Title

Algorithms for long-read assembly

Author

Kolmogorov, Mikhail

Publication Date

2019

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA SAN DIEGO

Algorithms for long-read assembly

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Mikhail A. Kolmogorov

Committee in charge:

Professor Pavel Pevzner, Chair
Professor Vineet Bafna
Professor Vikas Bansal
Professor Melissa Gymrek
Professor Glenn Tesler

2019

The dissertation of Mikhail A. Kolmogorov is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

____

____

____

____

____

Chair

University of California San Diego

2019

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank Pavel Pevzner for supervising me through my Ph.D. program. Pavel is an outstanding scientist with great vision of the field and mathematical attention to details. Pavel is also a unique mentor, who dedicates a lot of time and resources to all of his students. Pavel's support was invaluable for me growing up as a scientist.

I also grateful my committee members for their great support and crucial feedback.

I was fortunate to work with the great colleagues at the Department of Computer Science and Engineering: Yu Lin, Jeffrey Yuan, Anton Bankevich, Anrey Bzikadze, Yana Safonova, Viraj Deshpande and many others.

I also thank Son Pham, who was my master program advisor in St. Petersburg, Russia, and who encouraged my to apply to grad schools in the U.S.

Chapter 1, in full, is a reprint of the material as it appears in Y. Lin, J. Yuan, M. Kolmogorov, M. Shen, M. Chaisson and P. Pevzner, "Assembly of long error-prone reads using de Bruijn graphs", Proceedings of the National Academy of Sciences (2016). The dissertation author was the primary developer of the ABruijn project and one of the three lead authors of this paper.

Chapter 2, in full, is a reprint of the material as it appears in M. Kolmogorov, J. Yuan, Y. Lin and P. Pevzner, "Assembly of long error-prone reads using repeat graphs", Nature Biotechnology (2019). The dissertation author was the primary developer of Flye and first author of this paper.

Chapter 3, in full, has been submitted for a journal publication, and also appears as a bioRxiv pre-print: M. Kolmogorov, M. Rayko, J. Yuan, E. Polevikov and P. Pevzner, "metaFlye: scalable and accurate long-read metagenome assembler", bioRxiv (2019) doi: https://doi.org/10.1101/637637. The dissertation author was the primary developer of metaFlye and first author of this paper.

VITA

2008-2012	B. S. in Applied Mathematics, ITMO University, St. Petersburg, Russia
2012-2014	M. S. in Bioinformatics, St. Petersburg Academic University, Russia
2014-2019	Ph. D. in Computer Science, University of California, San Diego

PUBLICATIONS

Mikhail Kolmogorov, Mikhail Rayko, Jeffrey Yuan, Evgeny Polevikov and Pavel Pevzner. metaFlye: scalable and accurate long-read metagenome assembler. *bioRxiv*, 2019, doi: https://doi.org/10.1101/637637.

Evgeny Polevikov and Mikhail Kolmogorov. Synteny paths for assembly graphs comparison. *Workshop in Algorithmic Bioinformatics*, 2019.

Mikhail Kolmogorov, Jeffrey Yuan, Yu Lin and Pavel Pevzner. Assembly of long error-prone reads using repeat graphs. *Nature Biotechnology*, 37(5):540-546, 2019.

Alla Mikheenko and Mikhail Kolmogorov. Assembly Graph Browser: interactive visualization of assembly graphs. *Bioinformatics*, 35(18):3476-3478, 2019

Mikhail Kolmogorov, Joel Armstrong, Brian J. Raney, Ian Streeter, Matthew Dunn, Fengtang Yang, Duncan Odom, Paul Flicek, Thomas Keane, David Thybert, Benedict Paten and Son Pham. Chromosome assembly of large and complex genomes using multiple references. *Genome Research*, 28(11):1720-1732, 2018.

Alex Bishara, Eli L Moss, Mikhail Kolmogorov, Alma Parada, Ziming Weng, Arend Sidow, Anne E Dekas, Serafim Batzoglou, Ami S Bhatt. High-quality genome sequences of uncultured microbes by assembly of read clouds. *Nature Biotechnology*, 36:1067-1075, 2018.

Jingtao Lilue, Anthony G Doran, Ian T Fiddes, Monica Abrudan, Joel Armstrong, ... Mikhail Kolmogorov ... David Thybert, James Torrance, Kim Wong, Jonathan Wood, Binnaz Yalcin, Fengtang Yang, David J Adams, Benedict Paten, Thomas M Keane. Sixteen diverse laboratory mouse reference genomes define strain-specific haplotypes and novel functional loci. *Nature Genetics*, 50:1574-1583, 2018.

David Thybert, Maa Roller, Fbio CP Navarro, Ian Fiddes, Ian Streeter, Christine Feig, David Martin-Galvez, Mikhail Kolmogorov, ... Thomas M Keane, Duncan T Odom, Paul Flicek. Repeat associated mechanisms of genome evolution and function revealed by the Mus caroli and Mus pahari genomes. *Genome Research*, 28(4):448-459, 2018.

Mikhail Kolmogorov, Eamonn Kennedy, Zhuxin Dong, Gregory Timp and Pavel Pevzner. Single-Molecule Protein Identification by Sub-Nanopore Sensors. *PloS Computational Biology*, 13(5):e1005356, 2017.

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

Mikhail Kolmogorov, Xiaowen Liu and Pavel Pevzner. SpectroGene: a tool for proteogenomic annotations using top-down spectra. *Journal of Proteome Research*, 15(1):144-151, 2015.

Mikhail Kolmogorov, Brian Raney, Benedict Paten, Son Pham. Ragout – a reference-assisted assembly tool for bacterial genomes. *Bioinformatics*, 30(12):i302-i309, 2014.

Ilya Minkin, Anand Patel, Mikhail Kolmogorov, Nikolay Vyahhi, Son Pham. Sibelia: a scalable and comprehensive synteny block generation tool for closely related microbial genomes. *Workshop in Algorihmic Bioinformatics*, 215-229, 2013.

ABSTRACT OF THE DISSERTATION

Algorithms for long-read assembly

by

Mikhail A. Kolmogorov

Doctor of Philosophy in Computer Science

University of California San Diego, 2019

Professor Pavel Pevzner, Chair

The recently introduced long-read sequencing technologies (such as Pacific Biosciences or Oxford Nanopore) have substantially improved genome assemblies of many organisms, including the human reference genome. The technologies are, however, facing the challenge of high read errors. In this dissertation, we describe multiple algorithms for assembly and analysis of long-read sequencing data. First, we introduce the ABruijn algorithm for long-read assembly that bypasses the expensive read error-correction step by identifying reliable k-mers in reads. We then describe the Flye package, that combines ABruijn with a new repeat graph approach that accurately resolves the genomic structure. Finally, we extend Flye to the assembly of complex metagenomic communities using long reads.

Chapter 1

Assembly of long, error-prone reads using de Bruijn graphs

1.1 Abstract

The recent breakthroughs in assembling long error-prone reads were based on the overlap-layout-consensus (OLC) approach and did not utilize the strengths of the alternative de Bruijn graph approach to genome assembly. Moreover, these studies often assume that applications of the de Bruijn graph approach are limited to short and accurate reads and that the OLC approach is the only practical paradigm for assembling long error-prone reads. We show how to generalize de Bruijn graphs for assembling long error-prone reads and describe the ABruijn assembler, which combines the de Bruijn graph and the OLC approaches and results in accurate genome reconstructions.

1.2 Introduction

The key challenge to the success of single-molecule sequencing (SMS) technologies lies in the development of algorithms for assembling genomes from long but inaccurate reads. The pioneer in long reads technologies, Pacific Biosciences, now produces accurate assemblies from long error-prone reads [Berlin et al., 2015, Chin et al., 2013]. [Goodwin et al., 2015] and [Loman et al., 2015] demonstrated that high-quality assemblies can be obtained from even less-accurate Oxford Nanopore reads. Advances in assembly of long error-prone reads recently resulted in the accurate reconstructions of various genomes [Koren et al., 2013, Koren and Phillippy, 2015, Lam et al., 2015, Chaisson et al., 2015, Huddleston et al., 2014, Ummat and Bashir, 2014]. However, as illustrated in [Booher et al., 2015], the problem of assembling long error-prone reads is far from being resolved even in the case of relatively small bacterial genomes.

Previous studies of SMS assemblies were based on the overlap-layout-consensus (OLC) approach [Kececioglu and Myers, 1995] or a similar string graph approach [Myers, 2005], which require an all-against-all comparison of reads [Myers, 2014] and remain computationally challenging (see [Idury and Waterman, 1995, Li et al., 2012, Pevzner et al., 2001] for a discussion of the pros and cons of this approach). Moreover, there is an assumption that the de Bruijn graph approach, which has dominated genome assembly for the last decade, is inapplicable to long reads. This is a misunderstanding, because the de Bruijn graph approach, as well as its variation called the A-Bruijn graph approach, was developed to assemble rather long Sanger reads [Pevzner et al., 2004]. There is also a misunderstanding that the de Bruijn graph approach can only assemble highly accurate reads and fails when assembling long error-prone reads. Although this is true for the original de Bruijn graph approach to assembly [Idury and Waterman, 1995, Li et al., 2012, Pevzner et al., 2001], the A-Bruijn graph approach was originally designed to assemble inaccurate reads as long as any similarities between reads can be reliably identified. Moreover, A-Bruijn graphs have proven to be useful even for assembling mass spectra, which represent highly inaccurate fingerprints of amino acid sequences of peptides [Bandeira et al., 2007, 2008]. However, although A-Bruijn graphs have proven to be useful in assembling Sanger reads and mass spectra, the question of how to apply A-Bruijn graphs for assembling long error-prone reads remains open.

De Bruijn graphs are a key algorithmic technique in genome assembly [Idury and Waterman, 1995, Butler et al., 2008, Simpson et al., 2009, Zerbino and Birney, 2008, Bankevich et al., 2012]. In addition, de Bruijn graphs have been used for sequencing by hybridization [Pevzner, 1989], repeat classification [Pevzner et al., 2004], de novo protein sequencing [Bandeira et al., 2008], synteny block construction [Pham and Pevzner, 2010], genotyping [Iqbal et al., 2012], and Ig classification [Bonissone and Pevzner, 2015]. A-Bruijn graphs are even more general than de Bruijn graphs; for example, they include breakpoint graphs, the workhorse of genome-rearrangement studies [Lin et al., 2014].

However, as discussed in [Lin et al., 2014], the original definition of a de Bruijn graph is far from being optimal for the challenges posed by the assembly problem. Below, we describe the concept of an A-Bruijn graph, introduce the ABruijn assembler for long error-prone reads, and demonstrate that it generates accurate genome reconstructions.

1.3 Methods

1.3.1 The Key Idea of the ABruijn Algorithm

The Challenge of Assembling Long Error-Prone Reads. Given the high error rates of SMS technologies, accurate assembly of long repeats remains challenging. Also, frequent k-mers dramatically increase the number of candidate overlaps, thus, complicating the choice of the correct path in the overlap graph. A common solution is to mask highly repetitive k-mers as done in the Celera Assembler [Myers et al., 2000] and Falcon [Chin et al., 2016]. However, such masking may lead to losing some correct overlaps. Below we illustrate these challenges using the

Xanthomonas genomes as an example.

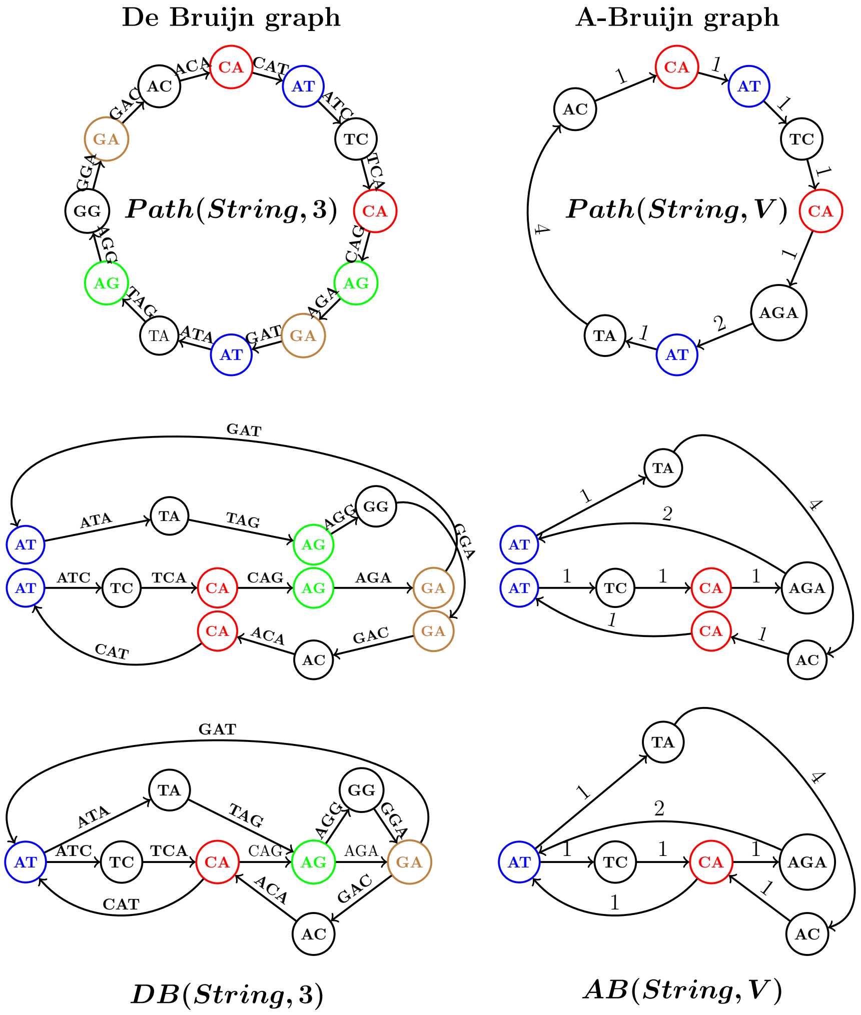
[Booher et al., 2015] recently sequenced various strains of the plant pathogen *Xanthomonas oryzae* and revealed the striking plasticity of transcription activator-like (tal) genes, which play a key role in *Xanthomonas* infections. Each tal gene encodes a TAL protein, which has a large domain formed by nearly identical TAL repeats. Because variations in tal genes and TAL repeats are important for understanding the pathogenicity of various *Xanthomonas* strains, massive sequencing of these strains is an important task that may enable the development of novel measures for plant disease control [Schornack et al., 2013, Doyle et al., 2013]. However, assembling *Xanthomonas* genomes using SMS reads (let alone, short reads) remains challenging.

Depending on the strain, *Xanthomonas* genomes may harbor over 20 tal genes with some tal genes encoding over 30 TAL repeats. Assembling *Xanthomonas* genomes is further complicated by the aggregation of various types of repeats into complex regions that may extend for over 30 kb in length. These repeats render *Xanthomonas* genomes nearly impossible to assemble using short reads. Moreover, as [Booher et al., 2015] described, existing SMS assemblers also fail to assemble *Xanthomonas* genomes. The challenge of finishing draft genomes assembled from SMS reads extends beyond *Xanthomonas* genomes (e.g., many genomes sequenced at the Centers for Disease Control are being finished using optical mapping [Williams et al., 2016]).

Another challenge is using SMS technologies to assemble metagenomics datasets with highly variable coverage across various bacterial genomes. Because the existing assemblers for long error-prone reads generate fragmented assemblies of bacterial communities, there are as yet no publications describing metagenomics applications of SMS technologies. Below we benchmark ABruijn and other state-of-the-art SMS assemblers on various genomes and the Bugula neritina metagenome.

From de Bruijn Graphs to A-Bruijn Graphs. In the A-Bruijn graph framework, the classical de Bruijn graph $DB(String,k)$ of a string $String$ is defined as follows. Let *Path(String,k)* be a path consisting of $|String|-k+1$ edges, where the i-th edge of this path is labeled by the

i-th k-mer in *String* and the i-th vertex of the path is labeled by the i-th $(k-1)$-mer in *String*. The de Bruijn graph *DB(String,k)* is formed by gluing together identically labeled vertices in *Path(String,k)* (Figure 1.1). Note that this somewhat unusual definition results in exactly the same de Bruijn graph as the standard definition (see [Compeau and Pevzner, 2015] for details).



De Bruijn graph

A-Bruijn graph

$Path(String,3)$

$Path(String,V)$

$DB(String,3)$

$AB(String,V)$

Figure 1.1: Constructing the de Bruijn graph (Left) and the A-Bruijn graph (Right) for a circular $String = CATCAGATAGGA$. (Left) From $Path(String,3)$ to $DB(String,3)$. (Right) From $Path(String,V)$ to $AB(String,V)$ for $V = CA, AT, TC, AGA, TA, AC$. The figure illustrates the process of bringing the vertices with the same label closer to each other (middle row) to eventually glue them into a single vertex (bottom row). Note that some symbols of *String* are not covered by strings in V. We assign integer $shift(v,w)$ to the edge (v,w) in this path to denote the difference between the positions of v and w in *String* (i.e., the number of symbols between the start of v and the start of w in *String*).

We now consider an arbitrary substring-free set of strings V (which we refer to as a set of solid strings), where no string in V is a substring of another one in V. The set V consists of words (of any length) and the new concept *Path(String,V)* is defined as a path through all words from V appearing in *String* (in order) as shown in Figure 1.1. Afterwards, we glue identically labeled vertices as before to construct the A-Bruijn graph *AB(String,V)* as shown in Figure 1.1. Clearly, *DB(String,k)* is identical to $AB(String, \Sigma_{k-1})$, where Σ_{k-1} stands for the set of all $(k-1)$-mers in alphabet Σ.

The definition of *AB(String,V)* generalizes to *AB(Reads,V)* by constructing a path for each read in the set *Reads* and further gluing all identically labeled vertices in all paths. Because the draft genome is spelled by a path in *AB(Reads,V)* [Pevzner et al., 2004], it seems that the only thing needed to apply the A-Bruijn graph concept to SMS reads is to select an appropriate set of solid strings V, to construct the graph *AB(Reads,V)*, to select an appropriate path in this graph as a draft genome, and to correct errors in the draft genome. Below we show how ABruijn addresses these tasks.

The Challenge of Selecting Solid Strings. Different approaches to selecting solid strings affect the complexity of the resulting A-Bruijn graph and may either enable further assembly using the A-Bruijn graph or make it impractical. For example, when the set of solid strings $V = \Sigma_{k-1}$ consists of all $(k-1)$-mers, $AB(Reads, \Sigma_{k-1})$ may be either too tangled (if k is small) or too fragmented (if k is large).

Although this is true for both short accurate reads and long error-prone reads, there is a key difference between these two technologies with respect to their resulting A-Bruijn graphs. In the case of Illumina reads, there exists a range of values k so that one can apply various graph simplification procedures (e.g., bubble and tip removal [Pevzner et al., 2004, Zerbino and Birney, 2008]) to enable further analysis of the resulting graph. However, these graph simplification procedures were developed for the case when the error rate in the reads does not exceed 1% and fail in the case of SMS reads where the error rate exceeds 10%.

An Outline of the ABruijn Algorithm. We classify a *k*-mer as genomic if it appears in the genome and nongenomic otherwise. Ideally, we would like to select a set of solid strings containing all genomic *k*-mers and no nongenomic *k*-mers.

Although the set of genomic *k*-mers occurring in the set of reads is unknown, we show how to identify a large set of predominantly genomic *k*-mers by selecting sufficiently frequent *k*-mers in reads. However, this is not sufficient for assembly, because some genomic *k*-mers are missing and some nongenomic k-mers are present in the constructed set of solid *k*-mers. Moreover, even if we were able to construct a very accurate set of genomic *k*-mers, the de Bruijn graph constructed on this set would be too tangled because typical values of *k* range from 15 to 25 (otherwise it is difficult to construct a good set of solid *k*-mers). Instead, we construct the A-Bruijn graph on the set of identified solid *k*-mers rather than the de Bruijn graph on all *k*-mers in reads. Although only a small fraction of the *k*-mers in each read are solid (and hence this is a very incomplete representation of reads), overlapping reads typically share many solid *k*-mers (compared with non-overlapping reads). Therefore, a rough estimate of the overlap between two reads can be obtained by finding the longest common subpath between the two read-paths using a fast dynamic programming algorithm. Hence, the A-Bruijn graph can function as an oracle, from which one can efficiently identify the overlaps of a given read with all other reads by considering all possible overlaps at once. The genome is assembled by repeatedly applying this procedure and borrowing the path extension paradigm from short read assemblers [Boisvert et al., 2012, Prjibelski et al., 2014, Vasilinetc et al., 2015].

Each assembler should minimize the number of misassemblies and the number of basecalling errors. The described approach minimizes the number of misassemblies but results in an inaccurate draft genome with many basecalling errors. We later describe an error-correction approach, which results in accurate genome reconstructions.

1.3.2 Selecting Solid Strings for Constructing A-Bruijn Graphs

We define the frequency of a k-mer as the number of times this k-mer appears in the reads and argue that frequent k-mers (for sufficiently large k) are good candidates for the set of solid strings. We define a (k,t)-mer as a k-mer that appears at least t times in the set of reads.

We classify a k-mer as unique (repeated) if it appears once (multiple times) in the genome. Figure 1.2 shows the histogram of the number of unique/repeated/nongenomic 15-mers with given frequencies for the Ecoli SMS dataset described in Results. As Figure 1.2 illustrates, the lion's share of 15-mers with frequencies at least t are genomic ($t = 7$ for the Ecoli dataset). To automatically select the parameter t, we compute the number of k-mers with frequencies exceeding t, and select a maximal t such that this number exceeds the estimated genome length. As Figure 1.2 illustrates, this selection results in a small number of nongenomic k-mers while capturing most genomic k-mers.

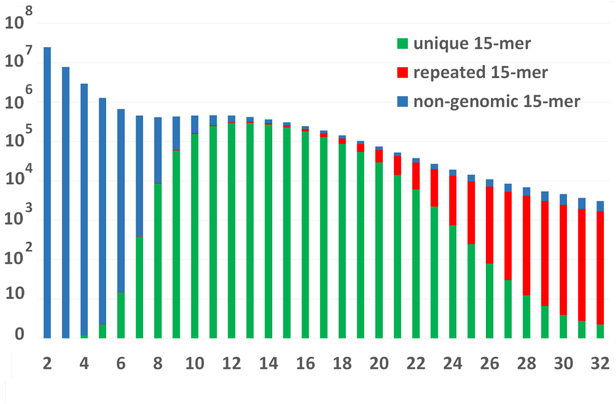


Figure 1.2: The histograms of the number of 15-mers with given frequencies for the Ecoli dataset from *Escherichia coli*. The bars for unique/repeated/nongenomic 15-mers for the *E. coli* genome are stacked and shown in green/red/blue according to their fractions. ABruijn automatically selects the parameter t and defines solid strings as all 15-mers with frequencies at least $t = 7$ for the Ecoli dataset. We found that increasing the automatically selected values of t by 1 results in equally accurate assemblies. There exist 4.1, 0.1, and 0.5 million (3.9, 0.1, and 0.3 million) unique, repeated, and nongenomic 15-mers, respectively, for Ecoli at $t = 7$ ($t = 8$). Although larger values of k (e.g., $k = 25$) also produce high-quality SMS assemblies, we found that selecting smaller rather than larger k results in slightly better performance.

