001l	chromosomal	43611
	utg000002l	chromosomal	159710
	utg000003l	chromosomal	60098
	utg000004l	chromosomal	148353
	utg000005l	chromosomal	202718
	utg000006l	chromosomal	19551
	utg000007l	chromosomal	50893
	utg000008l	chromosomal	131157
	utg000009l	chromosomal	154075
	utg000010l	chromosomal	213751
	utg000011l	chromosomal	29643
	utg000012l	chromosomal	39758
	utg000013l	chromosomal	79432
	utg000014l	chromosomal	47565
	utg000016l	chromosomal	24983
	utg000017l	chromosomal	60388
	utg000018l	chromosomal	36168
	utg000019l	chromosomal	96948
	utg000020l	chromosomal	163752
	utg000021l	chromosomal	90608
	utg000022l	chromosomal	56793
	utg000023l	chromosomal	182187
	utg000024l	chromosomal	51393
	utg000025l	chromosomal	53299
	utg000026l	chromosomal	128083
	utg000027l	chromosomal	124407
	utg000028l	chromosomal	48273
	utg000029l	chromosomal	126118
	utg000030l	chromosomal	43482
	utg000032l	chromosomal	169005
	utg000033l	chromosomal	48040
	utg000034l	chromosomal	28371
	utg000035l	chromosomal	54647
	utg000036l	chromosomal	17500
	utg000037l	chromosomal	99129
	utg000038l	chromosomal	109018
	utg000039l	chromosomal	169360
	utg000040l	chromosomal	79322
	utg000041l	chromosomal	29162
	utg000042l	chromosomal	12864
	utg000043l	chromosomal	135445
	utg000044l	chromosomal	77604
	utg000045l	chromosomal	212393
	utg000046l	chromosomal	53152
	utg000047l	chromosomal	23857
	utg000048l	chromosomal	87995

	utg000050l	chromosomal	8353
	utg000053l	chromosomal	61272
	utg000054l	chromosomal	19953
	utg000055l	chromosomal	76527
	utg000056l	chromosomal	30382
	utg000057l	chromosomal	38221
	utg000058l	chromosomal	67870
	utg000059l	chromosomal	33128
	utg000060l	chromosomal	12588
	utg000061l	chromosomal	93957
	utg000062l	chromosomal	37019
	utg000063l	chromosomal	24109
	utg000064l	chromosomal	68089
	utg000066l	chromosomal	39231
	utg000067l	chromosomal	53886
	utg000068l	chromosomal	49692
	utg000069l	chromosomal	69014
	utg000070l	chromosomal	15069
	utg000071l	chromosomal	12499
	utg000072l	chromosomal	34026
	utg000073l	chromosomal	33264
	utg000074l	chromosomal	11225
	utg000075l	chromosomal	75997
	utg000077l	chromosomal	23613
	utg000078l	chromosomal	18064
	utg000079l	chromosomal	13912
	utg000080l	chromosomal	20491
	utg000081l	chromosomal	19414
	utg000082l	chromosomal	17026
	utg000084l	chromosomal	10403
	total chromosomal length		5162355
	utg000015l	plasmidic	289749
	utg000031c	plasmidic	85333
	utg000052l	plasmidic	89214
	utg000083l	plasmidic	9793
	utg000065l	undefined	27749
	utg000076l	undefined	14407
	utg000049l	none	20985
	utg000051l	none	21105
NCTC9646	utg000001l	chromosomal	1824786
	utg000002l	chromosomal	1185729
	utg000004l	chromosomal	918891
	utg000005l	chromosomal	1259981
	utg000006l	chromosomal	169448
	utg000007l	chromosomal	22892
	utg000008l	chromosomal	205862
	utg000009l	chromosomal	61801
	utg000010l	chromosomal	105678
	total chromosomal length		5755068
	utg000003l	plasmidic	181111
	utg000011l	plasmidic	38663
	utg000012l	plasmidic	16804
	utg000015l	plasmidic	26818
	utg000016l	plasmidic	6887
	utg000014l	undefined	37740
	utg000013c	none	2141
NCTC9695	utg000001l	chromosomal	4677804
	total chromosomal length		4677804
	utg000002c	none	3416

Table A.7: `Miniasm` contigs classification per dataset. Total length of chromosomal contigs is given.

A.6 Assembly length

In Table A.8 we report the sum of lengths of all contigs in assemblies computed by `Miniasm`, `Canu`, and the assemblies produced by NCTC.