1.3.3 Finding the Genomic Path in an A-Bruijn Graph

After constructing an A-Bruijn graph, one faces the problem of finding a path in this graph that corresponds to traversing the genome and then correcting errors in the sequence spelled by this path (this genomic path does not have to traverse all edges of the graph). Because the long reads are merely paths in the A-Bruijn graph, one can use the path extension paradigm [Boisvert et al., 2012, Prjibelski et al., 2014, Vasilinetc et al., 2015] to derive the genomic path from these (shorter) read-paths. exSPAnder [Prjibelski et al., 2014] is a module of the SPAdes assembler [Bankevich et al., 2012] that finds a genomic path in the assembly graph constructed from short reads based either on read-pair paths or read-paths, which are derived from SMS reads as in hybridSPAdes [Antipov et al., 2015]. Recent studies of bacterial plankton [Labonté et al., 2015], antibiotics resistance [Ashton et al., 2015], and genome rearrangements [Risse et al., 2015] demonstrated that hybridSPades works well even for coassembly with less-accurate nanopore reads. Below we sketch the HYBRIDSPADES algorithm [Antipov et al., 2015] and show how to modify the path extension paradigm to arrive at the ABruijn algorithm.

hybridSPAdes. hybridSPAdes uses SPAdes to construct the de Bruijn graph solely from short accurate reads and transforms it into an assembly graph by removing bubbles and tips [Bankevich et al., 2012]. It represents long error-prone reads as read-paths in the assembly graph and uses them for repeat resolution.

A set of paths in a directed graph (referred to as *Paths*) is consistent if the set of all edges in Paths forms a single directed path in the graph. We further refer to this path as *ConsensusPath(Paths)*. The intuition for the notion of the consistent (inconsistent) set of paths is that they are sampled from a single segment (multiple segments) of the genomic path in the assembly graph [Antipov et al., 2015].

A path P' in a weighted graph overlaps with a path P if a sufficiently long suffix of P (of total weight at least *minOverlap*) coincides with a prefix of P' and P does not contain the entire path P' as a subpath. Given a path P and a set of paths *Paths*, we define $Paths_{minOverlap}(P)$ as the set of all paths in *Paths* that overlap with *P*.

Our sketch of hybridSPAdes omits some details and deviates from the current implementation to make similarities with the A-Bruijn graph approach more apparent (e.g., it assumes that there are no chimeric reads and only shows an algorithm for constructing a single contig).

```
function HYBRIDSPADES(ShortReads, LongReads, k, minOverlap)
    construct the de Bruijn graph on k-mers from ShortReads
    transform the de Bruijn graph into the assembly graph
    ReadPaths ← the set of paths in the assembly graph corresponding to
       all reads from LongReads
    InitialPath ← an arbitrary read-path from ReadPaths
    GrowingPath ← InitialPath
    while forever do
        OverlapPaths ← $ReadPaths_{minOverlap}(GrowingPath)$
        if the set OverlapPaths is consistent then
            if CONSENSUSPATH(OverlapPaths) contains InitialPath then
                return string spelled by GrowingPath (as a complete genome)
            end if
            if CONSENSUSPATH(OverlapPaths) overlaps with GrowingPath then
                extend GrowingPath by CONSENSUSPATH(OverlapPaths)
            end if
        else
            return string spelled by GrowingPath (as one of the contigs)
        end if
    end while
end function
```

From hybridSPAdes to longSPAdes. Using the concept of the A-Bruijn graph, a similar approach can be applied to assembling long reads only. The pseudocode of longSPAdes differs from the pseudocode of hybridSPAdes by only the top three lines shown below:

```
function LONGSPADES(LongReads, k, t, minOverlap)
    construct the A-Bruijn graph on $(k,t)$-mers from LongReads
    transform the A-Bruijn graph into the assembly graph
end function
```

We note that longSPAdes constructs a path spelling out an error-prone draft genome that requires further error correction. However, error correction of a draft genome is faster than the error correction of individual reads before assembly in the OLC approach [Berlin et al., 2015, Chin et al., 2013, Goodwin et al., 2015, Loman et al., 2015].

Although hybridSPAdes and longSPAdes are similar, longSPAdes is more difficult to implement because bubbles in the A-Bruijn graph of error-prone long reads are more complex than bubbles in the de Bruijn graph of accurate short reads. As a result, the existing graph simplification algorithms fail to work for A-Bruijn graphs made from long error-prone reads. Although it is possible to modify the existing graph simplification procedures for long error-prone reads (to be described elsewhere), this paper focuses on a different approach that does not require graph simplification.

From longSPAdes to ABruijn. Instead of finding a genomic path in the simplified A-Bruijn graph, ABruijn attempts to find a corresponding genomic path in the original A-Bruijn graph. This approach leads to an algorithmic challenge: Although it is easy to decide whether two reads overlap given an assembly graph, it is not clear how to answer the same question in the context of the A-Bruijn graph. Note that although the ABruijn pseudocode below uses the same terms "overlapping" and "consistent" as longSPAdes, these notions are defined differently in the context of the A-Bruijn graph. The new notions (as well as parameters *jump* and *maxOverhang*) are described below.

The constructed path in the A-Bruijn graph spells out an error-prone draft genome (or one of the draft contigs). For simplicity, the pseudocode below describes the construction of a single contig and does not cover the error-correction step. In reality, after a contig is constructed, ABruijn maps all reads to this contig and uses the remaining reads to construct other contigs. The contig generation procedure iteratively extends the current path in the positive strand direction. If the extension halts due to a path inconsistency, ABruijn attempts to extend the current contig to the opposite strand direction starting from the initial contig read.

```
function ABRUIJN(LongReads, k, t, minOverlap, jump, MaxOverhang)
    construct ABruijn graph on $(k,t)$-mers from LongReads
    ReadPaths← the set of paths in the assembly graph corresponding to
       all reads from LongReads
    InitialPath← an arbitrary read-path in the A-Bruijn graph
    GrowingPath← InitialPath
    ReadPath← InitialPath
    while forever do
        OverlapPaths← all paths in ReadPaths
             (w.r.t. minOverlap, jump and maxOverhang)
        if the set OverlapPaths is consistent then
            if InitialPath is a consistent path in OverlapPaths then
                return string spelled by GrowingPath (as a circular contig)
            end if
            ConsistentPath← most-consistent path in OverlapPaths
            extend GrowingPath by ConsensusPath
            ReadPath← ConsensusPath
        else
            return string spelled by GrowingPath (as one of the contigs)
        end if
    end while
end function
```

1.3.4 Common jump-Subpaths

Given a path P in a weighted directed graph (weights correspond to shifts in the A-Bruijn graph), we refer to the distance $dP(v,w)$ along path P between vertices v and w in this path (i.e., the sum of the weights of all edges in the path) as the P-distance. The span of a subpath of a path P is defined as the P-distance from the first to the last vertex of this subpath.

Given a parameter *jump*, a *jump*-subpath of P is a subsequence of vertices $v_1...v_t$ in P such that $dP(v_i, v_{i+1}) \leq jump$ for all i from 1 to $t-1$. We define $Path_{jump}(P)$ as a jump-subpath with the maximum span out of all jump-subpaths of a path P.

A sequence of vertices in a weighted directed graph is called a common *jump*-subpath of paths P_1 and P_2 if it is a *jump*-subpath of both P_1 and P_2 (Figure 1.3). The span of a common jump-subpath of P_1 and P_2 is defined as its span with respect to path P_1 (note that this definition is nonsymmetric with respect to P_1 and P_2). We refer to a common *jump*-subpath of paths P_1 and

P_2 with the maximum span as $Path_{jump}(P_1, P_2)$ (with ties broken arbitrarily).

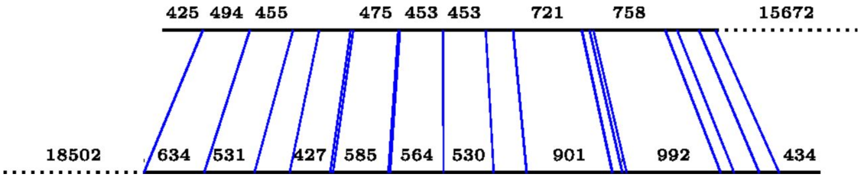


Figure 1.3: Two overlapping reads from the Ecoli dataset and their common *jump*-subpath with maximum span that contains 50 vertices and has span 6,714 with respect to the bottom read (for *jump* = 1,000). The left and right overhangs for these reads are 425 and 434. The weights of the edges in the A-Bruijn graph are shown only if they exceed 400 bp.

Below we describe how the ABruijn assembler uses the notion of common jump-subpaths with maximum span to detect overlapping reads.

Finding a Common jump-Subpath with Maximum Span. For the sake of simplicity, below we limit our attention to the case when paths P_1 and P_2 traverse each of their shared vertices exactly once.

A vertex w is a jump-predecessor of a vertex v in a path P if P traverses w before traversing v and $dP(w,v) \leq jump$.

We define $P(v)$ as the subpath of P from its first vertex to v. Given a vertex v shared between paths P_1 and P_2, we define $span_{jump}(v)$ as the largest span among all common jump-subpaths of paths $P_1(v)$ and $P_2(v)$ ending in v. The dynamic programming algorithm for finding a common jump-subpath with the maximum span is based on the following recurrence:

$$span_{jump}(v) = max_{all\ jump-predecessors\ w\ of\ v\ in\ P_1\ and\ P_2}\{span_{jump}(w) + dP1(w,v)\} \qquad (1.1)$$

A heuristic for finding a maximum common jump-subpath with maximum. We define $Pred_{jump}(v)$ as the set of all *jump*-predecessors of a vertex v in paths P_1 and P_2. A vertex w in $Pred_{jump}(v)$ is called *dominant* if it is not a *jump*-predecessor of any other vertex in $Pred_{jump}(v)$. If paths P_1 and P_2 traverse $Pred_{jump}(v)$ in the same order, then there is one dominant vertex in $Pred_{jump}(v)$, denoted as w, and $span_{jump}(v) = \{span_{jump}(w) + d_{P_1}(w,v)\}$. To speed-up the dynamic programming algorithm based on the recurrence in the main text, ABruijn stores and checks only the dominant vertices in $Pred_{jump}(v)$. A similar approach is used to find a common $(jump, \Delta)$-subpath with maximum span.

Our use of *k*-mers to identify overlapping reads has similarities with MHAP [Berlin et al., 2015] that utilizes hashing of all *k*-mers on every read as a way to identify overlaps. The key difference is that, while MHAP is applied to a pair of reads, ABruijn utilizes information from all reads in order to identify the set of solid *k*-mers that one should focus on, make extension decisions, identify chimeric reads, etc.

1.3.5 Path extensions in A-Bruijn graphs

Overlapping Paths in A-Bruijn Graphs. We define the right overhang between paths P_1 and P_2 as the minimum of the distances from the last vertex in $Path_{jump}(P_1, P_2)$ to the ends of P_1 and P_2. Similarly, the left overhang between paths P_1 and P_2 is the minimum of the distances from the starts of P_1 and P_2 to the first vertex in $Path_{jump}(P_1, P_2)$.

Given parameters *jump, minOverlap* and *maxOverhang*, we say that paths P_1 and P_2 overlap if they share a common *jump*-subpath of span at least *minOverlap* and their right and left overhangs do not exceed *maxOverhang*. To decide whether two reads have arisen from two overlapping regions in the genome, ABruijn checks whether their corresponding read-paths P_1 and P_2 overlap (with respect to parameters *jump, minOverlap*, and *maxOverhang*). Given overlapping paths P_1 and P_2, we say that P_1 is supported by P_2 if the P_1-distance from the last vertex in $Path_{jump}(P_1, P_2)$ to the end of P_1 is smaller than the P_2-distance from the last vertex in $Path_{jump}(P_1, P_2)$ to the end of P_2.

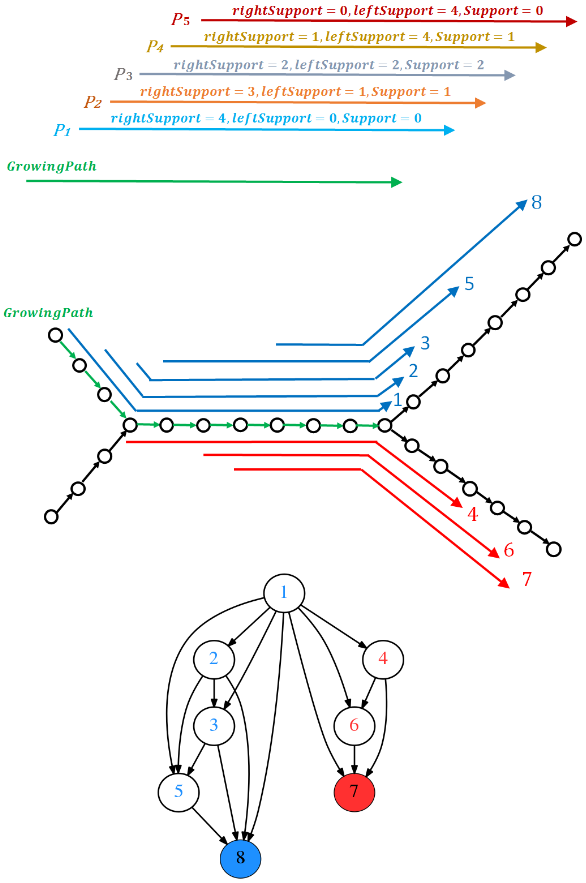
Most-Consistent Paths. Although it seems that the notion of overlapping paths allows us to implement the path extension paradigm for A-Bruijn graphs, there are two complications. First, the path extension algorithm becomes more complex when the growing path ends in a long repeat [Vasilinetc et al., 2015]. Second, chimeric reads may end up in the set of overlapping read-paths extending the growing path in the ABruijn algorithm. Also, a set of extension candidates may include a small fraction of spurious reads from other regions of the genome. Below we describe how ABruijn addresses these complications.

Given a path *P* in a set of paths *Paths*, we define *rightSupportPaths(P)* as the number of paths in *Paths* that support *P*. *leftSupportPaths(P)* is defined as the number of paths in *Paths* that are supported by *P*. We also define *SupportPaths(P)* as the minimum of *rightSupportPaths(P)* and *leftSupportPaths(P)*. A path *P* is most-consistent if it maximizes *SupportPaths(P)* among all paths in *Paths* (Figure 1.4 , Top).

Given a set of paths *Paths* overlapping with *ReadPath*, ABruijn selects a most-consistent path for extending *ReadPath*. Our rationale for selecting a most-consistent path is based on the observation that chimeric and spurious reads usually have either limited support or themselves support few other reads from the set *Paths*. For example, a chimeric read in *Paths* with a spurious suffix may support many reads in *Paths* but is unlikely to be supported by any reads in *Paths*.

Support Graphs. When exSPAnder extends the growing path, it takes into account the local repeat structure of the de Bruijn graph, resulting in a rather complex decision rule in the case when the growing path contains a repeat [Prjibelski et al., 2014, Vasilinetc et al., 2015]. Figure 1.4 (Middle) shows a fragment of the de Bruijn graph with a repeat of multiplicity 2 (internal edge), a growing path ending in this repeat (shown in green), and eight read-paths that extend this growing path. exSPAnder analyzes the subgraph of the de Bruijn graph traversed by the growing path, ignores paths starting in the edges corresponding to repeats, and selects the remaining paths as candidates for an extension (reads 1, 2, and 3 in Figure 1.4, Middle). Below we show how to detect that a growing path ends in a repeat in the absence of the de Bruijn graph and how to analyze read-paths ending/starting in a repeat in the A-Bruijn graph framework.

Figure 1.4, Bottom shows a support graph with eight vertices (each vertex corresponds to a read-path in Figure 1.4, Middle. There is an edge from a vertex *v* to a vertex *w* in this graph if



P_5 rightSupport = 0, leftSupport = 4, Support = 0
P_4 rightSupport = 1, leftSupport = 4, Support = 1
P_3 rightSupport = 2, leftSupport = 2, Support = 2
P_2 rightSupport = 3, leftSupport = 1, Support = 1
P_1 rightSupport = 4, leftSupport = 0, Support = 0
GrowingPath
8
5
GrowingPath
3
2
1
4
6
7
1
2
4
3
6
5
7
8

Figure 1.4: (Top) A growing path (shown in green) and a set of five paths *Paths* above it (extending this path). The gray path with *SupportPaths(P) = 2* is the most-consistent path in the set *Paths*. (Middle) A growing path (shown in green) ending in a repeat (represented by the internal edge in the graph), and eight read-paths that extend this growing path (five correct extensions shown in blue and three incorrect extensions shown in red. (Bottom) A support graph for the above eight read-paths. Note that the blue read-path 1 is connected by edges with all red read-paths because it is supported by all red paths even though these paths do not contain any short suffix of read-path 1 (the ABruijn graph framework is less sensitive than the de Bruijn graph framework with respect to overlap detection).

read v is supported by read w. The vertex of this graph with maximal indegree corresponds to the rightmost blue read-path (read 8) and reveals four other blue read-paths as its predecessors, that is, vertices connected to the vertex 8 (cluster of blue vertices in Figure 1.4, Bottom). The remaining three vertices in the graph represent incorrect extensions of the growing path and reveal that this growing path ends in a repeat (cluster of red vertices in Figure 1.4, Bottom). This toy example illustrates that decomposing the vertices of the support graph into clusters helps to answer the question of whether the growing path ends in a repeat (multiple clusters) or not (single cluster).

Although exSPAnder and ABruijn face a similar challenge while analyzing repeats, the ABruijn graph, in contrast to the de Bruijn graph, does not reveal local repeat structure. However, it allows one to detect reads ending in long repeats using an approach that is similar to the approach illustrated in Figure 1.4. Below we show how to detect such reads and how to incorporate their analysis in the decision rule of ABruijn.

Identifying Reads Ending/Starting in a Repeat. Given a set of reads Reads supporting a given read, we construct a support graph $G(Reads)$ on $|Reads|$ vertices. We further construct the transitive closure of this graph, denoted $G^{\star}(Reads)$, using the Floyd-Warshall algorithm. Figure 1.5 presents the graph $G(Reads)$ for a read that does not end in a long repeat and for another read that ends in a long repeat.

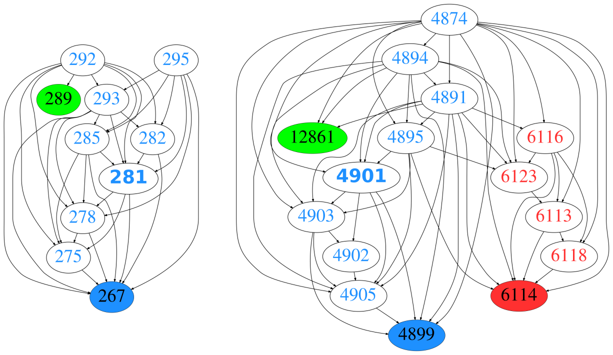


Figure 1.5: (Left) Support graph $G(Reads)$ for a read in the BLS dataset (Results, Datasets) that does not end in a long repeat. Reads in the BLS dataset are numbered in order of their appearance along the genome. The green vertex represents a chimeric read. The blue vertex has maximum degree in $G^{\star}(Reads)$ and reveals a single cluster consisting of all vertices but the green one. A vertex 281 with large indegree (5) and large outdegree (3) in $G^{\star}(Reads)$ is a most-consistent read-path, and it is selected for path extension (unless it ends in a repeat). (Right) Support graph $G^{\star}(Reads)$ for a read in the BLS dataset that ends in a long repeat. The green vertex represents a chimeric read. The blue vertex has maximum degree in $G^{\star}(Reads)$ and reveals a cluster consisting of nine blue vertices. The vertex 4901 with large indegree (4) and large outdegree (4) in $G^{\star}(Reads)$ is a most-consistent read-path, and it is selected for path extension if it does not start in a repeat. The red vertex reveals another cluster consisting of five red vertices. Generally, we expect that a read ending in a long repeat of multiplicity m will result in m clusters because reads originating different instances of this repeat are not expected to support each other and, thus, are not connected by edges in $G^{\star}(Reads)$.

ABruijn partitions the set of vertices in the graph $G^{\star}(Reads)$ into nonoverlapping clusters as follows. It selects a vertex v with maximum indegree in $G^{\star}(Reads)$ and, if this indegree exceeds a threshold (the default value is 1), removes this vertex along with all its predecessors from the graph. We refer to the set of removed vertices as a cluster of reads and iteratively repeat this procedure on the remaining subgraph until no vertex in the graph has indegree exceeding the threshold. Figure 1.5 illustrates that this decomposition results in a single cluster for a read that does not end in a repeat and in two clusters for a read that ends in a repeat.

We classify a read as a read ending in a repeat if the number of clusters in $G^{\star}(Reads)$ exceeds 1 (the notion of a read starting from a repeat is defined similarly). A set of reads is called inconsistent if all reads in this set either end or start in a repeat, and consistent otherwise. ABruijn detects all reads ending and starting in a repeat before the start of the path extension algorithm; 3.2% and 6.4% of all reads in Ecoli and BLS datasets, respectively, end in repeats.

The Path Extension Paradigm and Repeats. ABruijn attempts to exclude reads ending in repeats while selecting a read that extends the growing path. Because this is not always possible, below we describe two cases: The growing path does not end in a repeat and the growing path ends in a repeat.

If the growing path does not end in a repeat, our goal is to exclude chimeric and spurious reads during the path extension process. ABruijn, thus, selects a read from *Reads* that (i) does not end in a repeat and (ii) supports many reads and is supported by many reads. Condition (ii) translates into selecting a vertex whose indegree and outdegree are both large (i.e., a most-consistent path). In the case that all reads in *Reads* end in a repeat, ABruijn selects a read that satisfies the condition (ii) but ends in a repeat.

If the growing path ends in a repeat, ABruijn uses a strategy similar to exSPAnder to avoid reads that start in a repeat as extension candidates (e.g., all reads in Figure 1.4, Middle except for reads 1, 2, and 3). It thus selects a read from Reads that (i) does not start in a repeat and (ii) supports many reads and is supported by many reads. To satisfy condition (ii), ABruijn selects a most-consistent read among all reads in *Reads* that do not start in a repeat. If there are no such reads, ABruijn halts the path extension procedure.

1.3.6 Correcting Errors in the Draft Genome

Matching Reads Against the Draft Genome. ABruijn uses BLASR [Chaisson and Tesler, 2012] to align all reads against the draft genome. It further combines pairwise alignments of all reads into a multiple alignment. Because this alignment against the error-prone draft genome is rather inaccurate, we need to modify it into a different alignment that we will use for error correction.

Our goal now is to partition the multiple alignment of reads to the entire draft genome into thousands of short segments (mini-alignments) and to error-correct each segment into the consensus string of the mini-alignment. The motivation for constructing mini-alignments is to enable accurate error-correction methods that are fast when applied to short segments of reads but become too slow in the case of long segments.

The task of constructing mini-alignments is not as simple as it may appear. For example, breaking the multiple alignment into segments of fixed size will result in inaccurate consensus sequences because a region in a read aligned to a particular segment of the draft genome has not necessarily arisen from this segment (e.g., it may have arisen from a neighboring segment or from a different instance of a repeat). Because many segments in BLASR alignments are misaligned, the accuracy of our error-correction approach (that is designed for well-aligned reads) may deteriorate.

We, thus, search for a good partition of the draft genome that satisfies the following criteria: (i) Most segments in the partition are short, so that the algorithm for their error-correction is fast, and (ii) with high probability, the region of each read aligned to a given segment in the partition represents an error-prone version of this segment. Below we show how to construct a good partition by building an A-Bruijn graph.

Defining Solid Regions in the Draft Genome. We refer to a position (column) of the alignment with the space symbol "-" in the reference sequence as a non-reference position (column) and to all other positions as a reference position (column). We refer to the column in the multiple alignment containing the *i*-th position in a given region of the reference genome as the *i*-th column. The total number of reads covering a position *i* in the alignment is referred to as *Cov(i)*.

A non-space symbol in a reference column of the alignment is classified as a match (or a substitution) if it matches (or does not match, respectively) the reference symbol in this column. A space symbol in a reference column of the alignment is classified as a deletion. We refer to the number of matches, substitutions, and deletions in the *i*-th column of the alignment as *Match(i), Sub(i)*, and *Del(i)*, respectively. We refer to a non-space symbol in a non-reference column as an insertion and denote $Ins(i)$ as the number of nucleotides in the non-reference columns flanked between the reference columns *i* and $i+1$ (Figure 1.6).

For each reference position *i*, *Cov(i) = Match(i) + Sub(i) + Del(i).* We define the match, substitution, and insertion rates at position *i* as *Match(i) / Cov(i)*, *Sub(i) / Cov(i)*, *Del(i) / Cov(i)*, and *Ins(i) / Cov(i)*, respectively. Given an *l*-mer in a draft genome, we define its local match rate as the minimum match rate among the positions within this *l*-mer. We further define its local insertion rate as the maximum insertion rate among the positions within this *l*-mer.

An *l*-mer in the draft genome is called (α, β)-solid if its local match rate exceeds α and its local insertion rate does not exceed β. When α is large and β is small, (α, β)-solid *l*-mers typically represent the correct *l*-mers from the genome. The last row in Figure 1.6, Bottom Left shows all of the (0.8, 0.2)-solid 4-mers in the draft genome. The contiguous sequence of (α, β)-solid *l*-mers forms a solid region. Our goal now is to select a position within each solid region (referred to as a landmark) and to form mini-alignments from the segments of reads spanning the intervals between two consecutive landmarks.

Breaking the Multiple Alignment into Mini-Alignments. An *l*-mer is called simple if

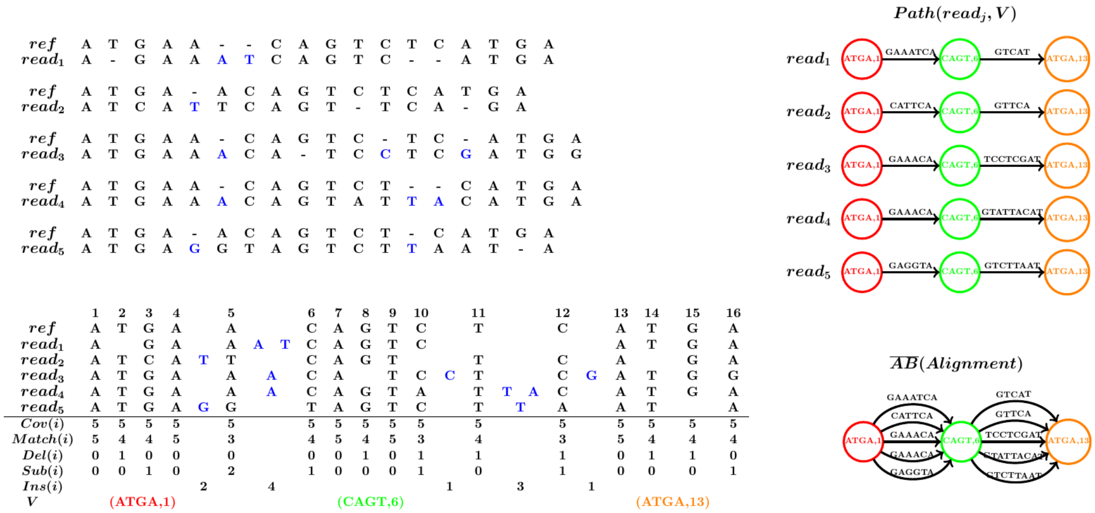

ref A T G A A - - C A G T C T C A T G A
read1 A - G A A A T C A G T C - - A T G A
ref A T G A - A C A G T C T C A T G A
read2 A T C A T T C A G T - T C A - G A
ref A T G A A - C A G T C - T C - A T G A
read3 A T G A A A C A - T C C T C G A T G G
ref A T G A A - C A G T C T - - C A T G A
read4 A T G A A A C A G T A T T A C A T G A
ref A T G A - A C A G T C T - C A T G A
read5 A T G A G G T A G T C T T A A T - A
Path(read_j, V)
read1 ATGA,1 GAAATCA CAGT,6 GTCAT ATGA,13
read2 ATGA,1 CATTCA CAGT,6 GTTCA ATGA,13
read3 ATGA,1 GAAACA CAGT,6 TCCTCGAT ATGA,13
read4 ATGA,1 GAAACA CAGT,6 GTATTACAT ATGA,13
read5 ATGA,1 GAGGTA CAGT,6 GTCTTAAT ATGA,13
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
ref
read1
read2
read3
read4
read5
Cov(i) 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
Match(i) 5 4 4 5 3 4 5 4 5 3 4 3 5 4 4 4
Del(i) 0 1 0 0 0 0 0 1 0 1 1 1 0 1 1 0
Sub(i) 0 0 1 0 2 1 0 0 0 1 0 1 0 0 0 1
Ins(i) 2 4 1 3 1
V (ATGA,1) (CAGT,6) (ATGA,13)
AB(Alignment)
GAAATCA
CATTCA
GAAACA
GAAACA
GAGGTA
GTCAT
GTTCA
TCCTCGAT
GTATTACAT
GTCTTAAT
ATGA,1
CAGT,6
ATGA,13

Figure 1.6: (Top Left) The pairwise alignments between a reference region ref in the draft genome and five reads $Reads = read_1, read_2, read_3, read_4, read_5$. All inserted symbols in these reads with respect to the region ref are colored in blue. (Bottom Left) The multiple alignment Alignment constructed from the above pairwise alignments along with the values of *Cov(i), Match(i), Del(i), Sub(i)* and *Ins(i)*. The last row shows the set V of (0.8,0.2)-solid 4-mers. The non-reference columns in the alignment are not numbered. (Right) Constructing $\overline{AB}(Alignment)$, that is, combining all paths $Path(read_j, V)$ into $\overline{AB}(Alignment)$. Note that the 4-mer ATGA corresponds to two different nodes with labels 1 and 13. The three boundaries of the mini-alignments are between positions 2 and 3, 7 and 8, and 14 and 15. The two resulting necklaces are formed by segments $\{GAATCA, GATTCA, GAAACA, GAAACA, GAGGTA\}$ and $\{GTCAT, GTTCA, TCCTCGAT, GTATTACAT, GTCTTAAT\}$.

all its consecutive nucleotides are different. For example, *CAGT* and *ATGA* are simple 4-mers, and *GTTC* is not a simple 4-mer. ABruijn selects simple 4-mers that are at least l positions away from each other within solid regions as landmarks. We introduce multiple (rather than a single) landmarks in some solid regions to minimize the size of mini-alignments resulting from long solid regions. We further use the middle points (i.e., a point between its 2nd and 3rd nucleotides) of selected simple 4-mers as landmarks. This procedure resulted in 159,142 mini-alignments for the Ecoli dataset. ABruijn analyzes each mini-alignment and error-corrects each segment between consecutive landmarks.

Constructing the A-Bruijn Graph on Solid Regions in the Draft Genome. We refer to the multiple alignment of all reads against the draft genome as Alignment. We label each landmark by its landmark position in *Alignment* and break each read into a sequence of segments aligned between consecutive landmarks. We further represent each read as a directed path through the vertices corresponding to the landmarks that it spans over. To construct the A-Bruijn graph $\overline{AB}(Alignment)$, we glue all identically labeled vertices in the set of paths resulting from the reads (Figure 1.6, Right).

Labeling vertices by their positions in the draft genome (rather than the sequences of landmarks) distinguishes identical landmarks from different regions of the genome and prevents excessive gluing of vertices in the A-Bruijn graph $\overline{AB}(Alignment)$. We note that whereas the A-Bruijn graph constructed from reads is very complex, the A-Bruijn graph $\overline{AB}(Alignment)$ constructed from reads aligned to the draft genome is rather simple. Although there are many bubbles in this graph, each bubble is simple, making the error correction step fast and accurate.

The edges between two consecutive landmarks (two vertices in the A-Bruijn graph) form a *necklace* consisting of segments from different reads that align to the region flanked by these landmarks (Figure 1.6, Right shows two necklaces). Below we describe how ABruijn constructs a consensus for each necklace (called the necklace consensus) and transforms the inaccurate draft genome for the Ecoli dataset into a polished genome to reduce the error rate to 0.0004% for the Ecoli dataset (only 19 putative errors for the entire genome).

1.3.7 Error-Correcting Mini-Alignments

A Probabilistic Model for Necklace Polishing. Each necklace contains read-segments

$$Segments = seg_1, seg_2, ..., seg_m \tag{1.2}$$

and our goal is to find a consensus sequence *Consensus* maximizing

$$Pr(Segments|Consensus) = \Pi_{i=1}^{m} Pr(seg_i|Consensus) \tag{1.3}$$

where $Pr(seg_i|Consensus)$ is the probability of generating a segment seg_i from a consensus sequence *Consensus*. Given an alignment between a segment seg_i and a consensus *Consensus*, we define $Pr(seg_i|Consensus)$ as the product of all match, mismatch, insertion, and deletion rates for all positions in this alignment. The match, mismatch, insertion, and deletion rates should be derived using an alignment of any set of reads to any reference genome.

ABruijn selects a segment of median length from each necklace and iteratively checks whether the consensus sequence for each necklace can be improved by introducing a single mutation in the selected segment. If there exists a mutation that increases $Pr(Segments|Consensus)$, we select the mutation that results in the maximum increase and iterate until convergence. We further output the final sequence as the error-corrected sequence of the necklace. As described in [Chin et al., 2013], this greedy strategy can be implemented efficiently because a mutation maximizing $Pr(Segments|Consensus)$ among all possible mutated sequences can be found in a single run of the forward-backward dynamic programming algorithm for each sequence in *Segments*. The error rate after this step drops to 0.003% for the Ecoli dataset.

Error-Correcting Homonucleotide Runs. The probabilistic approach described above works well for most necklaces but its performance deteriorates when it faces the difficult problem of estimating the lengths of homonucleotide runs, which account for 46% of the *E. coli* genome (see discussion on pulse merging in [Chin et al., 2013]). We, thus, complement this approach with a homonucleotide likelihood function based on the statistics of homonucleotide runs. In contrast to previous approaches to error-correction of long error-prone reads, this new likelihood function incorporates all corrupted versions of all homonucleotide runs across the training set of reads and reduces the error rate sevenfold (from 0.003% to 0.0004% for the Ecoli dataset) compared with the standard likelihood approach.

To generate the statistics of homonucleotide runs, we need an arbitrary set of reads aligned against a training reference genome. For each homonucleotide run in the genome and each read spanning this run, we represent the aligned segment of this read simply as the set of its nucleotide counts. For example, if a run *AAAAAAA* in the genome is aligned against *AATTACA* in a read, we represent this read-segment as *4A3X*, where *X* stands for any nucleotide differing from *A*. After collecting this information for all runs of *AAAAAAA* in the reference genome, we obtain the statistics for all read segments covering all instances of the homonucleotide run *AAAAAAA*. We further use the frequencies in this table for computing the likelihood function as the product of these frequencies for all reads in each necklace (frequencies below a threshold 0.001 are ignored). It turned out that the frequencies in the resulting table hardly change when one changes the dataset of reads, the reference genome, or even the sequencing protocol from P6-C4 to the older P5-C3. To decide on the length of a homonucleotide run, we simply select the length of the run that maximizes the likelihood function.

Although the described error-correcting approach results in a very low error rate even after a single iteration, ABruijn realigns all reads and error-corrects the pre-polished genome in an iterative fashion (three iterations by default).

1.4 Results

1.4.1 Datasets

The *E. coli* K12 dataset [Kim et al., 2014] (referred to as Ecoli) contains 10,277 reads with ≈ 55× coverage generated using the P6-C4 Pacific Biosciences technology.

The *E. coli* K12 Oxford Nanopore dataset [Loman et al., 2015] (referred to as EcoliNano) contains 22,270 reads with ≈ 29× coverage.

The BLS and PXO datasets were derived from *Xanthomonas oryzae* strains BLS256 and PXO99A previously assembled using Sanger reads [Bogdanove et al., 2011, Salzberg et al., 2008] and reassembled using Pacific Biosciences P6-C4 reads in [Booher et al., 2015]. The BLS dataset contains 89,634 reads (≈ 234× coverage), and the PXO dataset contains 55,808 reads (≈ 141× coverage). The assembly of BLS and PXO datasets is particularly challenging because these genomes have a large number of tal genes.

The *Bryozoa neritina* dataset (referred as BNE) contains 1,127,494 reads (estimated coverage $\approx 25\times$) generated using the P6-C4 Pacific Biosciences technology. *B. neritina* is a microscopic marine eukaryote that forms colonies attached to the wet surfaces and forms symbiotic communities with various bacteria. *B. neritina* is the source of bryostatin, an anticancer and memory-enhancing compound [Trost and Dong, 2008]. *B. neritina* is also a model organism for biofouling, studies of accumulation of various organisms on wetted surfaces that present a risk to underwater construction.

The symbiotic bacteria live inside of *B. neritina* making it impossible to isolate the *B. neritina* DNA from the bacterial DNA for genome sequencing. As the result, despite the importance of *B. neritina*, all attempts to sequence it so far have failed [Lopanik et al., 2008]. The total genome size of the symbiotic bacteria in *B. neritina* is significantly larger than the estimated size of the *B. neritina* genome (135 Mb). Thus, sequencing *B. neritina* presents a complex metagenomics challenge.

The *S. cerevisiae* W303 dataset [Kim et al., 2014] (referred as SCE) contains 232,230 reads with $\approx$117X coverage generated using the P5-C3 Pacific Biosciences technology.

1.4.2 Assembling the Ecoli Dataset

The Challenge of Benchmarking SMS Assemblies. Because Canu [Koren et al., 2017] improved on PBcR [Koren et al., 2012] with respect to both speed and accuracy, we limited our benchmarking to ABruijn and Canu v1.2.

High-quality short-read bacterial assemblies typically have error-rates on the order of 105, which typically result in 50 to 100 errors per assembled genome [Ronen et al., 2012]. Because assemblies of high-coverage SMS datasets are often even more accurate than assemblies of short reads, short-read assemblies do not represent a gold standard for estimating the accuracy of SMS assemblies. Moreover, the *E. coli* K12 strain used for SMS sequencing of the Ecoli dataset differs from the reference genome. Thus, the standard benchmarking approach based on comparison with the reference genome [Gurevich et al., 2013] is not applicable to these assemblies.

We used the following approach to benchmark ABruijn and Canu against the reference *E. coli* K12 genome. There are 2,892 and 2,887 positions in *E. coli* K12 genome where the reference sequence differs from ABruijn and Canu+Quiver, respectively. However, ABruijn and Canu+Quiver agree on 2,873 of them, suggesting that most of these positions represent mutations in *E. coli* K12 compared with the reference genome. Both Canu+Quiver and ABruijn suggest that the Ecoli dataset was derived from a strain that differs from the reference *E. coli* K12 genome by a 1,798-bp inversion, two insertions (776 and 180 bp), one deletion (112 bp), and seven other single positions. We, thus, revised the *E. coli* K12 genome to account for these variations and classified a position as an ABruijn error if the Canu+Quiver sequence at this position agreed with the revised reference but not with the ABruijn sequence (Canu errors are defined analogously).

Comparing ABruijn and Canu using the Ecoli dataset. ABruijn and Canu assembled the Ecoli dataset into a single circular contig structurally concordant with the *E. coli* genome. We further estimated the accuracy of ABruijn and Canu in projects with lower coverage by down-sampling the reads from Ecoli. For each value of coverage, we made five independent replicas and analyzed errors in all of them.

In contrast to ABruijn, Canu does not explicitly circularize the reconstructed bacterial chromosomes but instead outputs each linear contig with an identical (or nearly identical) prefix and suffix. We used these suffixes and prefixes to circularize bacterial chromosomes and did not count differences between some of them as potential Canu errors. However, for some replicas with coverage 40×, 35×, 30×, and 25×, Canu missed short 2-kb to 7-kb fragments of the genome (possibly due to low coverage in some regions), thus, preventing us from circularization. To enable benchmarking, we did not count these missing regions as Canu errors. Also, at coverage 30×, Canu (i) failed to assemble the Ecoli dataset into a single contig for one out of five replicas and (ii) correctly assembled bacterial chromosome for another replica but also generated a false contig (probably formed by chimeric reads). In contrast, ABruijn correctly assembled all replicas for all values of coverage.

Table 1.1 illustrates that, in contrast to ABruijn, Canu generates rather inaccurate assemblies without Quiver, a tool that uses raw machine-level HDF5 signals for polishing: 637 errors (160 insertions and 477 deletions) and 19 errors (12 insertions and 7 deletions) for Canu and ABruijn, respectively. However, after applying Quiver, the number of errors reduces to 14 (1 insertion and 13 deletions) and 15 (2 insertions and 13 deletions) for Canu and ABruijn, respectively. ABruijn assembled the Ecoli dataset in ≈ 8 min and polished it in ≈ 36 min (the memory footprint was 2 Gb). ABruijn and Canu have similar running times: 2,599 s and 2,488 s, respectively (4,873 s and 4,803 s for ABruijn+Quiver and Canu+Quiver, respectively).

Table 1.1: Summary of errors for Canu and ABruijn assemblies of the Ecoli, BLS, and PXO datasets as well as for the downsampled Ecoli datasets with coverage varying from 50× to 25×

Dataset	Canu	ABruijn	Canu+Quiver	ABruijn+Quiver
BLS	73	5	51	31
PXO	1,162	21	130	15
Ecoli	637	19	14	15
Ecoli 50x	703	33	20	18
Ecoli 45x	829	45	29	29
Ecoli 35x	1,541	153	88	84
Ecoli 30x	2,470	291	175	154
Ecoli 25x	3,053	687	322	329

To enable a fair benchmarking and to offset the artifacts of Canu assemblies at 30× coverage, we collected statistics of errors for four out of five best assemblies for each value of coverage. Table 1.1 illustrates that both ABruijn and Canu maintain accuracy even in relatively low coverage projects but Canu assemblies become fragmented and may miss short segments when the coverage is low.

1.4.3 Assembling the EcoliNano Dataset

Both the Nanocorrect assembler [Loman et al., 2015] and ABruijn assembled the EcoliNano dataset into a single circular contig structurally concordant with the *E. coli* K12 genome with error rates 1.5% and 1.1%, respectively (2,475 substitutions, 9,238 insertions, and 40,399 deletions for ABruijn). We note that, in contrast to the more accurate Pacific Biosciences technology, Oxford Nanopore technology currently has to be complemented by hybrid coassembly with short reads to generate finished genomes [Antipov et al., 2015, Labonté et al., 2015, Ashton et al., 2015, Risse et al., 2015].

Although further reduction in the error rate in Oxford Nanopore assemblies can be achieved by processing of the signal resulting from DNA translocation [Loman et al., 2015], it is still two orders of magnitude higher that the error rate for the down-sampled Ecoli dataset with similar 30× coverage by Pacific Biosciences reads (Table 1.1) and below the acceptable standards for finished genomes. Because Oxford Nanopore technology is rapidly progressing, we decided not to optimize it further using signal processing of raw translocation signals.

1.4.4 Assembling *Xanthomonas* Genomes

Because HGAP 2.0 failed to assemble the BLS dataset, [Booher et al., 2015] developed a special PBS algorithm for local tal gene assembly to address this deficiency in HGAP. They further proposed a workflow that first launches PBS and uses the resulting local tal gene assemblies as seeds for a further HGAP assembly with custom adjustment of parameters in HGAP/Celera workflows. Although HGAP 3.0 resulted in an improved assembly of the BLS dataset, [Booher et al., 2015] commented that the PBS algorithm is still required for assembling other *Xanthomonas* genomes. Because PBS represents a customized assembler for tal genes that is not designed to work with other types of complex repeats, development of a general SMS assembly tool that accurately reconstructs repeats remains an open problem.

We launched ABruijn with the automatically selected parameters $t = 28$ and $t = 18$ for the BLS and PXO datasets, respectively (all other parameters were the same default parameters that we used for the Ecoli dataset). ABruijn assembled the BLS dataset into a circular contig structurally concordant with the BLS reference genome. It also assembled the PXO dataset into a circular contig structurally concordant with the PXO reference genome but, similarly to the initial assembly in [Booher et al., 2015], it collapsed a 212-kb tandem repeat.

Canu assembled the BLS dataset into a circular contig structurally concordant with the BLS reference genome but assembled the PXO dataset into two contigs, a long contig similar to the reference genome (with a collapsed 212-kb tandem repeat and three large indels of total length over 1,500 nucleotides) and a short contig. In summary, ABruijn+Quiver and Canu+Quiver assemblies of the BLS dataset resulted in only 31 and 51 errors, respectively. Surprisingly, ABruijn without Quiver resulted in a better assembly than ABruijn+Quiver with only five errors.

To evaluate errors for the PXO dataset, we decided to ignore the short contig generated by Canu and a collapsed 212-kb repeat (generated by both Canu and ABruijn). ABruijn+Quiver assembly of the PXO dataset resulted in only 15 errors whereas Canu+Quiver assembly resulted in 130 errors, including one insertion of 100 nucleotides.

1.4.5 Assembling the *B. neritina* Metagenome

We have assembled the *B. neritina* metagenome and further analyzed all long contigs at least 50 kb in size (1,319 and 1,108 long contigs for Canu and ABruijn, respectively). We ignored shorter contigs because they are often formed by a few reads or even a single read. The total length of long contigs was 171 Mb for Canu and 202 Mb for ABruijn. Figure 1.7 shows the histogram of the total length of contigs with a given coverage. Because the spread of the distribution of coverage for *B. neritina* significantly exceeds the spread we observed in other SMS datasets (typically within 15% of the average coverage), we attribute most bins with coverage below $20\times$ to contigs from symbiotic bacteria (the tallest peak in the histogram suggests that the average coverage of *B. neritina* is 25×). Running AntiSmash [Medema et al., 2011] on the ABruijn assembly revealed nine bacterial biosynthetic gene clusters encoding natural products that, similarly to bryostatin, may represent new bioactive compounds.