Dataset	Canu	Miniasm	NCTC
NCTC10006	5297110	5586964	5285365
NCTC10332	6353743	6510723	6316979
NCTC10444	5358052	5575543	5295042
NCTC10702	3140906	3113949	3044394
NCTC10766	5704549	5981360	5662808
NCTC10794	2323585	2220964	2164041
NCTC10988	3242798	3352903	3309451
NCTC11126	4887988	5192073	4875981
NCTC11343	6102368	6167977	5984896
NCTC11360	2189222	2060590	2078787
NCTC11435	4452377	4647620	4443087
NCTC11800	4482172	4760149	4461490
NCTC11872	1881379	1962919	1879445
NCTC12123	4889243	5041564	4785686
NCTC12126	5037863	5243635	5015169
NCTC12131	4799906	4967498	4743059
NCTC12132	3360769	3520932	3326136
NCTC12146	5648936	5930232	5621322
NCTC12694	4667053	2787030	4439218
NCTC12841	1998859	2025517	*
NCTC12993	5339609	5613096	5287156
NCTC12998	5754007	6052839	5723058
NCTC13095	6018682	6239133	5962730
NCTC13125	5868476	5966557	3814513
NCTC13348	5133106	5242304	5042742
NCTC13463	5298325	5481195	5268825
NCTC13543	5834577	6012216	5822755
NCTC4672	2113924	1980239	1942171
NCTC5050	5360294	5747419	5342107
NCTC5053	5838871	6003270	5769583
NCTC5055	5618297	5578757	4924715
NCTC7152	5136487	5429453	5086417
NCTC7922	5497660	5572395	5367566
NCTC8179	5979700	5913131	5715723
NCTC8500	4733397	4886947	4709652
NCTC8684	4936541	5023386	4860337
NCTC9075	5336449	5539086	5322538
NCTC9078	5458474	5609092	5430557
NCTC9098	5385311	5641175	5331923
NCTC9111	5942101	6011787	5859950
NCTC9112	5557088	5907239	5468741
NCTC9184	2541470	351158	679813
NCTC9645	5930294	5720690	5852841
NCTC9646	5874126	6065232	*
NCTC9695	4792855	4681220	4738566

Table A.8: Total length of chromosomal contigs for Canu and Miniasm assemblers, total length of all contigs for NCTC on the 45 NCTC datasets studied. "*" means that NCTC assembly is not available.

A.7 Detailed assembly results

Supplementary *T. roseus* figures

Total number of reads	11592
Total length	104,608,900bp
Longest reads	46,221bp
Shortest reads	4bp
Mean Length	9,024bp
Median Length	6,978bp
N10	364 reads
N50	2586 reads
N90	7236 reads
L10	24,917bp
L50	14,590bp
L90	4,550bp

Table A.9: Some statistics about reads produced by LongISLND for *T. roseus* synthetic dataset.

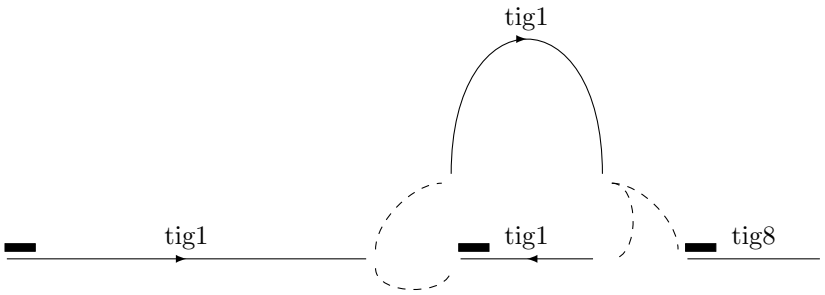


Figure A.2: Canu *T. roseus* tig1 reconstruction. Plain lines denote path without branches in the SG (the one shown Figure 4.15b). Boxes denote reads at contig extremities. Dashed lines denote overlaps between reads (and then contig extremities). Arrows show the path used by Canu to build tig1: it goes through the "loop" before going back.

A.8 Supplementary NCTC figures

NCTC12123

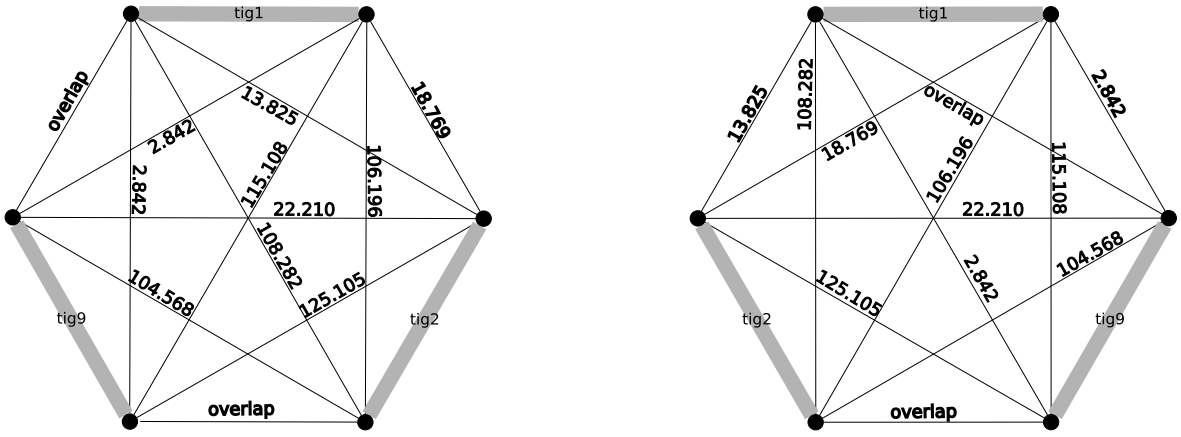


Figure A.3: Shortest paths in AAG. Both scenarii (paths that follow edges with bold label) use the edge of weight 7.178, the only remaining ambiguity concerns the order of tig1 against the pair tig2/tig9. 'overlap' indicate than our pipeline found an overlap between the contig extremities. The left scenario has a weight of 29.379 ($22.201 + 7.178$) while the right one has a weight of 30.736 ($17.209 + 7.178 + 6.349$).

NCTC5050

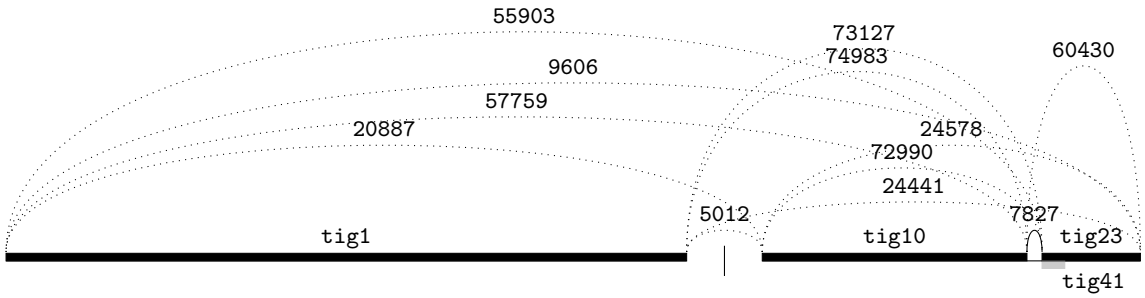


Figure A.4: NCTC5050 contigs mapped against NCTC reference for ordering. Paths are shown along with their number of bases as labels. The NCTC assembly consists of two contigs, hence the relative order of tig1 and tig10/tig23 cannot be inferred (vertical line in the figure). We observed that a portion of tig1 is inverted with respect to the NCTC assembly, with no impact on the path analysis as this putative misassembly does not involve an extremity of the contig.

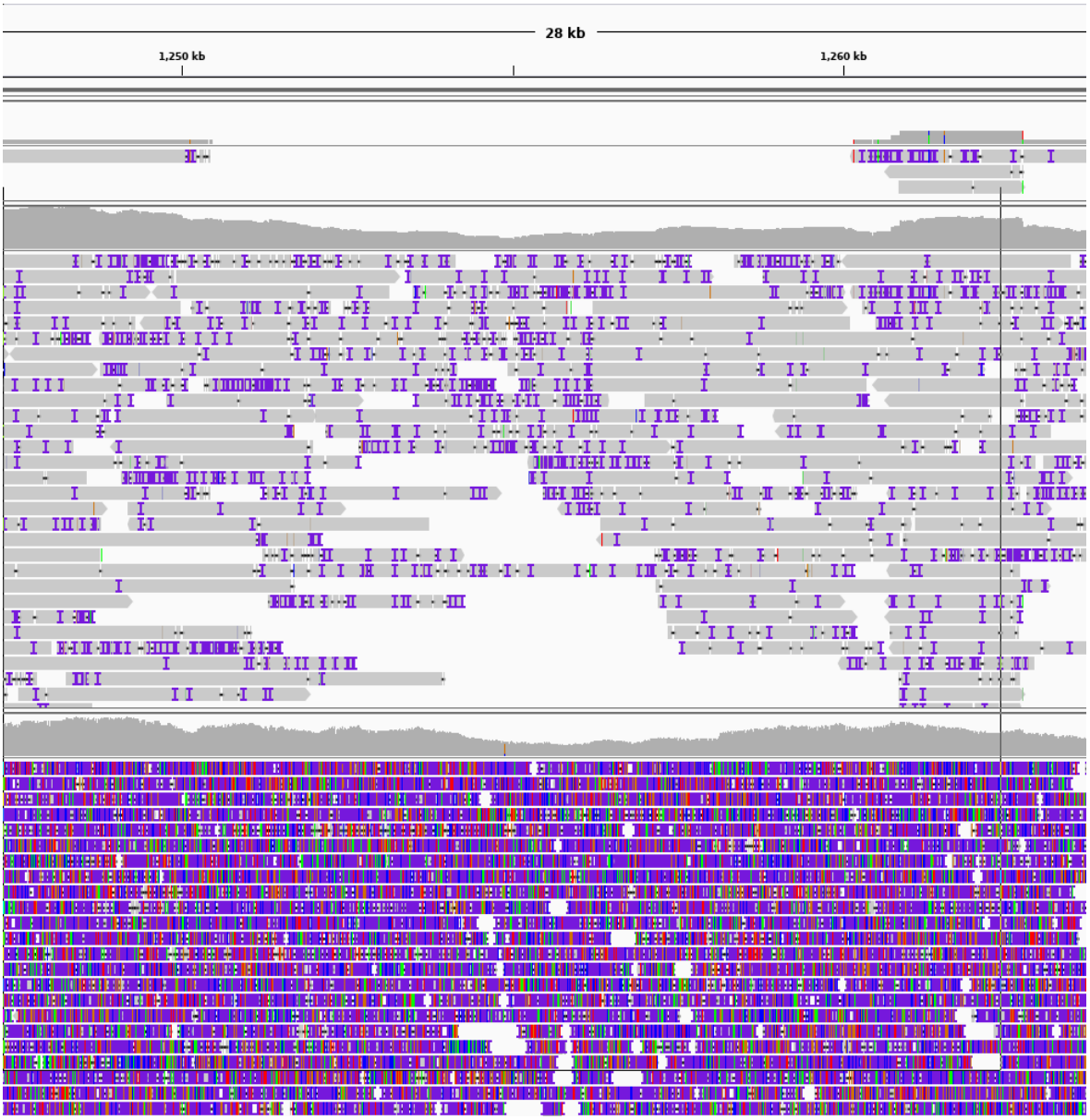


Figure A.5: IGV view of NCTC5050 mapping of `Canu` contig against NCTC contig, in junction between tig10 (first track at left) and tig23 (first track at right), tig41 are mapped on begin of tig23 in forward and reverse. The second track represent the mapping of `Canu` corrected read, the third track represent the raw reads. Above each this track we can observe the coverage curve and drop of this curve between the tig10 and tig23, for corrected read is around 50x coverage before junction, equal to 15x at minimal, and less than 40x after junction, this value are 90x, 25x and 40x for raw read. In addition we can observe more error in corrected read on this drop of coverage.