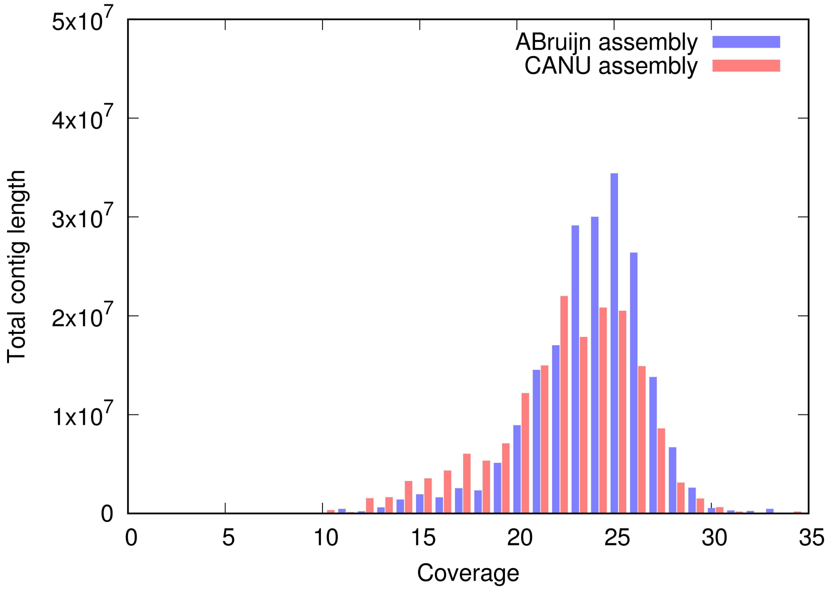


Figure 1.7: Contig length distribution for ABruijn and Canu assemblies of *B. neritina* metagenome.

We attribute the large difference in the total contig length to fragmentation in Canu assemblies in the case of low-coverage datasets, which we observed in our analysis of the downsampled Ecoli datasets. This fragmentation may have also contributed to differences in the N50 (98 kb vs. 242 kb) between Canu and ABruijn.

However, differences in N50 are poor indicators of assembly quality in the case when the reference genome is unknown. We, thus, conducted an additional analysis using the Core Eukaryotic Genes Mapping Approach (CEGMA) that was used in hundreds of previous studies for evaluating the completeness of eukaryotic assemblies [Parra et al., 2007]. CEGMA evaluates an assembly by checking whether its contigs encode all 248 ultraconserved eukaryotic core protein families. Canu and ABruijn assemblies missed 18 and 11 out of 248 core genes, respectively (7.3% vs. 4.4%). Thus, although both Canu and ABruijn generated better assemblies than typical eukaryotic short read assemblers (that often miss over 20% of core genes), the ABruijn assembly improved on the Canu assembly in this respect.

1.4.6 Assembling the *S. cerevisiae* W303 genome

Since *S. cerevisiae* W303 genome has not been finished using an alternative sequencing technology yet, we use its closest finished reference *S. cerevisiae* S288c (12,157,105 nucleotides, NCBI Assembly GCF_000146045.2) for esimating the accuracy of the ABruijn assembly. We estimated the average percent identity between the *S. cerevisiae* W303 and *S. cerevisiae* S288c genomes by comparing the longest contig assembled by ABruijn and PBcR-MHAP [Berlin et al., 2015] that is structurally concordant with the entire chromosome IV in *S. cerevisiae* S288c. ABruijn and PBcR-MHAP contigs featured 99.92% similarity with each other but only 97.8% similarity with chromosome IV. High similarity between ABruin and PBcR-MHAP assemblies suggests that many differences between these assemblies and chromosome IV represent structural variations rather than assembly errors.

Considering only long contigs (longer than 50 Kb), both PBcR-MHAP assemblies [Berlin et al., 2015] and ABruijn assemblies of the SCE dataset were largely structurally concordant with sixteen chromosomes of the *S. cerevisiae* S288C genome. Although QUAST with default parameters reported 77 and 72 misassemblies for 20 long contigs in the PBcR-MHAP assembly and 24 long contigs in the ABruijn assembly, respectively, most of these misassemblies represent structural variations or regions of high divergence as compared to the reference genome (e.g., the PBcR-MHAP and ABruijn assemblies coincided with each other in most regions where QUAST reported misassemblies). The total contig length for the PBcR-MHAP assembly was slightly longer than for the ABruijn assembly (12.18 Mb vs. 12.08 Mb) but its duplication ratio was slightly larger.

It is not clear whether the small difference in the total contig length represents an improvement in assembly or a reporting artifact. For example, while the longest contig in the

PBcR-MHAP and ABruijn assemblies (1.548 Mb and 1.532 Mb, respectively) are structurally concordant with chromosome IV in *S. cerevisiae* S288C, the PBcR-MHAP contig is slightly longer. However, the 14 kb long suffix of this contig does not align to the reference chromosome IV. Therefore, it remains unclear whether this suffix represents an extension of chromosome IV as compared to the *S. cerevisiae* S288C genome or an assembly artifact.

To offset the effect of differences with the reference genome on the number of misassemblies, we increased the QUAST parameter *extensive-mis-size* from its default value 1 kb to 40 kb to mask out the large structural variations between *S. cerevisiae* S288C and *S. cerevisiae* W303 genomes. After this increase, QUAST reported no misassemblies for PBcR-MHAP and one misassembly for ABruijn. Thus, most of misassemblies reported by QUAST with the default 1 kb value of the *extensive-mis-size* parameter likely represent insertions of mobile elements, large indels (longer than 1 kb), or long regions with high divergence as compared to the reference.

1.4.7 Running time and memory footprint

For the *Xanthomonas* genomes, which have complex repeat structure and high coverage, the assembly time and memory footprint increased as compared to the Ecoli dataset: 28 m assembly step, 160 m polishing step, 15 Gb memory for the PXO dataset and 68 m assembly step, 359 m polishing step, 21 Gb memory for the BLS dataset (Intel Core i7-4790 3.60 GHz with 4 cores (8 threads), 32Gb of RAM).

The running time increased to 4 h 53 m (memory footprint 2 Gb) for the EcoliNano dataset. The increase in the running time is attributed to the polishing step since Oxford Nanopore reads are less accurate than Pacific Biosciences reads (the assembly step took only 4 minutes).

In contrast, the running time for the SCE dataset was dominated by the assembly step (8h44m for the assembly step and 2 h 30 m for the polishing step). The increase in the running time of the assembly step is explained by many long and highly conserved *Ty1 - Ty 5* repeats and long segmental duplications.

For the BNE metagenome, assembly step took 19 h 10 m, polishing step took 28 h 56 m, and the memory footpring was 278 Gb (64 cores, AMD Opteron 6376 2.30 GHz, 512 Gb of RAM).

1.5 Discussion

We developed the ABruijn algorithm aimed at assembling bacterial and relatively small eukaryotic genomes from long error-prone reads. Because the number of bacterial genomes that are currently being sequenced exceeds the number of all other genome sequencing projects by an order of magnitude, accurate sequencing of bacterial genomes remains an important goal. Because short-read technologies typically fail to generate long contiguous assemblies (even in the case of bacterial genomes), long reads are often necessary to span repeats and to generate accurate genome reconstructions.

Because traditional assemblers were not designed for working with error-prone reads, the common view is that OLC is the only approach capable of assembling inaccurate reads and that these reads must be error-corrected before performing the assembly [Berlin et al., 2015]. We have demonstrated that these assumptions are incorrect and that the A-Bruijn approach can be used for assembling genomes from long error-prone reads. We believe that initial assembly with ABruijn, followed by construction of the de Bruijn graph of the resulting contigs, followed by a de Bruijn graph-aware reassembly with ABruijn may result in even more accurate and contiguous assemblies of SMS reads.

1.6 Acknowledgments

We thank Dmitry Antipov, Bahar Behsaz, Adam Bogdanove, Anton Korobeinikov, Mihai Pop, Steven Salzberg, and Glenn Tesler for their many useful comments; Mike Rayko for his help

with analyzing the *B. neritina* assemblies; and Alexey Gurevich for his help with QUAST and AntiSmash.

This chapter, in full, is a reprint of the material as it appears in Y. Lin, J. Yuan, M. Kolmogorov, M. Shen, M. Chaisson and P. Pevzner, “Assembly of long error-prone reads using de Bruijn graphs”, Proceedings of the National Academy of Sciences (2016). The dissertation author was the primary developer of the ABruijn project and one of the three lead authors of this paper.

Chapter 2

Assembly of long, error-prone reads using repeat graphs

2.1 Abstract

Accurate genome assembly is hampered by repetitive regions. Although long single molecule sequencing reads are better able to resolve genomic repeats than short-read data, most long-read assembly algorithms do not provide the repeat characterization necessary for producing optimal assemblies. Here, we present Flye, a long-read assembly algorithm that generates arbitrary paths in an unknown repeat graph, called disjointigs, and constructs an accurate repeat graph from these error-riddled disjointigs. We benchmark Flye against five state-of-the-art assemblers and show that it generates better or comparable assemblies, while being an order of magnitude faster. Flye nearly doubled the contiguity of the human genome assembly (as measured by the NGA50 assembly quality metric) compared with existing assemblers.

2.2 Introduction

Genome assembly is the process of reconstructing genomes from DNA sequence reads. In repetitive regions of the genome, accurately assembling short reads is challenging and can lead to inaccurate or unresolved assemblies. Single molecule sequencing (SMS) long-read technologies (such as Pacific Biosciences and Oxford Nanopore) have been used to improve the resolution of repetitive genomic regions, but many long stretches of repetitive DNA remain intractable to these approaches. Current SMS assemblers, such as PBcR [Koren et al., 2012, Chin et al., 2013, Berlin et al., 2015], Falcon [Chin et al., 2016], Miniasm [Li, 2016], ABruijn [Lin et al., 2016], HINGE [Kamath et al., 2017], Canu [Koren et al., 2017], and Marvel [Nowoshilow et al., 2018], have been used to successfully resolve some repeat regions across complex genomes, but correct assembly of long reads in long and highly repetitive genomic regions remains challenging. As a result, long-read technologies are often complemented by proximity ligation techniques (Hi-C) [Ghurye et al., 2017] and optical [Weissensteiner et al., 2017] mapping data to improve the contiguity of assemblies.

The de Bruijn graph has been used by short-read assembly approaches to represent genomic repeats as a repeat graph. Previous studies have demonstrated the value of this approach for improving the accuracy of genome assembly [Pevzner et al., 2004]. Recently, long-read assemblers such as ABruijn [Lin et al., 2016] and HINGE [Kamath et al., 2017], which capitalize on a similar de Bruijn graph-based approach, have been developed. Most short-read assemblers construct the de Bruijn graph based on all *k*-mers in reads and further transform it into a simpler de Bruijn assembly graph [Bankevich et al., 2012]. This approach collapses multiple instances of the same repeat into a single path in the assembly graph and represents the genome as a genome tour, which visits each edge in the assembly graph. However, in the case of SMS reads, the key assumption of the de Bruijn graph approach – that most *k*-mers from the genome are preserved in multiple reads – does not hold. As a result, various challenges that have been addressed for short-read assembly, such as how to deal with the fragmented de Bruijn graph and how to transform it into an assembly graph, remain largely unaddressed in long-read assemblers.

2.3 Methods

2.3.1 Flye outline

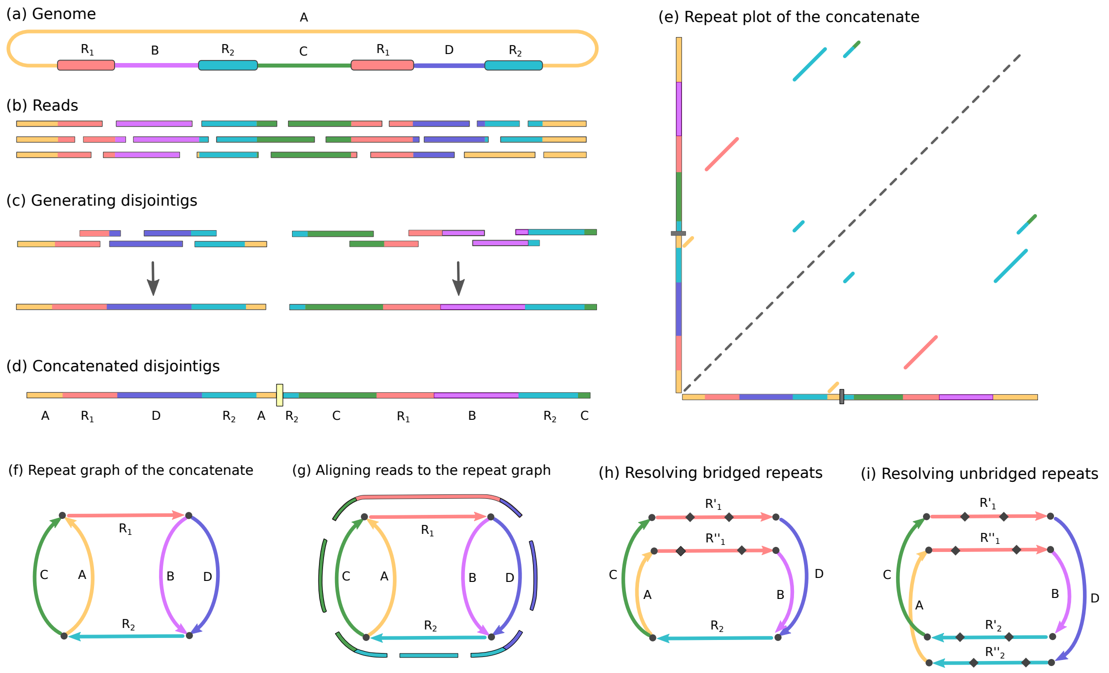

(a) Genome
A
R₁
B
R₂
C
R₁
D
R₂
(b) Reads
(c) Generating disjointigs
(d) Concatenated disjointigs
A
R₁
D
R₂
A
R₂
C
R₁
B
R₂
C
(e) Repeat plot of the concatenate
(f) Repeat graph of the concatenate
R₁
C
A
B
D
R₂
(g) Aligning reads to the repeat graph
R₁
C
A
B
D
R₂
(h) Resolving bridged repeats
R'₁
R''₁
C
A
B
D
R₂
(i) Resolving unbridged repeats
R'₁
R''₁
C
A
B
D
R'₂
R''₂

Figure 2.1: (a) A genome with two 99% identical copies of a repeat R_1 and two 99% identical copies of a repeat R_2. Segments A, B, C, and D represent non-repetitive regions. (b) A set of reads sampled from the genome. (c) Two (misassembled) disjointigs AR_1DR_2 A and $R_2CR_1BR_2C$ derived from the reads. (d) Concatenate of the disjointigs. (e) Repeat plot of the concatenate. (f) Repeat graph constructed by “gluing” vertices in the concatenate according to the repeat plot. For each two-dimensional point (x,y) in the repeat plot, we glue vertices x and y in the concatenate. (g) Aligning reads against the repeat graph. (h) Resolving the bridged repeat R_1 and reconstructing its two copies R'_1 and R''_1. The differences between each copy of this repeat and the consensus of this repeat are shown as small diamonds. (i) Resolving the unbridged repeat R_2 with two slightly diverged copies. Appendix 2.7.1 describes the Flye assembly of a simulated genome modeled after the genome shown in (a).

Here we describe the Flye algorithm for accurately assembling long reads (Fig. 2.1). Unlike existing assemblers that attempt to generate contigs, Flye initially generates disjointigs that represent concatenations of multiple disjoint genomic segments, concatenates all error-prone disjointigs into a single string (in an arbitrary order), constructs an accurate assembly graph from the resulting concatenate, uses reads to untangle this graph, and resolves bridged repeats (which are bridged by some reads in the repeat graph). Afterwards, it uses the repeat graph to resolve unbridged repeats (which are not bridged by any reads) using small differences between repeat copies and then outputs accurate contigs formed by paths in this graph. We benchmark Flye against five state-of-the-art SMS assemblers (Falcon, Miniasm, HINGE, Canu, and MaSuRCA) and show that it generates more accurate and contiguous assemblies and provides valuable information to aid in assembly finishing. Flye also reconstructs the mosaic structure of segmental duplications – a difficult problem even for finished genomes [Jiang et al., 2007, Pu et al., 2018].

2.3.2 Repeat graphs

Repeat graph construction. Repeats in a genome are often represented as pairwise local alignments and visualized as alignment-paths in a two-dimensional dot-plot of a genome. This pairwise representation is limited since it does not contribute to solving the repeat characterization problem [Pevzner et al., 2004, Bao and Eddy, 2002]. In contrast, the repeat graph compactly represents all repeats in a genome and reveals their mosaic structure [Pevzner et al., 2004, Jiang et al., 2007]. Assembly graph construction represents a special case of the repeat graph construction problem. Figure 2.2 outlines the algorithm for constructing the repeat graph of a finished (complete) genome. Flye applies this algorithm to construct the repeat graph of a pseudo-genome formed by concatenating all disjointigs (formed at the previous stage of the pipeline) in an arbitrary order. Below we explain why the resulting graph provides the correct representation of the assembled genome (as if it had been constructed from a complete genome) and describe additional algorithmic details.

Repeat characterization problem. Below we describe the abstract repeat characterization problem and explain how it relates to genome assembly. Consider a tour $T = v_1, v_2, ...v_n$ of

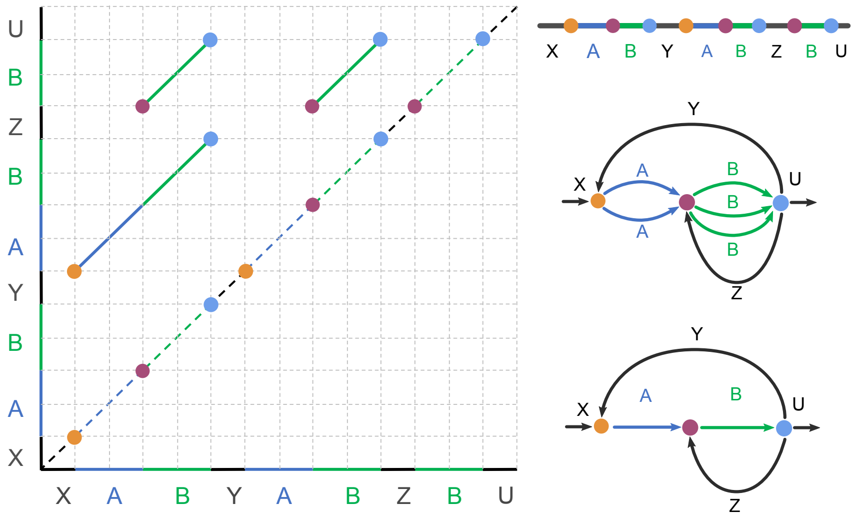


Figure 2.2: (a) Alignment-paths for all local self-alignments within a genome *XABYABZBU* formed by segments X, A, B, Y, Z, and U. Three instances of a mosaic repeat (*AB*, *AB*, and B) are represented as diagonal alignment-paths in the repeat plot. The self-alignment of the entire genome is shown by the main (dotted) diagonal. Alignment endpoints are clustered together if their projections on the main diagonal coincide or are close to each other (clusters of closely located endpoints for the distance threshold $d = 0$ are painted with the same color). For example, the right-most endpoints (shown in blue) of all three alignments form a single cluster because two of them have the same vertical projection and two of them have the same horizontal projection on the main diagonal. This clustering reveals three clusters (yellow, purple, and blue) with eight projections to the main diagonal. (b) Projections of the clustered endpoints on the main diagonal define eight vertices (breakpoints) that will be used for constructing the approximate repeat graph. (c) Breakpoints that belong to the same clusters are glued together. (d) Gluing parallel edges in the resulting graph produces the approximate repeat graph.

length n visiting all vertices of a directed graph G. We say that the i-th and j-th vertices in the tour T are equivalent if they correspond to the same vertex of the graph, that is, $v_i = v_j$. The set of all pairs of equivalent vertices forms a set of points (i, j) in a two-dimensional grid that we refer to as the repeat plot $Plot_T(G)$ of the tour T (Fig. 2.3). The transformation of a tour T traversing a known graph G into the repeat plot $Plot_T(G)$ is a simple procedure. Below, we address the reverse problem that is at the heart of genome assembly, repeat characterization and synteny block construction: given an arbitrary set of points $Plot$, in a two-dimensional grid, find a graph $G = G(Plot)$ and a tour T in this graph such that $Plot = Plot_T(G)$.

A dot-plot of a genome is a matrix that graphically represents all repeats in a genome [Gibbs

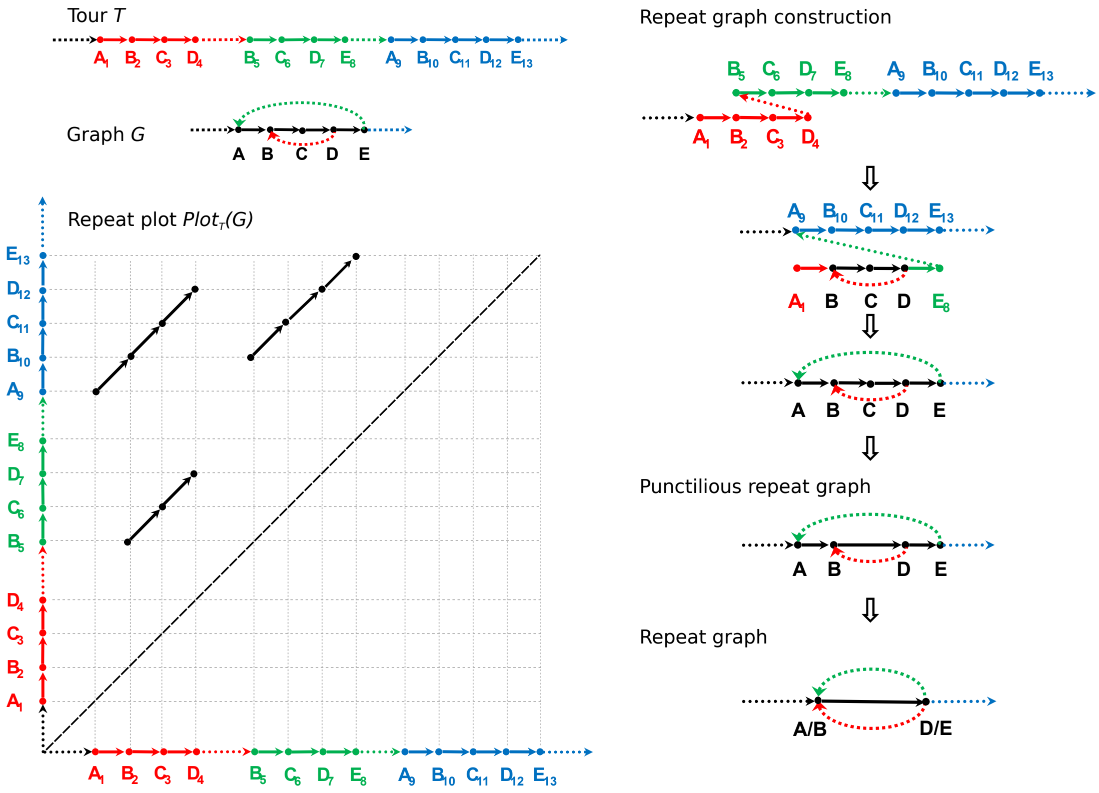

Tour T
A_1 B_2 C_3 D_4 B_5 C_6 D_7 E_8 A_9 B_{10} C_{11} D_{12} E_{13}
Graph G
A B C D E
Repeat plot $Plot_T(G)$
Repeat graph construction
B_5 C_6 D_7 E_8 A_9 B_{10} C_{11} D_{12} E_{13}
A_1 B_2 C_3 D_4
A_9 B_{10} C_{11} D_{12} E_{13}
A_1 B C D E_8
A B C D E
Punctilious repeat graph
A B D E
Repeat graph
A/B D/E

Figure 2.3: (a) A tour $T = ...A_1B_2C_3D_4...B_5C_6D_7E_8...A_9B_{10}C_{11}D_{12}E_{13}...$ in a graph G with red, green, and blue instances of a repeat that includes two copies of vertices A and E and three copies of vertices B, C, and D. Dots represent multiple vertices that appear before, between, and after these three instances of the repeat. The repeat plot $Plot_T(G)$ consists of three diagonals representing the three instances of the repeat in the tour. The trivial self-alignment of the entire genome against itself is shown by the main dotted diagonal (the points below this diagonal are not shown). Since vertex A in the graph is visited twice in tour T, it results in a single point (1, 9) in $Plot_T(G)$. Vertex B results in points (2, 5), (2, 10), and (5, 10); vertex C results in points (3, 6), (3, 11), and (6, 11); vertex D results in points (4, 7), (4, 12), and (7, 12); and vertex E results in the point (8, 13). (b) Constructing the punctilious repeat graph from the repeat plot by gluing vertices with indices i and j for each point (i, j) in the repeat plot. Each non-branching path in the graph is substituted by a single edge with length equal to the number of edges in this path. The lengths of the short edges (A,B) and (D,E) in the resulting graph are equal to 1 and the length of the long edge (B,D) is equal to 2 (for edge length threshold $d = 1$). The punctilious repeat graph (second graph from the bottom) is transformed into the repeat graph (bottom-most graph) by contracting the short edges (A,B) and (D,E).

and McIntyre, 1970]. In the case of repeat characterization, we are interested in the dot-plot *Plot* formed by non-overlapping alignment-paths representing all high-scoring local self-alignments of a genome against itself (below, we refer to these alignments as simply self-alignments). Each self-alignment reveals two instances of a repeat corresponding to contiguous segments x and y in the genome (x and y are called the spans of the alignment). Given a genome of length n and a set of its self-alignments *Plot*, the repeat characterization problem amounts to constructing a graph G and a tour T of length n in this graph (each segment of the genome corresponds to a subpath of the graph traversed by the tour) such that $Plot = Plot_T(G)$ and the tour T is alignment-compatible. A tour is alignment-compatible with respect to the dot-plot *Plot* if, for each alignment with spans x and y in Plot, paths in the graph corresponding to segments x and y coincide.