A.9 YACRD: Yet Another Chimeric Read Detector

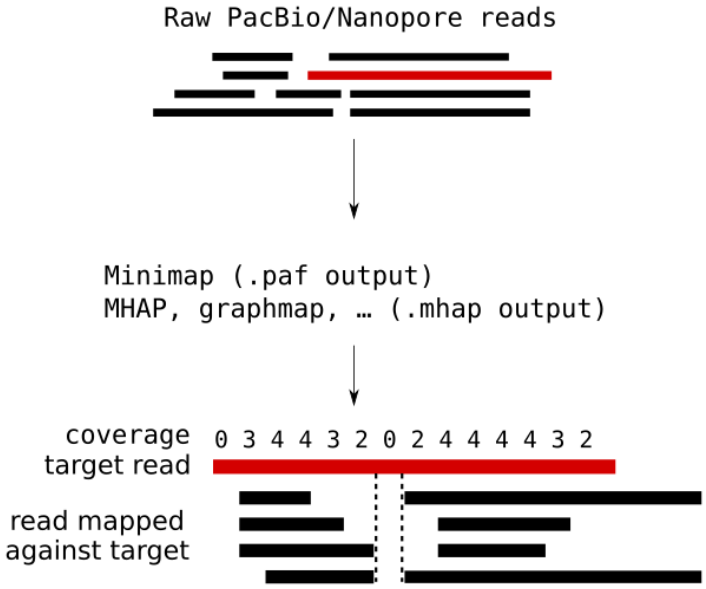


Figure A.6: YACRD (manuscript in preparation) detects chimeric regions present in the read dataset. To detect such regions, YACRD takes as input the output of an overlapper (both PAF and MHAP format are accepted). For each read in the dataset, YACRD computes positional coverage values based on the overlaps with that read. If there is a drop of coverage, the corresponding read is marked as 'chimeric'.

Appendix B

`yacrd` and `fpa`

B.1 Datasets

This section provides metrics about each dataset. The *E. coli* original dataset had large coverage (> 200x) so we subsampled it dataset with seqtk[1] down to target 50x.

	C. elegans	*D. melanogaster*	*H. sapiens*	*E. coli* Nanopore	*E. coli* Pacbio
|sequences|	740774	1327569	1075867	25469	37404
Shortest sequence (b)	35	5	30	152	35
Mean Length (Kb)	10958	6827	6744	10110	6894
Median Length (Kb)	9822	4568	5089	5515	6672
N50 (Kb)	16572	11853	10568	20073	9064
N90 (Kb)	6502	3533	3537	4701	4218
L10 (sequences)	27013	29049	21703	400	1354
L50 (sequences)	191637	243356	220369	3807	10502
L90 (sequences)	480553	779045	652055	13969	25787
Coverage	81x	63x	29x	49x	49x
Accession	*	SRR6702603	PRJEB23027	SRR8494911	SRR8494940
Publication			[35]	[61]	[61]

Table B.1: Information and metrics about the dataset using in our evaluation of `yacrd` and `fpa`. * The *C. elegans* dataset come from Pacbio DevNet https://github.com/PacificBiosciences/DevNet/wiki/C.-elegans-data-set.

[1] https://github.com/lh3/seqtk

B.2 Repeatability information

B.2.1 DASCRUBBER

DASCRUBBER commit number 0e90524 was used, and a custom pipeline was built using DASCRUBBBER-wrapper[2] as an inspiration, as well as recommendations from the authors: https://github.com/thegenemyers/DASCRUBBER/issues/7 and https://github.com/thegenemyers/DASCRUBBER/issues/20. See below (section B.2.11) for the URL of the custom pipeline.

B.2.2 miniscrub

We use version of commit 3d11d3e. We did not run miniscrub in GPU mode so we followed te authors instructions for installation and run https://bitbucket.org/berkeleylab/jgi-miniscrub/.

B.2.3 yacrd

We use version 0.5.1.

B.2.4 fpa

We use version 0.5.

B.2.5 BWA

We use version 0.7.17-r1188.

B.2.6 Minimap2

We use version 2.16-r922.

B.2.7 Miniasm

We use version 0.3-r179

B.2.8 wtdbg2

We use version 2.3.

B.2.9 QUAST

We use version v5.0.2.

[2] https://github.com/rrwick/DASCRUBBER-wrapper

B.2.10 Porechop

We use version 0.2.3_seqan2.1.1

B.2.11 Script and reproduction of analysis

All information to repeat our analysis can be found at this address

https://gitlab.inria.fr/pmarijon/yacrd-and-fpa-upstream-tools-for-lr-genome-assembly

B.3 yacrd parameter optimisation

yacrd is very dependent on the mechanism used to find common regions between reads. We rely on Minimap2 for this task. Minimap2 is based on short sequence seeds to find common regions between reads. In all-against-all alignment, it takes as parameter a distance between two seeds (-g, default: 10,000 bases). In yacrd we assume that regions with low seed coverage have low quality, and therefore need to be scrubbed. Yet with the default seed distance, it may happen that Minimap2 finds two consecutive seeds that correspond to two "good" read regions separated by one "bad" read region. Therefore this parameter needs to be tuned.