Generating the repeat plot of a genome. Our goal is to construct both the repeat graph of a genome and an alignment-compatible tour in this graph. Constructing the de Bruijn graph of a genome based on long k-mers will not solve this problem since the differences between imperfect repeat copies mask the repeat structure of the genome. Constructing the de Bruijn graph based on short k-mers will not solve this problem due to the presence of repeating short k-mers within long repeats (these k-mers lead to a tangled repeat graph). Thus, at the initial stage, Flye generates all self-alignments (repeats) of a genome and combines them into a repeat plot *Plot*. However, it is unclear how to solve the reverse problem of generating the repeat graph $G(Plot)$ of the genome.

To address this problem for a “genome” representing a concatenate of accurate short reads, a previous study [Pevzner et al., 2004] described various graph simplification procedures, for example, bubble and whirl removals, that are now at the heart of various short-read assemblers such as SPAdes [Bankevich et al., 2012]. However, it is not clear how to generalize these procedures to make them applicable to error-prone SMS reads. Below, we show how to modify the concept of a punctilious repeat graph [Pevzner et al., 2004] so that it can be applied to assembling SMS reads.

Constructing a punctilious repeat graph. Let *Alignments = Alignments(Genome, minOverlap)* be the set of all sufficiently long (of length at least *minOverlap*) self-alignments of a genome *Genome*. Flye sets the *minOverlap* parameter as the N90 of the read-set (the N90 of reads is the largest possible number *N* such that all reads of length *N* or longer have a total length of at least 90% of the total sequence; *minOverlap* varies from 3,000 to 5,000 nucleotides for the SMS datasets analyzed in this paper).

Given a set of self-alignments *Alignments* of a genome *Genome*, we construct the punctilious repeat graph *RepeatGraph(Genome, Alignments)* by representing *Genome* as a path consisting of $|Genome|$ vertices (Fig. 2.3) and by "gluing" each pair of vertices (positions in the genome) that are aligned against each other in one of the alignments in *Alignments* [Pevzner et al., 2004]. Gluing vertices v and w amounts to substituting them by a single vertex that is connected by edges to all vertices that either vertex v or vertex w was connected to. We consider branching vertices (that is, vertices with either in-degree or out-degree differing from 1) in the resulting graph and substitute each non-branching path between them by a single edge of length equal to the number of original edges in this path. Edges in the punctilious repeat graph are classified as long (longer than a predefined threshold d with default value 500 nucleotides) and short (Fig. 2.3).

The punctilious repeat graphs of real genomes are very complex due to various artifacts [Pevzner et al., 2004, Jiang et al., 2007]. For example, the starting/ending points of alignment-paths corresponding to three repeat copies starting at positions x, y, and z in the genome hardly ever start at points (x,y), (x,z), and (y,z) in the repeat plot. Because each repeat with $\binom{m}{2}$ copies in the genome results in pair-wise alignments and each of the corresponding $\binom{m}{2}$ alignment-paths may have unique starting (ending) vertices that differ from all other starting/ending positions, there will be many gluing operations for the starting (ending) positions of this repeat. Note that each of these operations may form a new branching vertex in the punctilious repeat graph. For example, gluing the endpoints of the three diagonals in Fig. 2.3 results in the branching vertices A, B, D, and E in the graph. Punctilious repeat graphs of real genomes often contain many branching vertices, making it difficult to compactly represent repeats. We address this challenge by transforming the punctilious repeat graph into a simpler graph.

From punctilious repeat graph to repeat graph. As described before, the endpoints of alignment-paths representing the same repeat might not be coordinated among all pair-wise alignments of this repeat. These uncoordinated alignments result in a complex repeat graph with an excessive number of branching vertices and many short edges (shorter than a threshold d). The repeat graph *RepeatGraph(Genome, Alignments, d)* is defined as the result of contracting all short edges in the punctilious repeat graph (Fig. 2.3). The contraction of an edge is the gluing of the endpoints of this edge, followed by the removal of the loop-edge resulting from this gluing. Since the genome represents a tour visiting all edges in the repeat graph, we define the multiplicity of an edge in the repeat graph as the number of times this edge is traversed in the tour. Edges of multiplicity 1 are called unique edges and all other edges are called repeats.

2.3.3 Approximate repeat graphs

Hiding graph artifacts. The described approach, although simple in theory, results in various complications in the case of real genomes, particularly in the case of inconsistent pair-wise alignments (see Appendix 2.7.5). In the case of short reads, various graph simplification procedures [Pevzner et al., 2004, Bankevich et al., 2012] result in a modified repeat graph that represents a more sensible repeat characterization, but sacrifice the fine details of some repeats in favor of revealing the mosaic structure shared by different repeat copies. However, in the case of SMS assemblies, repeat graph (and A-Bruijn graph) construction results in excessively complex graphs that make the previously proposed graph simplification algorithm for A-Bruijn graph construction [Pevzner et al., 2004] inefficient and make it difficult to select sensible parameters for graph simplification. For example, it is unclear how to select an adequate *bubble_size* parameter for bubble removal (small values of this parameter result in complex A-Bruijn graphs while large values result in oversimplified A-Bruijn graphs). While there exists a "sweet spot" for this parameter in short-read assembly, we were not able to find such a spot for long-read assembly. That is why we departed from the original A-Bruijn graph framework and opted to construct a different version of the repeat graph (called the approximate repeat graph) based only on the endpoints of diagonals in the genomic dot-plot rather than the entire diagonals as in a previous study [Pevzner et al., 2004]. This approach led to a great reduction in running time and allowed us to bypass the bubble/whirl-removal steps (and the challenge of choosing parameters for these operations) altogether.

Some branching vertices in the repeat graph arise from the contraction of multiple vertices in the punctilious repeat graph; for example, vertices A and B were contracted into a single vertex A/B in the repeat graph in Fig. 2.3. Consider the set of all vertices in the punctilious repeat graph that gave rise to branching vertices in the repeat graph (vertices A,B,D, and E in Fig. 2.3) and let *Breakpoints = Breakpoints(Genome, Alignments, d)* be the set of all positions in the genome that gave rise to these vertices (*Breakpoints* = $\{1,2,4,5,7,8,9,10,12,13\}$ in Fig. 2.3). This set of vertices forms a set of short, contiguous genomic segments (segments $(1,2)$, $(4,5)$, (710), and $(12,13)$ in Fig. 2.3) that contain all horizontal and vertical projections of the endpoints of all alignments in *Alignments*.

Flye approximates the set *Breakpoints* by recruiting all horizontal and vertical projections of the endpoints of alignments from *Alignments* to the main diagonal in the repeat plot. Figure 2.2 presents three alignments, resulting in eight projected points on the main diagonal. Two alignment endpoints are close if either of their projections on the main diagonal are located within a distance threshold d (including the case when a vertical projection of one endpoint coincides with or is close to a horizontal projection of another endpoint).

Flye clusters close endpoints together based on single linkage clustering. Applying this procedure (with $d = 0$) to eight breakpoints (projected endpoints) in Fig. 2.2 results in three clusters (breakpoints in the same cluster are painted with the same color). Figure 2.2 illustrates that gluing breakpoints that belong to the same clusters (and further collapsing parallel edges) results in an approximate repeat graph of the genome. However, although this procedure led to the correct repeat graph in the simple case shown in Fig. 2.2, the approximate repeat graph constructed based on the clustering of closely located breakpoints may differ from the repeat graph constructed based on the punctilious repeat graph. Appendix 2.7.6 illustrates that mosaic repeats and inconsistencies of local alignments may result in an “incorrect” clustering-based repeat graph. Below, we explain how Flye extends the set *Breakpoints* to address this complication.

Extending the set of breakpoints. As described above, Flye constructs the initial set *Breakpoints* by projecting all endpoints of the alignments (in the set of self-alignments *Alignments*) onto the main diagonal in the repeat plot. Each point in an alignment-path in the $|Genome| \times |Genome|$ grid has two projections (horizontal and vertical) on the main diagonal. Note that projections of some internal points in an alignment-path may belong to *Breakpoints*; for example, both projections of the middle point of the longest alignment-path in Fig. 2.2 (shown in purple) belong to *Breakpoints*. Such internal points should be reclassified as new alignment endpoints (by breaking the alignment-path into two parts) to avoid inconsistencies during the construction of the repeat graph. However, for some internal points, only one of their two projections belongs to *Breakpoints*, leading to complications in the path-breaking process. Below, we explain how to break the alignment-paths into subpaths (and, at the same time, extend the set *Breakpoints*) to address this complication.

A point in an alignment-path is called valid if both of its projections belong to *Breakpoints*, and invalid if only one of its projections belongs to *Breakpoints*. A set *Breakpoints* is called valid if all points in all alignment-paths are valid, and invalid otherwise. In the case that the constructed set *Breakpoints* is invalid, our goal is to add the minimum number of points to this set to make it valid. See Appenidx 2.7.6 for an example of an invalid point and a discussion on the importance of extending the set *Breakpoints* to make it valid.

Flye iteratively adds the missing projection for each invalid point to the set *Breakpoints* on the main diagonal until there are no invalid points left. Afterwards, it combines close points in *Breakpoints* into segments using single linkage clustering (as described above). The set of resulting segments (defined by their minimal and maximal positions on the main diagonal) forms a set *BpSegments*. Two segments from *BpSegments* are equivalent if there exists a point in one of the alignment-paths such that one of its projections on the main diagonal falls into the first segment and another falls into the second segment.

Each repeat of multiplicity m typically corresponds to *m* segments in *BpSegments* corresponding to m starting positions of this repeat in the genome (and the same number of segments corresponding to its ending positions). Note that the number of breakpoint segments resulting from this repeat is reduced as compared with the number of breakpoints, which can be as large as $\binom{m}{2}$ for the starting positions of the repeat (and the same number for its ending positions). Flye takes advantage of this reduction by selecting middle points of each breakpoint segment and only gluing these middle points rather than all breakpoints. Essentially, it redefines the endpoints of each alignment-path as the middle points of corresponding breakpoint segments.

Specifically, Flye constructs the approximate repeat graph by generating the set *BpSegments*, selecting a middle point from each segment in *BpSegments*, and gluing the two middle points for every pair of equivalent segments. Afterwards, it glues together parallel edges (edges that start and end at the same vertices) if the genome segments corresponding to these edges are aligned in *Alignments*, that is, if there exists an alignment with its *x*- and *y*-spans overlapping both these segments. For brevity, below we refer to the approximate repeat graph resulting from this procedure simply as the repeat graph.

2.3.4 Constructing repeat graphs from long reads

From the repeat graph of a genome to the assembly graph of contigs. The ABruijn assembler [Lin et al., 2016] constructs a set of contigs but stops short of constructing the repeat graph of a genome based on these contigs (Appenidx 2.7.7 describes the challenge of assembling contigs into a repeat graph). The contig construction in ABruijn essentially amounts to finding extension reads for extending paths in the (unknown) repeat graph of the genome. Each extension read increases the length of the growing path until the extension process becomes ambiguous, that is, when it reaches a branching vertex in the (unknown) repeat graph. Afterwards, ABruijn decides whether to continue or to stop the path extension to avoid assembly errors. Since ABruijn does not know the exact locations of branching vertices, it uses the last extension read to extend the path beyond the branching vertex by at least *minOverlap* nucleotides. As a result, each linear contig constructed by ABruijn satisfies the overhang property: it extends by at least *minOverlap* nucleotides before the first branching vertex and after the last branching vertex it traverses. Note that the same *minOverlap* value is used during repeat graph construction.

Constructing disjointigs. ABruijn and other existing SMS assemblers invest substantial time into making sure that generated contigs are correctly assembled (represent subpaths of the genomic tour in the repeat graph). In contrast to ABruijn, Flye does not attempt to construct accurate contigs at the initial assembly stage but instead generates disjointigs as arbitrary paths in the (unknown) repeat graph of the genome. However, it constructs an accurate repeat graph (assembly graph) from error-prone disjointigs.

Flye randomly walks in the (unknown) assembly graph to generate random paths from this graph. Each non-chimeric read from *Reads* defines a subpath of a genomic tour in an assembly graph. Flye extends this path by switching from the current read to any other overlapping read (with sufficiently long common jump-subpath) rather than a carefully chosen overlapping read [Lin et al., 2016], avoiding a time-consuming test to check whether this selection triggers an assembly error.

Since the resulting FLYEWALK algorithm (see Appenidx 2.7.8) does not invoke the contig correctness check, it constructs paths (chains of overlapping reads) that do not necessarily follow the genome tour through the assembly graph. Although it may appear counter-intuitive that inaccurate disjointigs constructed by FLYEWALK result in an accurate assembly graph, note that inaccurate paths (disjointigs) in the de Bruijn graph (a special case of the assembly graph) certainly result in an accurate assembly graph. Indeed, an assembly graph constructed from arbitrary paths in a de Bruijn graph is the same as the original de Bruijn graph (as long as these paths include all *k*-mers from the assembly graph). See Appendix 2.7.9 for additional details.

Constructing the assembly graph from disjointigs. Similarly to ABruijn, Flye generates disjointigs satisfying the overhang property, which, as will be explained below, represents an important condition for constructing the repeat graph. Flye further concatenates all disjointigs (separated by delimiters) in an arbitrary order into a single string Concatenate. It further uses the longest jump-subpath approach [Lin et al., 2016] to generate the set *Alignments* of all sufficiently long self-alignments within the resulting concatenate and constructs the assembly graph as the repeat graph of the concatenate *RepeatGraph(Concatenate, Alignments, d)*.

It has been shown that the repeat graph of concatenated accurate reads (when alignments between reads do not extend beyond delimiters in the concatenate of all reads) approximates the repeat graph of the genome [Pevzner et al., 2004]. Appendix 2.7.10 demonstrates that the assembly graph constructed from inaccurate disjointigs also approximates the repeat graph of the genome.

Figure 2.4 (left) presents the assembly graph of the SMS reads from an *Escherichia coli* genome. Flye further untangles this graph into a graph with just six edges (Fig. 2.4, middle) as described below.

2.3.5 Simplifying repeat graph

Resolving bridged repeats in the assembly graph. Flye aligns all reads to the constructed assembly graph (see Appendix 2.7.11) and uses them to identify the repeat edges in this graph (see Appendix 2.7.12). It further transforms the assembly graph into the condensed assembly graph by contracting all of its repeat edges. Aligning a read to the assembly graph induces its alignment to the condensed assembly graph, and we focus on bridging reads that align to multiple edges in the condensed assembly graph. Untangling incident edges $e = (w, v)$ and $f = (v, u)$ in the condensed assembly graph amounts to substituting them by a single edge (w, u). Below, we describe how Flye uses bridging reads to untangle the condensed assembly graph and how this untangling contributes to resolving repeats in the assembly graph.

A bridging read in the condensed assembly graph is called an (e, f)-read if it traverses two consecutive edges e and f in this graph. For each pair of incident edges e and f in the condensed assembly graph, we define $\mathit{transition}(e, f)$ as the number of (e, f)-reads plus the number of (f', e')-reads, where e' and f' are complementary edges for e and f, that is, edges representing a complementary strand.

Given a set of bridging reads in the condensed assembly graph, we construct a transition graph as follows. Each edge e in the condensed assembly graph corresponds to vertices e_h and e_t in the transition graph, representing the head (start) and tail (end) of e, respectively. A complementary edge for e corresponds to the same vertices, but in the opposite order. Each (e, f)-read defines an undirected edge between e_t and f_h in the transition graph with weight equal to $\mathit{transition}(e, f)$.

Note that the transition graph is bipartite for the simple case when the two subgraphs of the condensed assembly graphs, corresponding to complementary strands, do not share vertices. However, it is not necessarily bipartite in the case of genomes that contain long inverted repeats. Flye thus applies Edmonds algorithm [Edmonds and Johnson, 1973] to find a maximum weight matching in the transition graph and uses this matching for untangling the condensed assembly graph. For each edge (e_t, f_h) in the constructed matching, Flye additionally checks the confidence of the transition between edges e and f (see Appendix 2.7.13 for details) and untangles e and f for each edge (e_t, f_h) in the transition graph that passes this check. Flye iteratively untangles edges in the condensed assembly graph and performs the corresponding iterative repeat resolution in the assembly graph.

Note that consecutive edges e and f in the condensed assembly graph are not necessarily consecutive in the assembly graph. Thus, after Flye untangles e and f, it uses one of the bridging (e, f)-reads to fill the gap between the end of e and the start of f in the assembly graph. Afterwards, most repeat edges in the assembly graph either represent long unbridged repeat edges (that are not bridged by any reads) or form paths consisting of repeat edges with total lengths typically exceeding the median read length.

Resolving unbridged repeats in the assembly graph. Flye utilizes the constructed repeat graph for the resolution of unbridged repeats. Resolving unbridged and nearly identical repeats using SMS reads is a difficult problem since error-prone SMS reads make it difficult to distinguish repeat copies with divergence below 10%. As a result, SMS assemblers often fail to resolve unbridged repeats, which are common even in bacterial genomes [Kamath et al., 2017, Schmid et al., 2018]. This challenge is related to the challenge of constructing phased diploid genome assemblies [Chin et al., 2016] and overlap-filtering for repeat resolution [Koren et al., 2017]. The repeat graph constructed by Flye offers an approach for resolving unbridged repeats based on analyzing the topology of the repeat graph.

Figure 2.4 shows an unbridged repeat *REP* as an edge in the assembly graph. It would be impossible to resolve this repeat (that is, to pair each incoming edge into the initial vertex of *REP* with the corresponding outgoing edge from the terminal vertex of *REP*) if its two copies were identical. However, since there exist variations between these copies, it becomes possible to transform the single sequence *REP* into two different repeat instances, REP_1 and REP_2 , as shown in Fig. 2.4. Below we describe how Flye resolves unbridged repeats by (1) identifying variations between repeat copies, (2) matching each read with a specific repeat copy using these variations, and (3) using these reads to derive a distinct consensus sequence for each repeat copy.

Flye takes advantage of the small variations between different repeat copies to resolve unbridged repeats. It identifies the variations between repeat copies, matches each read with a specific repeat copy using these variations, and uses these matched reads to derive a distinct consensus sequence for each repeat copy. The success of this approach is contingent on the presence of a sufficiently large number of variations between the different repeat copies. Therefore, the first step is to estimate the number and positions of variations between the repeat copies and to calculate the divergence of the various repeat copies from reads alone. Appendix 2.7.14 describes how Flye calculates the divergence between repeat copies. The current version of Flye is limited

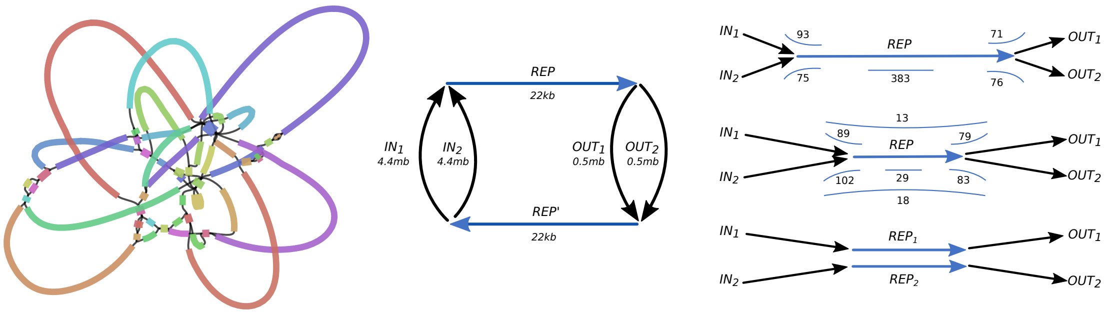


Figure 2.4: (a) An assembly graph of SMS reads from the *E. coli* strain EC9964 genome visualized with Bandage [Wick et al., 2015]. (b) The untangled assembly graph (after resolving bridged repeats in the graph on the left) contains a single unbridged repeat *REP* (and its complement REP') of length 22 kb. The incoming edges into the initial vertex (outgoing edges from the terminal vertex) of edge *REP* are denoted IN_1 and IN_2 (OUT_1 and OUT_2). Two complementary strands are fused together in a single connected component. It is unclear whether the genome traverses the assembly graph as $IN_1 \rightarrow REP \rightarrow OUT_1 \rightarrow REP'$ or as $IN_1 \rightarrow REP \rightarrow OUT_2 \rightarrow REP'$. (c) A total of 93, 71, 75, and 76 reads traverse both IN_1 and *REP*, IN_2 and *REP*, *REP* and OUT_1, and *REP* and OUT_2, respectively. The span of 383 reads falls entirely within edge *REP*. (d) After assigning 93 reads that traverse both IN_1 and *REP* to the first repeat copy, and 71 reads that traverse both IN_2 and *REP* to the second repeat copy, we "move forward" into the repeat and construct two differing consensus sequences for a 8.6-kb-long prefix of *REP* with divergence 9.8% (two consensus sequences for a 6.8-kb-long suffix of *REP* when we "move backward" into the repeat). The length of the repeat edge is reduced to 22.0 8.6 6.8 = 6.6 kb, resulting in the emergence of 13 + 18 = 31 spanning reads for this repeat, all of them supporting a cis transition (IN_1 with OUT_1 and IN_2 with OUT_2). (e) Resolved instances of the repeat with consensus sequences REP_1 and REP_2 and divergence 6.9%.

to resolving unbridged repeats of multiplicity two in both haploid (for example, bacterial) and diploid (for example, human) genomes.