Another important parameter is the read coverage threshold to consider that a read region is of sufficient quality (yacrd parameter -c).

We have changed these two parameters as follows: i) the maximum distance between the two seeds from 50 to 2450 with a step of 100, ii) the minimum coverage before eliminating the region from 1 to 15 with a step of 1.

We evaluated the influence of these parameters on several metrics:

- Number of chimeric reads
- Number of reads
- Number of bases
- And in Miniasm and wtdbg2 assemblies,
  - NGA50
  - Total length
  - Number of contigs
  - Number of indels per 100 kpb
  - Number of mismatches per 100 kbp

We ran this evaluation on *H. sapiens*, *C. elegans* and *E. coli* Pacbio dataset. The raw data is available in:

- *H. sapiens* (ONT ultra-long R9.4): https://gitlab.inria.fr/pmarijon/yacrd-and-fpa-upstream-tools-fo raw/master/data/yacrd_parameter_test_h_sapiens_ont.csv
- *C. elegans* (Pacbio P6-C4): https://gitlab.inria.fr/pmarijon/yacrd-and-fpa-upstream-tools-for-lr-g blob/master/data/yacrd_parameter_test_c_elegans_pb.csv
- *E. coli* (Pacbio Sequel): https://gitlab.inria.fr/pmarijon/yacrd-and-fpa-upstream-tools-for-lr-geno blob/master/data/yacrd_parameter_test_e_coli_pb.csv

For *H. sapiens* Nanopore dataset we find that a value of 500 for the -g parameter and 4 for the -c parameter optimizes the number of contigs in Miniasm assembly and NGA50, and remains reasonable across the other metrics. We therefore recommend to use this value for Nanopore data and we used it in all of our results.

For *C. elegans* PacBio dataset P6-C4, using a similar reasoning, optimal values are different and are 800 for the `-g` parameter and 4 for the `-c` parameter.

For *E. coli* PacBio Sequel dataset, using similar reasoning, optimal values are different and are 5000 for the `-g` parameter and 3 `-c` parameter.

We therefore used the above values for all datasets obtained with the same sequencing technology.

B.4 Mapping of scrubbed reads

To compute quality metrics, for each dataset we mapped both scrubbed and raw reads against their respective reference genomes with `BWA` (we used `ont2d` preset for Nanopore reads, and `pacbio` preset for Pacbio reads). The mapping results were analyzed using a custom Python script[3] which reports the number of mapped reads, the sum of edit distances between each read and the matching reference sequence, the sum of positions of the genome mapped by a read, and the error rate.

To count the number of chimeric reads for each dataset, we remapped reads against each reference genome with `Minimap2` (we used `map-ont` preset for Nanopore reads, and `map-pb` preset for Pacbio reads). We analyzed the PAF (Pairwise Alignment Format) file outputted by `Minimap2` with a custom Python script[4]. This script parses a PAF file and associates to each read a list of pairs of starting/ending mapping positions. For each read, if two pairs of positions overlap in the corresponding list, they are merged. If, after merging, there remains more than one pair of positions, the read is marked as chimeric. To manage circular genomes we ignore reads with mapping positions near to the beginning/ending of the genome (within a distance of $\texttt{reference length} - 0.1 \times \texttt{reference length}$ from the beginning/ending).

To count the number of adapters in Nanopore reads we use `Porechop` [108] with out any specific parameter and we sum the number of adapters at start and end of reads, we ignore the count of middle adapters.

Table B.2 shows that scrubbing reduces the number of reads and the number of bases mapped against the reference, but the error rate is reduced too (at least 1% for `yacrd` and at least 2% for `DASCRUBBER`) and the number of chimeric reads was reduced by two or more.

		`BWA`				`Minimap2`	`Porechop`
Dataset	Scrubber	# reads mapped	Edit distance	Mapping length	Error rate	# chimeric reads	# adaptaters
C. elegans	raw	643138	903621479	6542507928	13.8115	71704	*n/a*
	yacrd	575517	758579062	5932958881	12.7858	15157	*n/a*
	dascrubber	576467	700895648	6128772910	11.4361	9285	*n/a*
D. melanogaster	raw	954622	1238009380	7353191408	16.8364	59864	891571
	yacrd	843483	968115342	6468730379	14.9661	28076	0
	dascrubber	792138	857944894	6543861920	13.1107	24826	246779
H. sapiens	raw	808709	1274720337	6053626797	21.0571	25888	947531
	yacrd	698139	929843201	4889850725	19.0158	5216	153255
	dascrubber	615789	813646386	4823555914	16.8682	1640	311007
E. coli Nanopore	raw	19873	36411589	232858822	15.6368	351	39596
	yacrd	18790	29819875	207863123	14.3459	64	12132
	dascrubber	18275	29216052	223383847	13.0789	50	6222
	miniscrub	24242	15740209	136642860	11.5192	58	36776
E. coli Pacbio	raw	31945	29162389	175640234	16.6035	7374	*n/a*
	yacrd	24728	22150527	146552898	15.1143	15157	*n/a*
	dascrubber	26883	20315636	158261992	12.8367	63	*n/a*
	miniscrub	10304	3050308	32249990	9.4583	37	*n/a*

Table B.2: Statistics of reads mapped to their respective reference, before and after scrubbing.

[3] https://gitlab.inria.fr/pmarijon/optimizing-early-steps-of-lr-assembly/blob/master/script/get_mapping_info.py

[4] https://gitlab.inria.fr/pmarijon/optimizing-early-steps-of-lr-assembly/blob/master/script/found_chimera.py

B.5 Quality of assembly

To assess the quality of assemblies with and without scrubbing, we ran both `Miniasm` and `wtdbg2` from scrubbed reads and raw reads with recommended parameters for each sequencing technology. After assembly we ran QUAST with parameter `--min-identity 80.0`.

Table B.3 shows a summary of outputted metrics for `Miniasm`. Scrubbing increases both the NGA50 and the length of the largest alignment. The size of the largest contig is often decreased but the contigs quality seems better as the number of misassemblies decreases. Finally the number of indels and mismatches per 100kb are quite stable. We thus observe that scrubbing improves assembly metrics, `yacrd` and `DASCRUBBER` having similar results, better than `miniscrub`.

Table B.4 shows a summary of outputted metrics for `wtdbg2`. Contrarily to `Miniasm`, NGA50 is not always improved by scrubbing. The size of the largest contig increases while the number of misassemblies decreases. This could be interpreted as a better assembly. Regarding these two metrics, `yacrd` has better results than `DASCRUBBER`.

Dataset	Scrubber	#contigs	NGA50	Largest contig	Largest alignment	Asm/Ref length	Indels per 100kb	Mismatches per 100kb	# misassemblies
C. elegans (Pacbio P6-C4)	yacrd	0.68	1.02	1.48	1.1	1.1	0.96	0.94	0.72
	dascrubber	0.58	1.26	0.97	1.55	1.08	0.94	0.9	0.54
D. melanogaster (ONT Minion)	yacrd	0.8	1.57	0.64	1.27	0.93	0.96	0.96	0.65
	dascrubber	0.84	2.41	1.45	2.48	0.96	0.96	0.94	0.83
H. sapiens (ONT ultra-long R9.4)	yacrd	2.14	4.72	0.38	5.19	1.41	0.97	0.95	0.25
	dascrubber	2.78	4.26	0.32	4.48	1.36	0.97	0.95	0.12
E. coli (ONT Minion)	yacrd	1	2.6	1	2.43	0.99	0.96	0.95	0.6
	dascrubber	1	2.65	1	2.48	0.99	0.97	0.96	0.6
	miniscrub	9	0.46	0.62	1.6	0.98	0.83	0.77	0.8
E. coli (Pacbio Sequel)	yacrd	1	1.96	0.96	1	1.02	0.99	0.99	0.63
	dascrubber	0.75	2.73	2.55	2.36	1.02	0.99	1.01	0.36

(a) Ratio of assembly metrics after scrubbing on assembly without scrubbing. Column Asm/Ref length report the total length of assembly against reference length, not against raw assembly length.

Dataset	Scrubber	#contigs	NGA50	Largest contig	Largest alignment	Total length	Indels per 100kb	Mismatches per 100kb	# misassemblies
C. elegans (Pacbio P6-C4)	raw	226	432112	5422030	1231264	114194187	7842.91	1944.78	1396
	yacrd	154	440776	8039734	1362861	110987109	7587.54	1827.39	1015
	dascrubber	131	544677	5262439	1907915	108636024	7405.35	1744.85	754
D. melanogaster (ONT Minion)	raw	423	423007	8745435	2396453	138733599	5789.82	4233.35	2126
	yacrd	339	664130	5559421	3053469	134302689	5587.09	4044.89	1375
	dascrubber	357	1018097	12708694	5953687	137569022	5537.66	3988.95	1765
H. sapiens (ONT ultra-long R9.4)	raw	184	96225	15987693	857015	202082384	6554.02	4089.56	1745
	yacrd	394	453748	6008000	4444926	203039148	6366.5	3891.98	432
	dascrubber	512	410370	5041373	3837755	195781855	6377.04	3887.84	209
E. coli (ONT Minion)	raw	1	1450762	5147604	1553466	5147604	5279.79	4341.81	5
	yacrd	1	3775907	5161073	3775907	5161073	5083.69	4104.31	3
	dascrubber	1	3850663	5168753	3850663	5168753	5113.64	4160.78	3
	miniscrub	9	670066	3172759	2478579	5136537	4382.29	3337.49	4
E. coli (Pacbio Sequel)	raw	4	499610	1974889	1083557	5417095	8011.42	1856.96	11
	yacrd	4	983113	1910204	1089886	5345453	7974.73	1856.19	7
	dascrubber	3	1362738	5042223	2552164	5331569	7963.32	1870.92	4

(b) Exact value of assembly metrics without scrubbing and with scrubbing

Table B.3: `Miniasm` assembly statistics.