The idea of the algorithm is to assign each read to a specific repeat copy and then use the assigned reads to derive a distinct consensus sequence for each repeat copy. Figure 2.4 shows an example in which the 93 reads that traverse edges IN_1 and *REP* can be assigned to one repeat copy and the 75 reads that traverse edges IN_2 and *REP* can be assigned to another repeat copy. However, it is unclear how to assign other reads mapping to the edge REP to a specific repeat copy. Flye uses reads starting in the incoming edges (93 and 75 reads in Fig. 2.4) to "move forward" into the repeat and construct two different prefixes of the repeat *REP* corresponding to the two copies of the repeat. In parallel, it uses reads ending in the outgoing edges (71 and 76 reads in Fig. 2.4) to “move backward” into the repeat and construct two different suffixes of this repeat.

In each iteration of the algorithm, reads are assigned to a specific repeat copy, and then all of the reads assigned to each repeat copy are used to construct a consensus sequence for that copy. Thus, as the algorithm proceeds, more reads are assigned to specific repeat copies and the consensus sequence for each repeat copy grows longer. The algorithm terminates when no new reads can be assigned to read copies and the consensus sequences stop growing in length. There are two goals: to obtain distinct consensus sequences for each repeat copy and to determine the correct pairings of incoming and outgoing edges for each repeat copy. Appendix 2.7.15 describes each successive iteration of the algorithm in detail.

2.4 Results

Benchmarking Flye. We benchmarked Flye against SMS assemblers Canu, Falcon, HINGE, Miniasm, and MaSuRCA using six datasets. We used QUAST [Mikheenko et al., 2018] to evaluate all assemblers (Appendix 2.7.2). Since Miniasm returns assemblies with a much larger number of mismatches and indels than other assemblers, it is not well suited for a reference-based quality evaluation with QUAST. To make a fair comparison, we ran the ABruijn contig-polishing module [Lin et al., 2016] on the Miniasm output to improve the accuracy of its contigs (referred to as Miniasm + ABruijn).

2.4.1 Benchmarking with the Bacteria dataset

The dataset consists of 21 sets of Pacific Biosciences (PacBio) reads from the National Collection of Type Cultures (NCTC). These NCTC sets were studied in detail in [Kamath et al., 2017] and used to benchmark various assemblers. We only benchmarked Flye against HINGE on these datasets, since HINGE outperformed the other assemblers on bacterial genomes [Kamath et al., 2017]. We ignored small connected components in the bacterial assembly graphs (which represent plasmids that do not share repeats with chromosomes) and classified an assembly as (1) complete if the assembly graph consists of a single loop-edge representing a circular chromosome, (2) semicomplete if the assembly graph contains multiple edges but there exists a single Chinese postman tour in this graph [Edmonds, 1965], and (3) tangled if the assembly graph is neither complete nor semicomplete.

While HINGE does not distinguish between complete and semi-complete assemblies, we argue that ignoring this separation may lead to assembly errors. Indeed, a single Chinese postman tour in a semicomplete assembly graph results in a unique assembly only in the case of unichromosomal genomes without any plasmids that share repeats with the chromosome (repeat-sharing plasmids). In the case of multichromosomal genomes or in the case of repeat-sharing plasmids, there exist multiple possible assemblies from a semicomplete assembly graph. Since 10% of known bacterial genomes are multichromosomal and since a large fraction of unichromosomal genomes have repeat-sharing plasmids [Antipov et al., 2015], the assumption that a semicomplete assembly graph results in a complete genome reconstruction may lead to errors.

Before resolving unbridged repeats, Flye assembled the genomes from the Bacteria dataset into 4 complete, 1 semicomplete, and 16 tangled assembly graphs. After resolving unbridged repeats, the Flye assemblies resulted in 8 complete, 5 semicomplete, and 8 tangled assembly graphs with the number of edges varying from 3 to 25. Figure. 2.5 shows examples of assembly graphs generated by Flye and HINGE, and Table 2.1 illustrates that Flye and HINGE generated very similar assemblies.

2.4.2 Benchmarking with the Metagenome dataset

The Metagenome dataset consists of Pacific Biosciences reads from a synthetic community of 20 bacteria. Since 3 of 20 bacterial genomes in the metagenomic sample had coverage

Table 2.1: A comparison of Flye and HINGE on bacterial genomes from the Bacteria dataset. HINGE results were reproduced from [Kamath et al., 2017]. Tangled* stands for Tangled/Lacks Circularization. "n/a" indicates that the assembly graph is not complete and has no unbridged repeats of multiplicity two.

dataset	bacterial species	Flye	Flye + unbridged repeat resolution	HINGE
EC4450	Escherichia coli	Tangled	n/a	Tangled
KP5052	Klebsiella pneumoniae	Tangled	Tangled	Tangled
SA6134	Staphylococcus aureus	Complete	n/a	Complete
EC7921	Escherichia coli	Tangled	Complete	Complete
EC8333	Escherichia coli	Tangled*	n/a	Tangled
EC8781	Escherichia coli	Tangled	n/a	Tangled
EC9002	Escherichia coli	Complete	n/a	Complete
EC9006	Escherichia coli	Tangled	Tangled	Tangled
EC9007	Escherichia coli	Tangled	Tangled	Tangled
EC9012	Escherichia coli	Tangled	Tangled	Complete
EC9016	Escherichia coli	Tangled	Tangled	Tangled
EC9024	Escherichia coli	Tangled	n/a	Tangled
EC9103	Escherichia coli	Complete	n/a	Complete
KP9657	Klebsiella pneumoniae	Tangled	n/a	Tangled
EC9664	Escherichia coli	Tangled	Complete	Tangled
EC10864	Escherichia coli	Tangled	n/a	Complete
EC11022	Escherichia coli	Tangled	Semi-complete	Complete
KS11692	Klebsiella sp	Tangled	n/a	Complete
SA11962	Staphylococcus aureus	Tangled	Tangled	Tangled
KP12158	Klebsiella planticola	Semi-complete	n/a	Complete
KC12993	Kluyvera cryocrescens	Complete	n/a	Complete

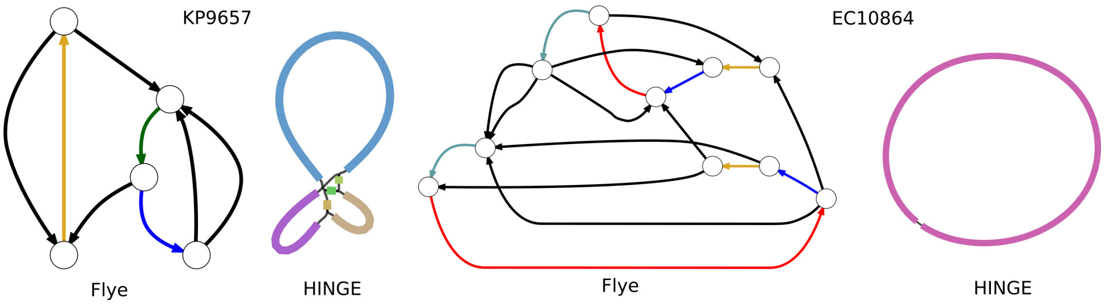


Figure 2.5: A comparison of Flye and HINGE assembly graphs on bacterial genomes from the Bacteria dataset. (Left) Flye and HINGE assembly graphs of the KP9657 dataset. There is a single unique edge entering into (and exiting) the unresolved yellow repeat and connecting it to the rest of the graph. Thus, this repeat can be resolved if one excludes the possibility that it is shared between a chromosome and a plasmid. In contrast to HINGE, Flye does not rule out this possibility and classifies the yellow repeat as unresolved. (Right) The Flye and Hinge assembly graphs of the EC10864 dataset show a mosaic repeat of multiplicity four formed by yellow, blue, red and green edges (the two copies of each edge represent complementary strands). HINGE reports a complete assembly into a single chromosome. .

below 1× (*Methanobrevibacter smithii, Candida albicans*, and *Streptococcus pneumoniae*), they were excluded from the benchmarking analysis. Since other assemblers performed poorly on the Metagenome dataset, we limited our benchmarking to Flye and Canu, which assembled this dataset with NGA50 = 1,277 kb (84 misassemblies) and NGA50 = 1,061 kb (99 misassemblies), respectively (see Table 2.2). Appendix 2.7.3 illustrates that most misassemblies in the Metagenome dataset probably represent differences between the genomes in the Metagenome sample and the reference genomes rather than real misassemblies.

Flye performed better than Canu for five genomes and Canu performed better that Flye for four genomes. In particular, Flye produced a better assembly of *Rhodobacter sphaeroides*, which has the lowest coverage (24×) among the 17 analyzed genomes (NGA50 = 2 Mb for Flye, compared with 54 kb for Canu). Comparison between the metagenome assemblies and the inferred isolate assemblies (from reads matched to the reference genomes) suggests that our metagenomics assemblies could be further improved by a better handling of datasets with uneven coverage.

2.4.3 Benchmarking with the Yeast dataset

The Yeast dataset contains PacBio and Oxford Nanopore Technology (ONT) reads from the *Saccharomyces cerevisiae* S288c genome of length 12.1 Mb at 30× coverage [Giordano et al., 2017]. Similarly to the original study, we used the full set of ONT reads in the Yeast-ONT dataset (30× coverage) but downsampled the PacBio reads from the original 120 coverage to 30 in the Yeast-PacBio dataset to have their coverage distribution be similar to the ONT data. Assembling this dataset with the original 120 coverage results in better assemblies; for example, the NGA50 increased from 560 kb to 732 kb for the Flye assembly (Flye fully assembled 14 of 16 yeast chromosomes). Table 2.2 illustrates that all of the assemblers tested except HINGE produced Yeast-PacBio assemblies with similar NGA50 values ranging from 560 kb for Flye to 603 kb for Canu (HINGE resulted in a lower NGA50 of 361 kb). Flye generated the most accurate assembly

Table 2.2: Assembly statistics for the Yeast, Worm, Human, and Human+ datasets generated using QUAST. The NG50 of an assembly is the largest possible number *L*, such that all contigs of length *L* or longer cover at least 50% of the genome. Given an assembled set of contigs and a reference genome, a corrected assembly is formed by breaking each erroneously assembled contig at its breakpoints, resulting in shorter contigs 19 . The NGA50 of an assembly is defined as the NG50 of its corrected assembly. The minimum contig size was set to 5 kb for the Yeast and Worm assemblies and to 50 kb for the Human assemblies. The human reference was modified by masking the low-complexity centromere regions of the chromosomes.

Dataset	Assembler	Length	No. contigs	NG50 (kb)	Ref. coverage, %	Ref. identity, %	No. misassemblies	NGA50 (kb)
Yeast-PacBio	Flye	12.1	28	670	98.3	99.95	5	560
	Canu	12.4	33	708	99.5	99.95	13	603
	Falcon	12.1	42	562	97.5	99.91	27	562
	HINGE	12.2	45	440	91.9	98.81	19	361
	Miniasm + ABruijn	12.2	36	600	98.2	99.93	11	592
Yeast-ONT	Flye	12.1	28	810	98.7	99.04	9	660
	Canu	12.2	41	800	99.1	98.96	18	655
	Falcon	11.9	41	662	97.4	98.81	17	637
	HINGE	12.2	64	309	92.5	97.94	59	292
	Miniasm + ABruijn	11.6	24	723	98.8	99.03	12	723
Worm	Flye	103	85	3,256	99.5	99.93	111	1,893
	Canu	108	175	2,954	99.7	99.93	190	1,974
	Falcon	101	106	2,291	98.7	99.78	118	1,242
	HINGE	103	64	2,710	98.0	99.40	174	1,441
	Miniasm + ABruijn	108	178	2,314	99.6	99.93	181	1,437
Human	Flye + Pilon	2,776	1,069	7,886	96.4	99.70	879	6,349
	Canu + Pilon	2,730	2,195	3,209	95.4	99.49	1,200	2,870
	MaSuRCA	2,768	1,269	4,670	95.1	99.84	1,500	3,812
Human+	Flye + Pilon	2,823	782	18,181	97.0	99.69	1,487	11,800
	Canu + Pilon	2,815	798	10,410	96.8	99.81	1,455	7,007
	MaSuRCA	2,876	1,111	8,425	97.5	99.80	2,101	5,581

with 5 errors (versus 13 errors for Canu). Although Miniasm generated an assembly with only 90% sequence identity, Miniasm + ABruijn contigs had 99.93% accuracy. Canu and Flye resulted in assemblies with the highest sequence identity (above 99.95%).

The Yeast-ONT assemblies show a similar trend, with all assemblers except HINGE producing similar NGA50 values ranging from 637 kb (Falcon) to 723 kb (Miniasm). Flye generated the most accurate assembly with 9 errors (18 errors for Canu). Figure 2.6 shows the assembly graph generated by Flye.

2.4.4 Analyzing the Worm dataset

The Worm dataset contains PacBio reads from the *Caenorhabditis elegans* genome of length 100 Mb at 40× coverage. Flye and Canu produced the most contiguous assemblies

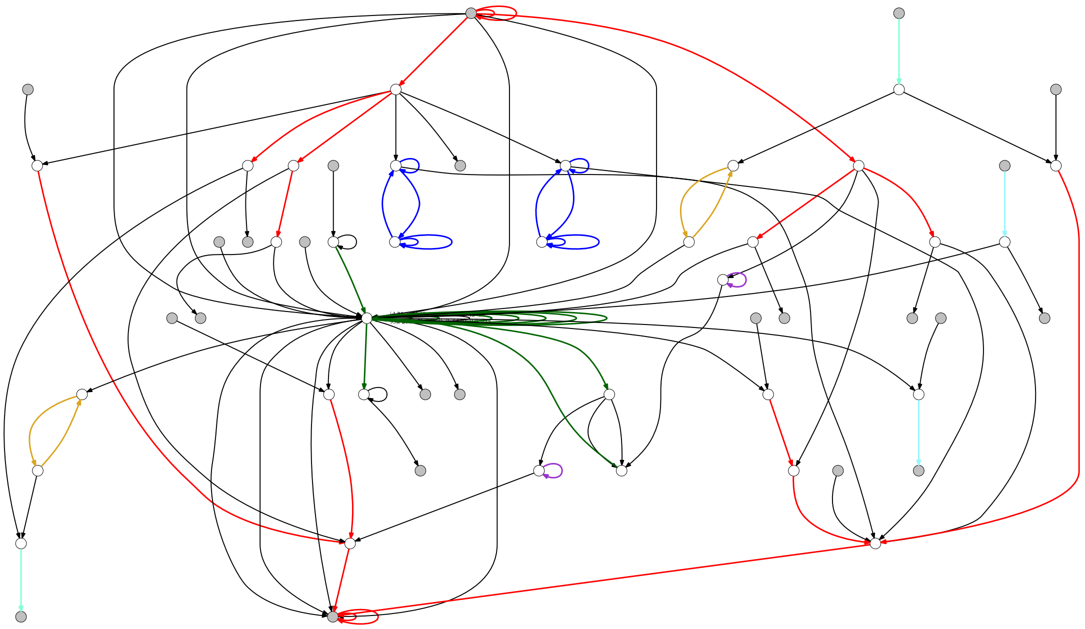


Figure 2.6: The assembly graph of the Yeast-ONT dataset. Edges that were classified as repetitive by Flye are shown in color, while unique edges are black. Flye assembled the Yeast-ONT dataset into a graph with 21 unique and 34 repeat edges and generated 21 contigs as unambiguous paths in the assembly graph. A path $v_1,...v_i,v_{i+1}...v_n$ in the graph is called unambiguous if there exists a single incoming edge into each vertex of this path before v_{i+1} and a single outgoing edge from each vertex after v_i. Each unique contig is formed by a single unique edge and possibly multiple repeat edges, while repetitive contigs consist of the repetitive edges which were not covered by the unique contigs. The visualization was generated using the graphviz tool (http://graphviz.org).

(NGA50 = 1,893 kb and 1,974 kb, respectively). However, Canu showed an increased number of misassemblies (190) compared with Flye (111) and Falcon (118). Flye was faster than Canu and Falcon in assembling the Worm dataset (128, 780, and 945 minutes of wall clock time, respectively (see Appendix 2.7.2 for more details). With an increase in genome size, Flye achieves close to an order of magnitude speed-up as compared with Canu: for example, 140 versus 1,100 hours to assemble the *Drosophila melanogaster* genome. This speed-up highlights the advantages of skipping the time-consuming read-correction step and replacing conventional contig generation with the much more rapid generation of disjointigs.

Since inferring the length of long tandem repeats is a difficult problem in short-read assembly, tandem repeats in many reference genomes might be misassembled. Figure 2.7

demonstrates that Flye improves on other long-read assemblers in reconstructing tandem repeats and reveals that some differences between the Flye assembly and the reference *C. elegans* genome probably represent differences with the reference rather than misassemblies by Flye.

2.4.5 Analyzing the Human and Human+ datasets

The Human dataset contains ONT reads from the GM12878 human cell line at 30× coverage complemented by a set of short Illumina reads at 50× coverage. The Human+ dataset combines the Human dataset with a dataset of ultra-long ONT reads (those with reads N50 $>$ 100 kb; that is, 50% of the total sequence data in reads longer than 100 kb) at 5× coverage [Jain et al., 2018]. Since Canu improved on Falcon and Miniasm in assembling large genomes [Koren et al., 2017], we only benchmarked Flye against Canu for the human genome datasets. The Canu Human assembly was generated in [Jain et al., 2018], and the assembly of the Human+ dataset was later updated by the authors using the latest Canu 1.7 version. We also analyzed hybrid MaSuRCA assemblies of the Human and Human+ datasets [Zimin et al., 2017], which are available from the MaSuRCA website.

Currently, the ONT assemblies have many base-calling errors (the Flye and Canu Human assemblies had 1.2% and 2.8% error rate, respectively) because of the biased error pattern in ONT reads. Although the Nanopolish tool contributed to a reduction in the base-calling errors of the ONT assemblies [Simpson et al., 2017], the resulting error rates still an order of magnitude higher than the error rates of Illumina or PacBio assemblies. Since most errors in the ONT assemblies are frameshift-introducing indels, they are particularly problematic for downstream applications.

To mitigate the high error rates of these ONT assemblies, we used Pilon [Walker et al., 2014] in the indel correction mode to polish Flye and Canu assemblies using Illumina reads. Although such polishing reduced the error rates (to 0.30% for Flye + Pilon and to 0.51% for Canu + Pilon), we note that Illumina-based read correction of ONT assemblies has limitations, especially for repetitive regions with low short-read mappability.

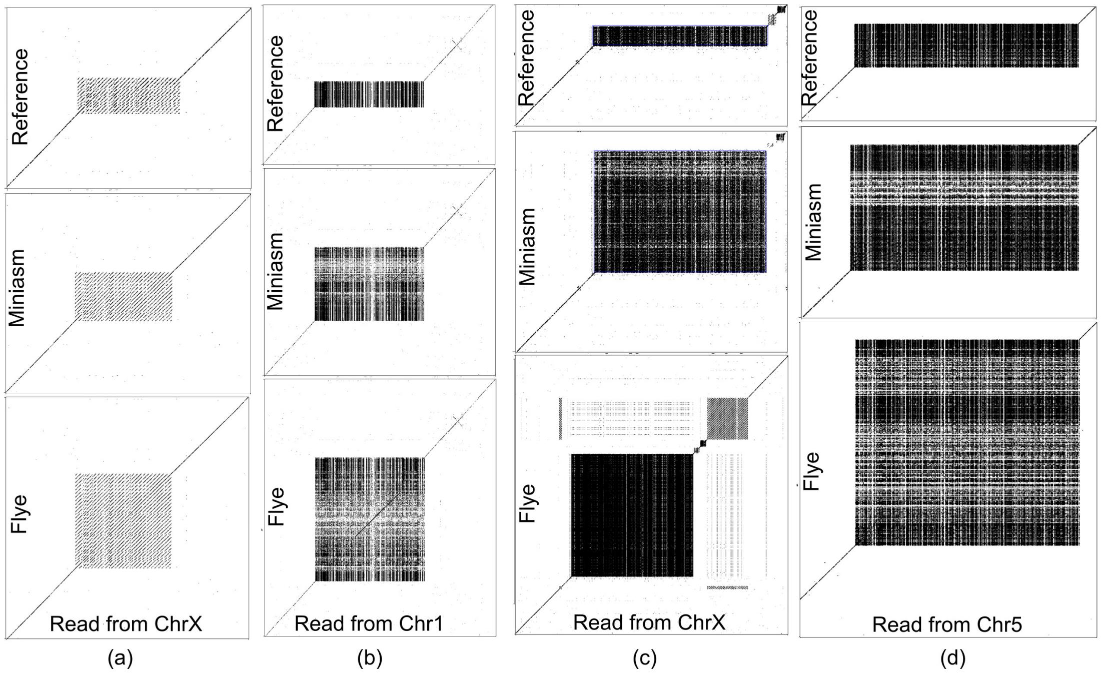


Figure 2.7: Dot-plots showing the alignment of reads against the Flye assembly, the Miniasm assembly and the reference *C. elegans* genome. (a) The reference genome contains a tandem repeat of length 1.9 kb (10 copies) on chromosome X with the repeated unit having length 190 nucleotides. In contrast, the Flye and Miniasm assemblies of this region suggest a tandem repeat of length 5.5 kb (27 copies) and 2.8 kb (13 copies), respectively. 15 reads that span over the tandem repeat support the Flye assembly (the mean length between the flanking unique sequence matches the repeat length reconstructed by Flye) and suggests that the Flye length estimate is more accurate. (b) The reference genome contains a tandem repeat of length 2 kb on chromosome 1. In contrast, the Flye and Miniasm assemblies of this region suggest a tandem repeat of length 10 kb and 5.6 kb, respectively. A single read that spans over the tandem repeat supports the Flye assembly. Since the mean read length in the WORM dataset is 11 kb, it is expected to have a single read spanning a given 10.0 kb region but many more reads spanning any 5.6 kb region (as implied by the Miniasm assembly) or 2.0 kb region (as implied by the reference genome). Six out of 23 reads cross the “left” border (two out of 18 reads cross the right border) of this tandem repeat by more than 5.6 kb, thus contradicting the length estimate given by Miniasm and suggesting that the Flye length estimate is more accurate. (c) The reference genome contains a tandem repeat of length 3 kb on chromosome X. In contrast, the Flye and Miniasm assemblies of this region suggest a tandem repeat of lengths 13.6 kb and 8 kb, respectively. A single read that spans over the tandem repeat reveals the repeat cluster to be of length 12.2k, which suggests that the Flye length estimate is more accurate. (d) The reference genome contains a tandem repeat of length 1.5 kb on chromosome 1. In contrast, the Flye and Miniasm assemblies of this region suggest tandem repeats of length 17 kb and 4.3 kb, respectively. One read that spans over the tandem repeat reveals the repeat cluster to be of length 18.0 kb, which suggests that the Flye length estimate is more accurate.

It turns out that Flye assembled a larger fraction of the human genome (96.4%) than Canu (95.4%) and MaSuRCA (95.1%). Interestingly, Flye and MaSuRCA, in contrast to Canu, assembled some difficult-to-assemble, low-complexity centromeric chromosome regions, which are hard to benchmark using reference-based methods. To provide a fair comparison between all three assemblers using QUAST, we thus modified the hg38 reference by masking the centromeric regions using the coordinates from the UCSC Genome Browser.

For the Human dataset, Flye, MaSuRCA, and Canu generated assemblies with NGA50 values equal to 6.35 Mb (879 assembly errors), 3.81 Mb (1,500 assembly errors), and 2.87 Mb (1,200 assembly errors), respectively. The MaSuRCA assembly had a slightly higher percentage identity with the reference (99.84% compared with 99.70% for Flye + Pilon and 99.49% for Canu + Pilon). For the Human+ dataset, Flye, Canu, and MaSuRCA generated assemblies with NGA50 values equal to 11.8 Mb (1,487 assembly errors), 7 Mb (1,455 assembly errors), and 5.6 Mb (2,101 assembly errors), respectively. As expected, incorporating ultra-long ONT reads resulted in a more contiguous assembly for all assemblers.

2.4.6 Segmental duplications in the human genome

The repeat graph constructed by Flye reveals the complex mosaic structure of segmental duplications. Flye classifies all edges in the graph into unique and repeat edges by analyzing how reads traverse the graph and by using coverage-based arguments. After removing all unique edges from the assembly graph, only the connected components formed by repeat edges remain, which reveal the segmental duplications encoded by the repeat edges in the graph. We define the complexity (length) of a segmental duplication as the number (total length) of edges in its connected component. Figure 2.8 (left) illustrates a mosaic segmental duplication of complexity 7 and length 25.7 kb (the 7 colored repeat edges form a connected component in the Flye assembly graph after removing all of the unique edges). A segmental duplication is classified as simple if its complexity is 1 and mosaic otherwise [Jiang et al., 2007, Pu et al., 2018]. Figure 2.8 (right) shows the distributions of lengths and complexities of segmental duplications identified by Flye and illustrates the power of the assembly graph for repeat resolution.

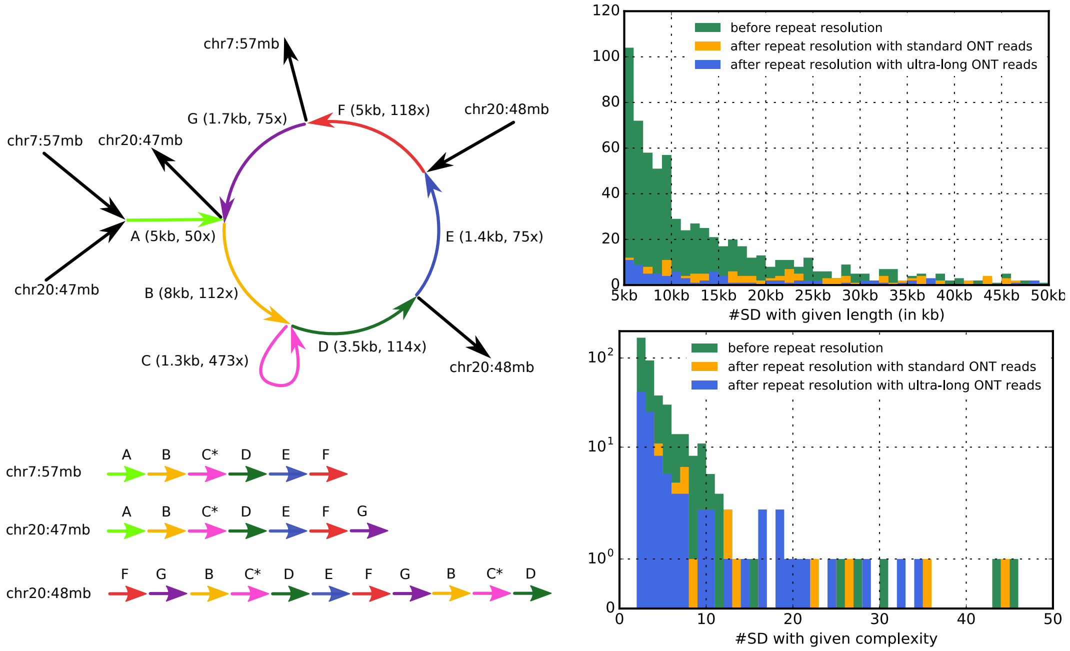


Figure 2.8: (a) A mosaic segmental duplication (SD) of complexity 7 represented as a connected component formed by repeat edges (7 colored edges of total length 25.7 kb) in the assembly graph of the Human dataset (flanking unique edges shown in black). The loop-edge C with coverage 473× represents a tandem repeat $C^\star$ with unit length 1.3 kb that is repeated 19 times. The colored edges of the assembly graph align to a region on chromosome 7 of length 31 kb and two regions on chromosome 20 of lengths 30 kb and 46 kb. These three instances of SDs were not resolved using standard ONT reads but were resolved using ultra-long reads in a way that is consistent with the reference human genome. (b) Statistics are given before resolving bridged repeats (green), after resolving bridged repeats with standard ONT reads (orange), and with ultra-long ONT reads (blue). Only SDs between 5 kb and 50 kb in length and with complexity between 2 and 50 contributed to the SD length and SD complexity histograms. Only two SDs have complexity exceeding 50 before bridged repeat resolution. Of the 688 SDs between 5 kb and 50 kb, 545 were resolved using the standard ONT reads, and ultra-long reads resolved an additional 58 SDs. There were 1,256 simple SDs before bridged repeat resolution and 143 after bridged repeat resolution with ultra-long reads. Since Flye usually resolves SDs shorter than the typical read length, the SDs identified by Flye do not include many known human SDs.

There are 1,748 repeat edges longer than 5 kb, forming 749 connected components in the Flye assembly graph of the Human dataset before performing bridged repeat resolution. After bridged repeat resolution with ultra-long reads, there are only 765 repeat edges, forming 107 connected components in the assembly graph. Of these, 73 (34) represent mosaic (simple) segmental duplications (most simple ones represent isolated edges and loop-edges). See Appendix 2.7.4 for more details.

2.5 Discussion

We describe the Flye algorithm for constructing an assembly graph from SMS reads and demonstrate that repeat characterization improves genome assembly. We show how to use the assembly graph to resolve unbridged repeats using variations between repeat copies and compared Flye with the Canu, Falcon, HINGE, Miniasm, and MaSuRCA assemblers.

In the case of the Bacteria datasets, Flye and HINGE showed good agreement in the structure of constructed assembly graphs. Flye showed substantial improvement compared with HINGE on more complex eukaryotic datasets and generated the most accurate assemblies of the Yeast and Worm datasets; Flye and Canu also produced the best assembly contiguity in the case of the Worm dataset. For the more complex Human and Human+ datasets, Flye generated more contiguous and accurate assemblies than Canu and MaSuRCA, while being notably faster. Although assemblies of ONT reads feature rather high base-calling error rates (1.2% for the Flye Human assembly), polishing the Flye assembly graph using Illumina reads has the potential to reduce the error rates by an order of magnitude.

The fact that Flye substantially improved on the Canu and MaSuRCA assemblies of the human genome suggests that there are still unexplored avenues for increasing the contiguity of SMS assemblies. We believe that better algorithms for resolving unbridged repeats in assembly graphs have the potential to greatly improve SMS assemblies, potentially increasing their NGA50 values by an order of magnitude. Flye constructed a repeat graph of the human genome with only 765 repeat edges representing various long segmental duplications. Our algorithm for resolving unbridged repeats resolved only a small fraction of these segmental duplications since it is currently limited to simple segmental duplications (the vast majority of human segmental duplications are mosaic). Moreover, it currently has difficulties resolving highly similar segmental duplications, for example, segmental dulications with less than 1% divergence. Although we reported the resolution of highly similar segmental duplications on simulated datasets, most unbridged repeats resolved by Flye and Canu are simple repeats with divergence exceeding 3%. Extending Flye to mosaic segmental duplications has the potential to resolve most of the remaining unbridged repeats, since the vast majority of segmental duplications in the human genome diverge by more than 1% [Pu et al., 2018]. Since there are only 53 long segmental duplications (with length exceeding 15 kb) in the human genome that diverge by less than 1%, an SMS assembler that accurately resolves highly similar unbridged repeats will result in highly contiguous human genome assemblies, thus reducing the need for additional genome-finishing experiments (such as using Hi-C and/or optical maps).

Assembly graphs represent a special case of breakpoint graphs [Lin et al., 2014], and they are therefore well suited for analyzing structural variations [Chaisson et al., 2015, Nattestad et al., 2018] and segmental duplications [Jiang et al., 2007, Pu et al., 2018]. Flye assembly graphs provide a useful framework for reconstructing segmental duplications and planning additional genome-finishing experiments.

2.6 Acknowledgements

We are indebted to S. Nurk for his multiple rounds of critique and suggestions that have improved the paper. We are also grateful to A. Mikheenko, B. Behsaz, L. Pu, and G. Tesler for their comments. This work is supported by NSF/MCB-BSF grant no. 1715911.

This chapter, in full, is a reprint of the material as it appears in M. Kolmogorov, J. Yuan, Y. Lin and P. Pevzner, “Assembly of long error-prone reads using repeat graphs”, Nature Biotechnology (2019). The dissertation author was the primary developer of Flye and first author of this paper.

2.7 Appendices

2.7.1 Benchmarking Flye on a simple simulated genome

We simulated the “genome” shown in Figure 2.1 with two 99% identical copies of repeat R_1 of length 10 kb and two 99% identical copies of repeat R_2 of length 30 kb. The unique segments A, B, C, and D were simulated as random strings of length 250 kb each so that the total genome length is 1 Mb. Afterwards, we simulated reads of length N randomly sampled from this genome at coverage 100× using the PBSIM tool [Ono et al., 2012] and assembled them with Flye. We simulated two sets of reads, one with N = 12 kb (slightly larger than the length of the repeat R_1 but shorter than the length of the repeat R_2) and another with N = 10 kb.

In the case of N = 12 kb, Flye constructed the repeat graph (Figure 2.1f), identified the bridged repeat R_1, and resolved it as shown in Figure 2.1h. Afterwards, it resolved the unbridged repeat R_2 and reconstructed its two 99% identical copies (Fig. 2.1i), assembling the entire genome into a single circular contig.

In the case N = 10 kb, Flye constructed the repeat graph (Figure 2.1f), identified both R_1 and R_2 as unbridged repeats and resolved them as shown in Fig. 2.1i. As the result, it assembled the entire genome into a single circular contig.

2.7.2 Additional information on benchmarking

Running QUAST. QUAST 5.0 was run using the “--large” option for all eukaryotic genomes, which is recommended for the analysis of large genomes with complex repeat structures. The minimum alignment identity was set to a low 90% to account for the higher error rate in some regions of SMS assemblies. The minimum contig length was set to 50 kb for the Human/Human+ assemblies and 5 kb for all of the other assemblies.

Software versions used. All assemblies were run with the default parameters.

- Flye 2.3.5 (commit 20afeda)
- Canu 1.7.1 (commit dfa60b8)
- Falcon - 0.3.0 (FALCON-Integrate commit 7498ef9)
- HINGE - 0.5.0 (commit 79fdf66)
- Miniasm - 0.2-r168-dirty (commit 40ec280) / Minimap2 2.8-r711 (commit 8fc5f8d)
- QUAST 5.0.0 (commit de6973bb)

The Human (but not the Human+) assembly was generated with the earlier Flye version 2.3.2 (released on Feb 20 2018) to provide a fair comparison with the Canu and MaSuRCA assemblies (which were not updated since the release of Flye 2.3.2). We note that the Human assembly using the latest Flye version 2.3.5 has NGA50 = 7.3 Mb and improves over the Flye 2.3.2 assembly (NGA50 = 6.3Mb). Human+ was assembled using the latest Flye and Canu versions (as of September 2018).

Information about running time and memory usage. Table 2.3 provides information on the running time and memory usage of various SMS assemblers for the Yeast and Worm datasets

Flye took 5,000 CPU hours to generate assemblies of the Human+ dataset using an Intel(R) Xeon(R) 8164 CPU @ 2.00GHz. RAM usage was 500GB at peak. The Canu authors reported 30,000 CPU hours of run-time using a cluster with 48-core Intel(R) Xeon(R) CPU @ 2.5GHz with 128 Gb of RAM each (24 nodes) and two 80-core 1 Tb machines. The memory usage of a single job did not exceed 120GB. The MaSuRCA authors reported needing approximately 50,000 CPU hours.

2.7.3 Assemblies of the Metagenome dataset

Table 2.4 presents information about the Flye and Canu assemblies of the Metagenome dataset.

Table 2.3: Running time and memory usage of various SMS assemblers. We used a desktop machine with an Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz (up to 8 threads available) for the Yeast dataset assemblies and a single computational node with an Intel(R) Xeon(R) CPU X5680 @ 3.33GHz for the WORM dataset assemblies (up to 24 threads available). Since we performed the ABruijn polishing step on the Miniasm output, the running time for Flye and Miniasm are given for runs with and without contig polishing; e.g., 25m (9m) for Flye in the case of Yeast-PacBio dataset indicates 9m without polishing and 25 m with polishing.

Dataset	Assembler	Wall clock time	Peak RAM
Yeast-PacBio	Flye (w/o polishing)	20m (9m)	7G
12 Mb genome, 31x	Canu	80m	5G
8 threads max	Falcon	62m	10G
	HINGE	9m	5G
	Miniasm (w/o polishing)	16m (1m)	5G
Yeast-ONT	Flye (w/o polishing)	19m (12m)	7G
12 Mb genome, 31x	Canu	184m	6G
8 threads max	Falcon	103m	11G
	HINGE	11m	8G
	Miniasm (w/o polishing)	31m (3m)	5G
Worm	Flye (w/o polishing)	128m (77m)	30G
100 Mb genome, 40x	Canu	780m	41G
24 threads max	Falcon	945m	18G
	HINGE	803m	52G
	Miniasm (w/o polishing)	290m (10m)	23G

Synthetic metagenomic datasets often contain genomes with inaccurate references that present problems for follow-up benchmarking efforts [Nurk et al., 2017]. To estimate the expected number of misassemblies caused by the differences between the assembled and reference bacterial strains, we performed the assembly of each of 17 bacteria separately (separate assemblies) by binning the initial reads using alignments to the references and running Flye and Canu on the resulting set of reads (see Table S3). Six out of 17 separate assemblies *(R. sphaeroides, A. baumannii, B. cereus and B. vulgatus)* were fragmented into 2-4 contigs per chromosome (by both Flye and Canu), while the remaining 11 resulted in a single contig per chromosome. Nevertheless, metaQUAST reported 92 misassemblies in total for the Flye separate assemblies (and 103 misassemblies for Canu). The misassemblies reported for Flye and Canu were highly correlated: 80% of Flye (70% of Canu) misassembly breakpoints had a matching breakpoint in

Table 2.4: Information about of the Flye and Canu assemblies of the Metagenome dataset. Statistics were computed using MetaQUAST v5.0 with default parameters for the bacterial genomes. Entries in bold highlight five assemblies where Flye significantly improved on Canu and four assemblies where Canu significantly improved on Flye. Flye and Canu produced 84 and 99 misassemblies in total, respectively.

			Flye			Canu		
bacteria	length (kb)	coverage	% assembled	NGA50 (kb)	NGA50 (kb)	% assembled	NGA50 (kb)	NGA50 (kb)
A. baumannii	3,976	40	99.8	906	18	99.8	906	19
A. odontolyticus	2,393	41	99.5	622	6	99.8	1,285	5
B. cereus	5,224	25	99.8	2,716	4	99.5	581	4
B. vulgatus	5,163	46	99.6	832	18	98.9	539	20
D. radiodurans	3,060	52	99.5	253	25	99.6	224	27
E. faecalis	2,793	43	99.9	2,738	0	99.9	2,747	0
E. coli	4,640	46	99.9	4,637	0	99.9	4,643	0
H. pylori	1,667	317	100	165	2	100	1,314	3
L. gasseri	1,894	83	97.9	898	1	97.7	969	1
L. monocytogenes	2,944	98	96.4	2,008	0	100	1,507	1
P. acnes	2,560	65	100	2,560	0	100	2,566	0
P. aeruginosa	6,264	55	99.9	4,001	3	99.9	3,998	9
R. sphaeroides	4,131	24	99.4	2,006	1	90.1	54	0
S. aureus	2,872	66	98.2	1,003	0	100	1,543	2
S. epidermidis	2,499	59	99.7	1,276	1	100	2,465	2
S. agalactiae	2,160	42	98.8	1,836	0	99.9	2,159	0
S. mutans	2,032	82	99.9	1,554	0	99.9	1,085	3

the Canu (Flye) contigs (two breakpoints are matching if their reference coordinates are within 1 kb; note that a single misassembly might have two breakpoints). We thus concluded that the misasemblies reported by metaQUAST were mainly caused by the differences between the genomes in the Metagenome sample and the reference genomes rather than assembly artifacts.

2.7.4 Human segmental duplications identified by Flye

After all unique edges are removed from the assembly graph of the Human+ dataset, it breaks into connected components formed by repeat edges and reveals putative segmental duplications (which might also include short edges corresponding to unresolved common repeats). Figure 2.9 shows the distribution of lengths of repeat edges exceeding 5 kb and the distributions of lengths of ultra-long segmental duplications (longer than 50 kb).

We illustrate how Flye resolves unbridged repeats using all five unbridged repeats of multiplicity two in the assembly graph of the HUMAN+ dataset constructed by Flye (Table 2.6). Flye resolved all five repeats, which range in length from 37 kb to 152 kb, in coverage from 26x

Table 2.5: Analysis of the separate assemblies of 17 genomes from the Metagenome dataset. Initial reads were binned into 17 groups using alignments to their respective references. Flye and Canu produced 92 and 104 misassemblies in total, respectively. Statistics were computed using MetaQUAST v5.0. All genomes but the six marked with * (R. sphaeroides, A. baumannii, B. cereus, B. vulgatus, D. radiodurans and P. aeruginosa) were assembled into a single contig per chromosome. Six of the remaining 11 Flye assemblies (marked with +) had no misassemblies compared to the reference. Canu generated four assemblies without reported errors.

			Flye			Canu		
bacteria	length (kb)	coverage	% assembled	NGA50 (kb)	NGA50 (kb)	% assembled	NGA50 (kb)	NGA50 (kb)
A. baumannii*	3,976	40	99.8	906	21	99.8	906	18
A. odontolyticus	2,393	41	99.8	1,286	4	99.8	1,285	5
B. cereus*	5,224	25	99.6	4,948	3	99.8	4,625	3
B. vulgatus*	5,163	46	99.3	832	21	99.2	1,112	28
D. radiodurans*	3,060	52	99.6	253	31	99.5	222	31
E. faecalis+	2,793	43	99.9	2,738	0	99.9	2,745	0
E. coli+	4,640	46	99.9	4,638	0	99.9	4,643	0
H. pylori	1,667	317	100	1,123	2	100	1,617	2
L. gasseri	1,894	83	97.9	1,729	1	97.8	961	4
L. monocytogenes+	2,944	98	100	2,944	0	100	2,151	1
P. acnes+	2,560	65	100	2,560	0	100	2,566	0
P. aeruginosa*	6,264	55	99.8	1,982	2	99.9	3,998	6
R. sphaeroides*	4,131	24	99.9	2,669	2	99.9	2,578	0
S. aureus	2,872	66	99.8	2,665	1	100	1,571	2
S. epidermidis	2,499	59	100	2,498	1	100	1,319	2
S. agalactiae+	2,160	42	99.7	2,155	0	99.9	1,602	1
S. mutans+	2,032	82	99.9	2,032	0	99.9	1,546	1

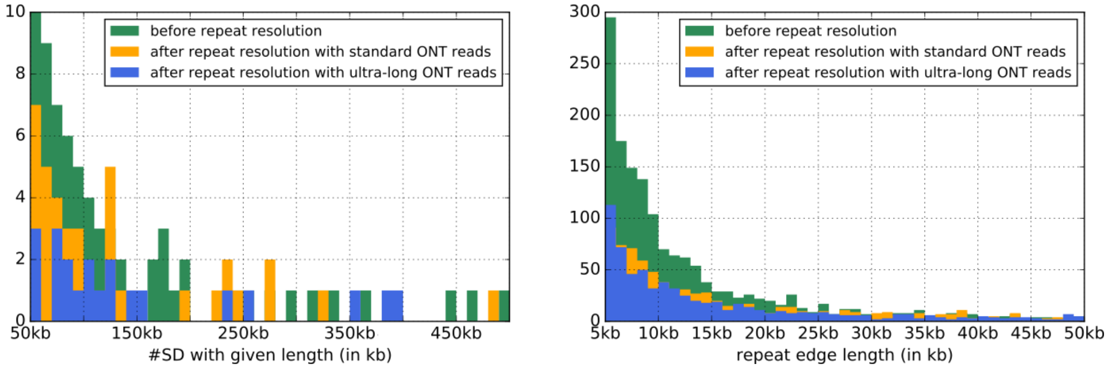


Figure 2.9: The distribution of lengths of segmental duplications (SDs) longer than 50 kb for the assembly graph constructed for the Human+ dataset (left) and the lengths of all other repeat edges (right). (Left) 39 out of 81 SDs (48%) longer than 50 kb were resolved using standard ONT reads. Ultra-long reads resolved an additional 20 SDs (28%) in this range of SD lengths. (Right) Only edges varying in length from 5 kb to 50 kb contributed to the histogram. In addition to these edges, there are 213 (90) repeat edges with length exceeding 50 kb before repeat resolution (after repeat resolution with ultra-long reads). Note that while a similar figure in the main text describes the lengths of SDs (connected components formed by repeat edges), this figure describes the length of individual repeat edges.

to 31x, and in divergence from 1.77% to 7.76%.

All resolved repeats correspond to known segmental duplications in the human genome. The sequences of the constructed repeat copies preferentially mapped to specific copies of segmental duplications, showing that our method is successful even in the presence of Single Nucleotide Polymorphisms (SNVs). For example, repeat 902 aligns to two 50 kb regions of chromosome X (separated by 65 kb), which are annotated as segmental duplications.

The diploid nature of the human genome may add some complications to the repeat resolution procedure, especially if many SNVs are present in the repeat. However, if the divergence of the repeat significantly exceeds the fraction of SNVs, the described algorithm will still be able to resolve the unbridged repeat. Since the divergence of repeats analyzed in Table S4 (above 4%) significantly exceeds the fraction of SNVs in the human genome (0.1%), SNVs do not significantly affect our approach. However, in the case of unbridged repeats with low divergence (e.g., below 1%), our algorithm has to be modified to take SNVs into account. When the algorithm is extended to repeats of higher multiplicity, it will automatically resolve haplotypes for diploid and polyploid genomes since they will simply be treated as additional repeat copies.