Dataset	Scrubber	#contigs	NGA50	Largest contig	Largest alignment	Asm/Ref length	Indels per 100kb	Mismatches per 100kb	# mis-assemblies
C. elegans (Pacbio P6-C4)	yacrd	0.87	1.05	1.09	1	1.06	0.93	0.93	0.93
	dascrubber	0.72	1.02	1.04	1.11	0.98	0.9	0.7	0.35
D. melanogaster (ONT Minion)	yacrd	0.51	1.02	1.09	0.98	0.93	0.84	0.57	0.43
	dascrubber	0.61	0.87	0.83	0.98	0.93	0.85	0.62	0.41
H. sapiens (ONT ultra-long R9.4)	yacrd	0.6	0.98	11.68	1	0.97	0.94	0.86	0.44
	dascrubber	0.61	0.36	2.97	0.49	0.92	0.9	0.82	0.13
E. coli (ONT Minion)	yacrd	0.56	1.57	1.7	1.7	1.02	1.01	1	0
	dascrubber	1.11	0.81	0.64	0.57	0.87	1.06	1.18	1.5
	miniscrub	1	1.65	1.13	1.13	0.95	1.2	1.77	2
E. coli (Pacbio RS II)	yacrd	0.27	5.49	1.65	2.42	0.98	1.55	1.99	1.5
	dascrubber	0.64	1.14	0.43	0.67	1.01	1.24	1.46	0.75

(a) Ratio of assembly metrics after scrubbing on assembly without scrubbing. Column Asm/Ref length report the total length of assembly against reference length, not against raw assembly length.

Dataset	Scrubber	#contigs	NGA50	Largest contig	Largest alignment	Total length	Indels per 100kb	Mismatches per 100kb	# mis-assemblies
C. elegans (Pacbio P6-C4)	raw	139	565278	6301328	1880328	106873707	212.25	114.82	1396
	yacrd	122	593039	6919398	1880831	106276350	106,89	198,35	577
	dascrubber	100	578041	6577520	2084274	105265557	191.21	79.93	485
D. melanogaster (ONT Minion)	raw	945	1274655	22883959	5747639	144439108	1589.69	523.13	3938
	yacrd	484	1305125	24923636	5624012	135024912	1331.34	298.73	1675
	dascrubber	578	1114519	18994352	5625082	134142906	1348.97	324.48	1633
H. sapiens (ONT ultra-long R9.4)	raw	810	1513450	2435917	9247318	217462699	3588.91	368.93	1316
	yacrd	485	1482513	28462688	9268500	210552669	3370.08	318.69	582
	dascrubber	496	545902	7234785	4524362	200220997	3224.69	302.44	177
E. coli (ONT Minion)	raw	9	678871	1434432	1432545	5045762	767.87	156.21	2
	yacrd	5	1068201	2435917	2434921	5133519	778.16	155.46	0
	dascrubber	10	546569	917645	821696	4395460	817.43	184.84	3
	miniscrub	9	1117217	1621361	1619652	4773046	924.23	275.84	4
E. coli (Pacbio RS II)	raw	11	583235	2474045	1323293	5021940	170.49	46.03	4
	yacrd	3	3207692	4100960	3207692	5134707	264,26	91,77	3
	dascrubber	7	664896	1075736	892884	5093533	211.5	67.38	3

(b) Exact value of assembly metrics without scrubbing and with scrubbing

Table B.4: `wtdbg2` assembly statistics.

B.6 fpa

To evaluate `fpa`, we ran two different pipelines. The first one uses directly `Miniasm` without `fpa` and with recommended parameters. The second one runs `fpa` to filter out reads (`Minimap2` output is piped to `fpa` directly) before running `Miniasm` on filtered reads with recommended parameters. Using `fpa` we removed `internal match` and overlap shorter than 2000 (options `drop -i -l 2000`). This sort of overlap is ignored by `Miniasm` during the assembly step but is used during the read filtering step.

Table B.5 shows the impact of using `fpa` on time, memory and assembly metrics. Using `fpa` decreases both disk usage and total computation time of downstream analysis while having no impact or a positive one on assembly metrics. Usage of `fpa` does not radically affect mapping wall-clock time and memory usage, but it reduces by 13% to 67% the memory usage and CPU time of the assembly step (the computation time of `fpa` is included in the mapping time). Moreover the size of the PAF file produced by `Minimap2` is reduced by 40% to 79 %.

	Dataset	*C. elegans*		*D. melanogaster*		*H. sapiens* chr 1	
	Pipeline	w/o `fpa`	`fpa`	w/o `fpa`	`fpa`	w/o `fpa`	`fpa`
Time (s)	Mapping	3296	3247	3510	3659	1570	1558
	Assembly	297	139	782	186	103	50
	Total	3593	3386	4292	3845	1673	1608
Memory	Mapping (GB)	51	51	53	54	41	40
	Assembly (Mbp)	4788	2594	13836	5335	1797	587
	PAF size	32G	9.5G	54G	11G	8.9G	3.2G
Assembly	# contigs	168	150	423	381	184	216
	NGA50	407821	438055	423007	455307	96225	106259
	# misassemblies	1212	1149	2126	1840	1745	1502
	length	112248122	111641079	138733599	136623341	202082384	198386315
per 100kb	# mismatches	1893.44	1854.95	4233.35	4190.43	4089.56	4065.95
	# indels	7700.42	7628.39	5789.82	5742.05	6554.02	6534.92

	Dataset	*E. coli* Nanopore		*E. coli* Pacbio	
	Pipeline	w/o `fpa`	`fpa`	w/o `fpa`	`fpa`
Time (s)	Mapping	26	29	23	24
	Assembly	4	2	2	1
	Total	30	31	25	25
Memory	Mapping (GB)	3	3	4	4
	Assembly (Mbp)	52	45	33	22
	PAF size	141M	82M	85M	38M
Assembly	# contigs	5	5	8	9
	NGA50	1450762	1246808	562741	292111
	# misassemblies	5	5	8	9
	length	5147604	5283927	5394119	5395896
per 100kb	# mismatches	4341.81	4425.24	1862.72	1841.66
	# indels	5279.79	5376.03	7968.63	7945.11

Table B.5: Impact of `fpa` on assembly using `Miniasm`.

B.7 Combination of yacrd and fpa

To evaluate the effect of running both `yacrd` and `fpa`, we ran two different pipelines. The first one uses a standard `Miniasm` pipeline (called 'basic'): `Minimap2` plus `Miniasm` with recommended parameters. The second one (called 'extended') runs `yacrd` with best parameters for each dataset, then `Minimap2` with recommend parameter on scrubbed reads and pipes the results in `fpa` to filter out `internal matches` and overlaps shorter than 2000 (option `drop -i -l 2000`), and finally runs `Miniasm` on scrubbed reads with filtered overlap.

Table B.6 shows how the integration of both `yacrd` and `fpa` in `Miniasm` pipeline ('extended' row) compares against standard `Miniasm` ('simple' row). Each pipeline is based on `Minimap2` so their memory usages are equivalent. The extended pipeline takes twice more time because `Minimap2` is run twice (once for `yacrd` and once for `Miniasm`). `Minimap2` is a time bottleneck in both pipelines.

The extended pipeline improves the quality of assemblies, in terms of NGA50, number of indels and mismatches per 100 kbp, and misassemblies. It also decreases the number of contigs while keeping the total length of assemblies similar.

Dataset	*C. elegans*		*D. melanogaster*		*H. sapiens*	
Pipeline	simple	extended	simple	extended	simple	extended
# contigs	226	171	423	345	184	367
NGA50	432112	451479	423007	715276	96225	488573
Largest contig	5422030	4224860	8745435	5559611	15987693	6875897
Largest alignment	1231264	1527213	2396453	3053469	857015	4444801
Total length	114194187	110177189	138733599	134443509	202082384	202405973
# indels per 100 kbp	7842.91	7380.12	5789.82	5593.09	6554.02	6359.25
# mismatches per 100 kbp	1944.78	1720.16	4233.35	4052.42	4089.56	3884.23
# misassemblies	1396	907	2126	1412	1745	363

Dataset	*E. coli* Nanopore		*E. coli* Pacbio	
Pipeline	simple	extended	simple	extended
# contigs	1	1	4	3
NGA50	1450762	3775889	499610	1271550
Largest contig	5147604	5186180	1974889	4960107
Largest alignment	1553466	3775889	1083557	1465922
Total length	5147604	5186180	5417095	5355278
# indels per 100 kbp	5279.79	5097.12	8011.42	7969.99
# mismatches per 100 kbp	4341.81	4113.01	1856.96	1844.42
# misassemblies	5	3	11	8

Table B.6: Impact of `yacrd` and `fpa` on assembly using `Miniasm`. Simple match to basic `Miniasm` pipeline and extend match to version with `yacrd` and `fpa`.

Abstract

The sequencing of genetic information provides better understanding for a large number of biological phenomena: e.g. genetic diseases, speciation events, fundamental mechanisms of cell function. Sequencing techniques have considerably evolved since the Sanger method (1977). Nowadays third-generation sequencing technologies greatly reduce the costs of sequencing complete genomes. They produce longer reads (sequence fragments), but require the design of specific assembly tools that take into account the high error rates in the produced fragments.

The study of methods used by third-generation read assembly pipelines has revealed that improvements in assembly were possible without modifying assembly tools themselves. Some improvements are thus proposed in this thesis work, and were implemented through publicly available tools. `yacrd` and `fpa` pre-process the set of reads prior to assembly, in order to improve efficiency and quality of the assembly process. `KNOT` combines information from both the input reads and an assembly, in order to provide insights on how to improve the contiguity of an assembly.

Keywords: Genome assembly, Third generation sequencing, Assembly graphs

Résumé

Le séquençage de l'information génétique a permis de mieux comprendre un grande nombre de phénomènes biologiques, maladies génétiques, évènements de spéciations, mécanismes fondamentaux du fonctionnement de nos cellules. Les techniques de séquençage ont beaucoup évolué depuis la méthode de Sanger (1977). De nos jours, les technologies de séquençage de troisième génération permettent le séquençage d'un génome complet à moindre coût, produisent des lectures (fragments de genomes) plus longs, mais nécessitent la création d'outils d'assemblage spécifiques pour tenir compte d'un taux d'erreur élevé dans les lectures produites.

L'étude des méthodes utilisées par les outils d'assemblage de lectures de troisième génération a permis d'observer que des améliorations des assemblages étaient possibles sans toutefois modifier les outils eux-mêmes. Certaines améliorations sont proposées dans ce travail de thèse, et sont mises en œuvre à travers des outils proposés à la communauté. `yacrd` et `fpa` interviennent en amont de l'assemblage en lui-même pour améliorer l'ensemble des lectures données en entrée à un assembleur. `KNOT` analyse et combine le résultat d'un assemblage avec les données brutes, pour donner des pistes permettant d'améliorer l'assemblage final.

Mots clef: Assemblage de génomes, Troisième generation de séquençage, Graphes d'assemblage