2.7.5 Inconsistent pairwise alignments

[Pevzner et al., 2004] introduced the concept of alignment-based de Bruijn graphs (A-Bruijn graphs) and applied them for repeat characterization and genome assembly. They further described the transformation of an A-Bruijn graph into a repeat graph that is particularly simple in the case of consistent alignments as described below.

Each multiple alignment of m sequences induces $\binom{m}{2}$ pairwise alignments. A set of pairwise alignments (described by the repeat plot) is *consistent* if its alignments can be combined into a single multiple alignment that induces each pairwise alignment in the set. The concept of multiple alignment is usually defined for the case of aligning multiple sequences rather than for aligning a sequence against itself. Below we describe the concept of a multiple self-alignment of

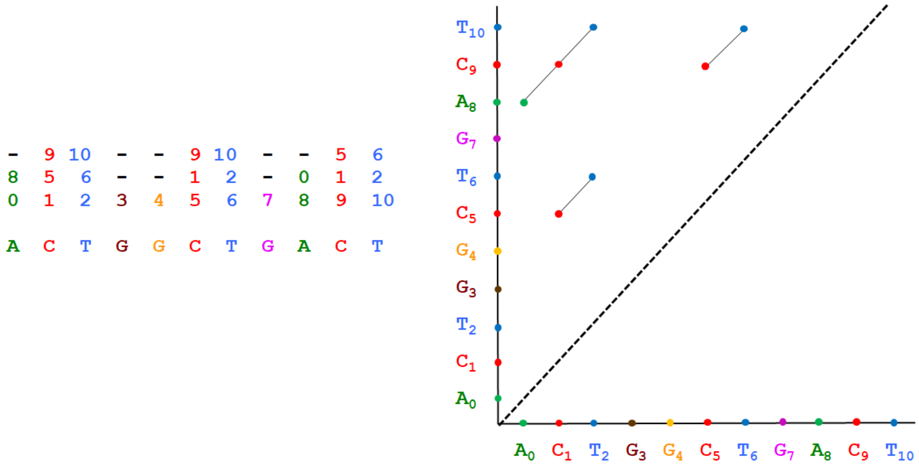
Table 2.6: Resolving unbridged repeats of multiplicity two in the assembly graph of the Human+ dataset. The assembly graph of the Human+ dataset has five unbridged repeats of multiplicity two. The identifier of each unbridged repeat is given by its edge identifier in the assembly graph. All repeats have been resolved. The "coverage" is calculated as the total length of reads covering the repeat divided by the repeat length, divided by the multiplicity of the repeat. The "divergence" is calculated based on the alignment of constructed repeat consensus sequences, dividing the total number of substitutions and indels by the total number of matches, substitutions, and indels (if the forward and reverse consensus sequences do not overlap, then the mean divergence of the forward and reverse sequences is calculated, weighted by the length of the sequences). "Maximal gap" refers to the maximum of all distances between adjacent confirmed divergent positions. "Remaining gap" refers to the length of the repeat remaining without separate consensus sequences for each copy after Flye has "moved into the repeat" from both the forward and reverse directions. In the case that the forward and reverse consensus sequences overlap, the remaining gap is set to 0.

repeat ID	length (kb)	coverage	divergence	tentative div. pos.	confirmed div. pos.	maximal gap (kb)	remaining gap (kb)	cis reads	trans reads
625	152	27x	5.36%	29713	3256	79.2	32.2	2	12
902	51	28x	1.77%	5694	1541	0.7	0	43	13
1018	86	26x	6.77%	17509	11360	0.7	0	17	154
1075	37	28x	3.05%	4379	1406	0.3	0	38	136
1233	49	31x	7.76%	11786	8590	0.3	0	45	2

a genome and define the notion of consistent pairwise self-alignments. This notion is important since A-Bruijn graphs result in a simple repeat graph in the case of consistent self-alignments but in a more complex graph in the case of inconsistent self-alignments (see [Pevzner et al., 2004] for a discussion of complications arising from inconsistent self-alignments).

A multiple self-alignment of a single sequence is a partition of its positions into non-overlapping subsets, with each subset corresponding to a column of the multiple self-alignment. For example, a multiple self-alignment of the sequence *ACTGGCTGACT* can be represented as a partition of its 11 positions into six "painted" subsets: $A_0C_1T_2G_3G_4C_5T_6G_7A_8C_9T_{10}$. Figure 2.10 visualizes such a partitioning as a multiple self-alignment where each column represents positions from the same subset:

Every pair of numbers $i < j$ in the same column of the multiple self-alignment defines a point (i, j) in the two-dimensional plot. For example, the leftmost column in Figure A5 corresponds to a point $(0, 8)$ and the rightmost column corresponds to points $(2, 6)$, $(2, 10)$, and $(6, 10)$. The collection of all such points defines the dot plot of the multiple self-alignment. We



-	9	10	-	-	9	10	-	-	5	6
8	5	6	-	-	1	2	-	0	1	2
0	1	2	3	4	5	6	7	8	9	10
A	C	T	G	G	C	T	G	A	C	T

T_{10}
C_9
A_8
G_7
T_6
C_5
G_4
G_3
T_2
C_1
A_0
A_0 C_1 T_2 G_3 G_4 C_5 T_6 G_7 A_8 C_9 T_{10}

Figure 2.10: Multiple self-alignment defined by the partitioning of $A_0C_1T_2G_3G_4C_5T_6G_7A_8C_9T_{10}$ into six subsets (left) and the corresponding dot-plot (right). In difference from the traditional representation of a multiple alignment (where each entry represents a nucleotide or a dash in the multiple alignment matrix), each entry in the multiple self-alignment matrix represents a position in the sequence or a dash.

refer to a rectangle in the dot plot with lower left corner (x,y) and upper right corner (x',y') as (x,y,x',y'). A pairwise alignment between segments (x,x') and (y,y') of a genome defines a set of two-dimensional points in the rectangle (x,y,x',y') corresponding to matches in this alignment. A multiple self-alignment and a pairwise alignment between segments (x,x') and (y,y') are consistent if the dot plot of the multiple self-alignment coincides with the dot plot of the pairwise alignment within the rectangle (x,y,x',y'). A set of pairwise alignments is consistent if there exists a multiple self-alignment that is consistent with all pairwise alignments in this set, and inconsistent otherwise.

2.7.6 Inconsistent alignments result in excessively complex repeat graphs

Figure 2.11 presents an example of inconsistent pairwise alignments and illustrates that they result in a repeat graph that differs from the repeat graph shown in Figure 2.2 of the main text. In difference from the graph in Figure 2.2, the graph in Figure 2.11 is not alignment-compatible; e.g., the repeat $A+B$ corresponds to a single path in Figure 2.2 but two paths in Figure 2.11.

Although it may appear to be a minor annoyance in the case of the toy example in Figure 2.11, inconsistent alignments may result in excessively complex repeat graphs in the case of real genomes, making it difficult to analyze repeats in the genome. While it is easy to make the pairwise alignments consistent in the simple case shown in Figure 2.11 (by adding a missing diagonal), transforming inconsistent pairwise alignments into consistent ones is a challenging task in the case of real genomes.

The approximate repeat graph in Figure 2.11 has seven vertices (since there exist seven projections of alignment endpoints to the main diagonal), in contrast to the approximate repeat graph in Figure 2.2 of the main text that has eight vertices. This deficiency of the approximate repeat graph in Figure 2.11 motivates the algorithm for extending the set Breakpoints that is described in the main text. Note that the middle point of the long diagonal in Figure 2.11 represents an invalid point since only one of its projections (shown as a purple point) belongs to the set of seven endpoint projections on the main diagonal. The algorithm described in the main text adds the missing projection to the set Breakpoints and results in the same approximate repeat graph as shown in Figure 2.2 in the main text.

2.7.7 The challenge of assembling contigs into a repeat graph

The ABruijn algorithm constructs a set of contigs but does not attempt to assemble them into even longer contigs (e.g. by utilizing ultra-long reads) and stops short of constructing the repeat graph of the genome [Lin et al., 2016]. We note that contig assembly (let alone constructing the repeat graph based on contigs) is a non-trivial problem. Although it may appear that contig assembly can be achieved by simply utilizing existing long read assemblers, [Bankevich and Pevzner, 2016] reported that Celera [Myers et al., 2000], Minimus [Treangen et al., 2013], and Lola [Sharon et al., 2015] assemblers produced suboptimal assemblies of contigs generated using TruSeq Synthetic Long Reads (TSLR) technology. Their attempts to modify the short read assembler SPAdes [Bankevich et al., 2012] for TSLR assembly improved on the results of Celera,

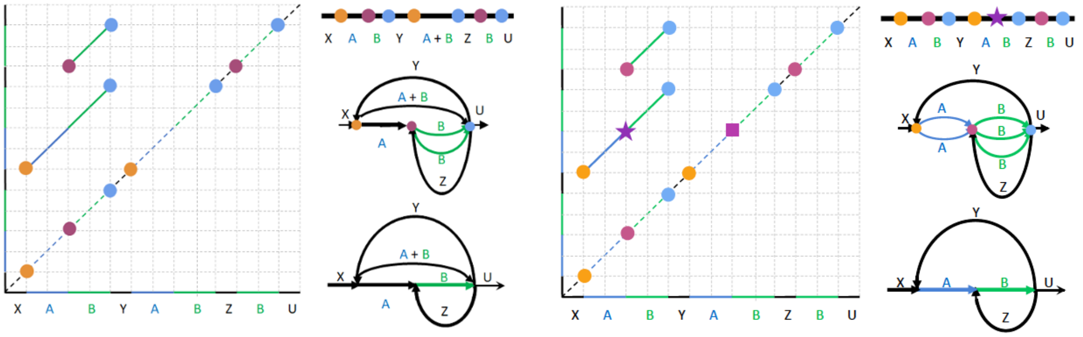


Figure 2.11: Inconsistent pairwise alignments result in an incorrect repeat graph (as compared to the graph shown in Figure 2.2 in the main text), thus necessitating an extension of the set of alignment endpoints. (Left) Alignment-paths for two pairwise self-alignments within a genome *XABYABZBU*. Only two out of three pairwise alignments between instances of a mosaic repeat (*AB*, *AB*, and *B*) are shown since the third alignment did not pass the percent identity threshold, resulting in an inconsistent set of pairwise alignments. Alignment endpoints are clustered together if their projections on the main diagonal coincide or are close to each other (clusters of closely located endpoints for $d = 0$ are painted with the same color). This clustering reveals three clusters with seven breakpoints on the main diagonal. (Top right) Projections of the clustered endpoints on the main diagonal define seven vertices of the approximate repeat graph. (Middle right). Gluing breakpoints that belong to the same clusters. (Bottom right) Gluing parallel edges in the resulting graph (parallel edges are glued if there exists an alignment between their sequences), which results in an approximate repeat graph that is not alignment-compatible. (Right) Extending the set Breakpoints by adding an additional point to the longest diagonal (shown as a star). Since the middle point of the longer alignment-path is invalid (its vertical projection on the main diagonal belongs to the set Breakpoints but its horizontal projection does not), we have added the missing projection to the set Breakpoints (shown as a purple square). Adding this breakpoint is equivalent to breaking the longer alignment-path into two subpaths (the breakage position is shown as a purple star). As a result of the breakpoint extension procedure, the approximate repeat graph constructed based on the extended set Breakpoints coincides with the approximate repeat graph shown in Figure 2.2 of the main text.

Minimus, and Lola but stopped short of constructing the contig-based repeat graph.

Similar challenges remain unresolved in the case of short reads. Although popular short read assemblers construct the assembly graph of single reads (before resolving repeats using paired reads), they output a set of contigs (after the repeat resolution step) rather than an assembly graph that utilizes information about paired reads. For example, SPAdes [Bankevich et al., 2012] constructs the assembly graph of single reads, uses it together with paired reads for repeat resolution, and outputs the resulting contigs [Prjibelski et al., 2014]. A better option would be to construct the assembly graph of these contigs (which is less tangled than the assembly graph of individual reads) and to apply the repeat resolution step again to this graph. Another advantage of this (less tangled) contig-based assembly graph lies in applications relating to hybrid assembly, e.g., co-assembly of short and long reads [Antipov et al., 2015, Wick et al., 2017]. However, although some studies attempted to construct the assembly graph from contigs or directly from paired reads [Vyahhi et al., 2012], the popular short read assemblers have failed to incorporate this approach into their pipelines so far.

2.7.8 FlyeWalk algorithm

The FLYEWALK algorithm (shown below) computes alignments (within the Overlap, MAPREADS, and EXTENDREAD procedures) using the longest jump-subpath approach [Lin et al., 2016]. In difference from other SMS assemblers, FLYEWALK does not generate all-versus-all pairwise alignments between reads (a major time bottleneck) since reads that align to a newly assembled disjointig are removed from the set UNPROCESSEDREADS.

Given a chain of reads *ChainOfReads* formed by reads $Read_1$ ... $Read_n$, we define $prefix(Read_i)$ as the overlapping region between consecutive reads $Read_{i-1}$ and $Read_i$ in the chain and $suffix(Read_i)$ as the suffix of the i-th read after the removal of $prefix(Read_i)$ (note that $suffix(Read_1)$ coincides with $Read_1$). We define *concatenate(ChainOfReads)* as the concatenate $suffix(Read_1)$ ... $suffix(Read_n)$ of read suffixes in this chain. The CONSENSUS procedure constructs an initial draft sequence (disjointig) of a chain *ChainOfReads* by constructing *concatenate(ChainOfReads)*. Afterwards, all reads from the dataset are aligned to the draft disjointig sequence using minimap2 [Li, 2018] and the consensus of the aligned reads is formed by taking the majority vote. This procedure reduces the error rate in the draft disjointig sequence from 13% to 1-5%, depending on the contig coverage. The follow-up polishing step reduces the error rate to 0.1% when the coverage exceeds 30×.

EXTENDREAD is run in a single thread but multiple EXTENDREAD procedures are run

```
function FlyeWalk(AllReads, MinOverlap)
    Disjointigs← empty set of contigs
    UnprocessedReads← AllReads
    for each Read in UnprocessedReads do
        ChainOfReads← ExtendRead(UnprocessedReads, Read, MinOverlap)
        DisjointigReads← MapReads(AllReads, ChainOfReads, MinOverlap)
        DisjointigSequence← Consensus(AllReads, DisjointigReads, MinOverlap)
        add DisjointigSequence to Disjointigs
        remove DisjointigReads from UnprocessedReads
    end for
    return Disjointigs
end function
function ExtendRead(UnprocessedReads, Read, MinOverlap)
    ChainOfReads← sequence of reads consisting of a single read Read
    while forever do
        NextRead← FindNextRead(UnprocessedReads, Read, MinOverlap)
        if $NextRead = \emptyset$ then
            return ChainOfReads
        else
            add NextRead to ChainOfReads
            Read← NextRead
            remove Overlap(Read) from UnprocessedReads
        end if
    end while
end function
```

algorithm 1: Pseudocode of the FlyeWalk algorithm. FlyeWalk iteratively extends each unprocessed read and organizes the selected reads into a chain. Each such chain contributes to a disjointig and FlyeWalk outputs the set of all disjointigs resulting from such extensions. ExtendRead generates a random walk in the assembly graph, which starts at a given read and constructs a chain of overlapping reads that contribute to a constructed disjointig. It terminates when there are no unprocessed reads overlapping the current read by at least *MinOverlap* nucleotides. FindNextRead finds an unprocessed read that overlaps with the given read by at least *MinOverlap* nucleotides and returns an empty string if there are no such reads. MapReads finds all reads that align to a given chain of reads over their entire length with the possible exception of a short suffix and/or prefix of length at most *MinOverlap*. Consensus constructs the consensus of all reads that contribute to a given disjointig. Overlap finds all reads that overlap a given read by at least MinOverlap nucleotides.

in parallel for each read that is not in *UnprocessedReads*. When one of the ExtendRead procedures finishes, the algorithm checks if the returned disjointig has a significant overlap (by more than 10% of its length) with one of the previously constructed disjointigs from *Disjointigs*.

If such an overlap is found, the new disjointig is discarded and the reads from this disjointig are returned to the set *UnprocessedReads*. This parallelization significantly speeds up FLYEWALK for assemblies that contain many contigs.

2.7.9 Flye constructs an accurate assembly graph from error-prone disjointigs

There exist two tours in the assembly graph for the *E. coli* strain NCTC9964 shown in Figure 2.4 (middle) of the main text: the correct genomic tour formed by edges IN_1, REP, OUT_1, and REP' (the corresponding complementary tour is formed by the complementary edges REP, OUT_2, REP', and IN_2) and the incorrect tour formed by edges IN_1, REP, OUT_2, and REP' (the corresponding complementary tour is formed by edges IN_2, REP, OUT_1, and REP').

Although paths $IN_1 \to REP \to OUT_2 \to REP'$ and $IN_2 \to REP \to OUT1 \to REP'$ form incorrect disjointigs, they are however assembled in the correct assembly graph by Flye. Below we explain why an arbitrary set of paths (disjointigs) constructed by FLYEWALK results in a correct assembly graph. Although our arguments apply to the punctilious repeat graph, the construction of the approximate repeat graph follows a similar logic, and the Results section demonstrates that these graphs constructed by Flye also result in accurate assemblies.

Let *Genome* be an (unknown) genomic sequence of an (unknown) length with an (unknown) alignment matrix *Alignments*. Let $Strings = \{s(1),...,s(t)\}$ be a covering set of strings for *Genome*, and $A(i,j)$ be the alignment snapshot, i.e., the sub-matrix of *Alignments* for substrings $s(i)$ and $s(j)$. Given a concatenate $Strings* = s(1)*s(2)*...*s(t)$ of all t substrings (with delimiters), their $t*(t-1)/2$ pairwise alignment snapshots $A(i,j)$ can be combined together to form the alignment matrix *Alignment** of the entire concatenate. We emphasize that the coordinates of the strings $s(1),...,s(t)$ and their ordering in the sequence *Genome* are unknown.

[Pevzner et al., 2004] demonstrated that *RepeatGraph(Genome, Alignments)* coincides with the repeat graph *RepeatGraph(Strings*, Alignments*)* of a concatenate of all substrings (in any order) for any covering set of substrings. As we explain below, this result implies that the Flye assembly of inaccurate disjointigs generated by FlyeWalk results in an accurate assembly graph. For simplicity, we assume that chimeric reads have been removed and that no read is contained within another read.

Consider the set of disjointigs $\{disjointig_1, disjointig_2, ..., disjointig_t\}$ constructed by FlyeWalk and map all reads to all these disjointigs. Since FlyeWalk utilizes all reads, each read will be mapped to one or more disjointigs. We now concatenate all reads starting from reads in the first disjointig, followed by reads in the second disjointig, etc., resulting in the sequence of reads:

$$\{s(1,1),s(1,2),...,s(1,n_1)\},\{s(1,1),s(2,1),...,s(2,n_2)\},...,\{s(t,1),s(t,1),...,s(t,n_t)\} \tag{2.1}$$

where $s(i,j)$ stands for the j-th read in the i-th disjointig (reads are ordered in the increasing order of their starting positions in each disjointig).

Since all reads are included in this concatenate, the repeat graph constructed from this concatenate coincides with the repeat graph of the genome [Pevzner et al., 2004]. Since the repeat graph does not depend on the order in which the reads are glued, we will perform gluing in two stages. At the first stage, we will perform some (but not all) gluing operations on reads from the first disjointig, followed by some gluing operations on reads from the second disjointig, etc. Specifically, with respect to the i-th disjointig, we will only glue overlapping reads within this disjointig (i.e., reads $s(i,n)$ and $s(i,m)$ if $n < m$ and read $s(i,m)$ starts before read $s(i,n)$ ends) and will only apply gluing operations to their overlap. Note that the first gluing stage does not necessarily includes all gluing operations applicable to reads from the i-th disjointig, e.g., non-overlapping reads within this disjointig may share sufficiently long alignments that however do not contribute to the first-stage gluing.

The first-stage gluing of reads that were sampled from a single disjointig results in the consensus of this disjointig constructed by FLYEWALK. Thus, the application of such "intra-disjointig" gluing operations to all disjointigs results in the set of disjointigs $\{disj_1, disj_2, ..., disj_t\}$. Note that only some but not all gluing operations have been performed at this point; e.g., inter-disjointig gluing has not been applied yet. Therefore, the second-stage gluing of all disjointigs constructed by FLYEWALK (some of them may be misassembled) results in the same assembly graph as gluing all reads, and thus results in the repeat graph of the genome.

2.7.10 Constructing the repeat graph from substrings of a genome

The repeat graph construction algorithm assumes that the genome Genome and the two-dimensional matrix *Alignments* (defining the pairwise alignments between similar substrings of the genome) are given. Any two substrings of the genome define a rectangle in the matrix *Alignments* that we refer to as an alignment snapshot imposed by these substrings. Given a set of substrings from *Genome*, [Pevzner et al., 2004] asked whether the repeat graph can be constructed from their pairwise snapshots without knowing Genome and the entire matrix *Alignments*. This question is relevant to genome assembly when the *Genome* and *Alignments* are unknown but the alignments between substrings of the genome (reads) can be computed as an approximation of alignment snapshots.

A set of substrings of a genome forms a covering set if, for every pair of consecutive positions in Genome, there exists a substring containing these positions. [Pevzner et al., 2004] demonstrated that if substrings of a genome (reads) form a covering set, then gluing an arbitrary concatenate of these substrings (separated by delimiters), according to their alignment snapshots, produces the same repeat graph as gluing the entire *Genome*.

This result holds for the ideal case when the alignment snapshots are inherited from the matrix *Alignments* representing all self-alignments of *Genome*. Since Genome and the matrix *Alignments* are unknown in the case of genome assembly, the alignment snapshot between two substrings (reads) is computed as their pairwise alignment rather than derived as the corresponding rectangle in the *Alignments* matrix. This pairwise alignment may differ from the alignment snapshot; for example, an alignment between two reads overlapping by a single nucleotide will be captured in their alignment snapshot (since it is a part of a larger matrix *Alignments*) but not in their pairwise alignment since it does not pass a statistical significance threshold. That is why [Pevzner et al., 2004] introduced a more stringent condition for the concept of the covering set of substrings: for each *m* consecutive positions in Genome (where m is a pre-defined threshold), there must exist a substring (read) spanning all these positions. This condition explains why it is important that Flye generates disjointigs satisfying the overhang property.

2.7.11 Aligning reads to the assembly graph

Flye aligns all reads to the constructed assembly graph using the concept of common jump-subpaths [Lin et al., 2016]. First, each read is matched against the edges of the assembly graph. For each repeat edge in the assembly graph, we store all copies of the corresponding repeat (from the original disjointigs), rather than a single consensus of all sequences contributing to this repeat edge. We then match a read to all these copies and select the best alignment to improve the recruitment of reads to the edges of the assembly graph. If a read is aligned to multiple edges in the assembly graph, we find a maximum scoring path in the graph formed by such edges using dynamic programming.

2.7.12 Identifying repeat edges in the assembly graph

After constructing the assembly graph, Flye aligns all reads to this graph and forms a read-path for each read. Given the alignments of all reads against the assembly graph, Flye computes the mean depth of coverage cov across the entire assembly graph and classifies an edge as low-coverage (if its coverage is below $2 * cov$) and high-coverage (if its coverage is at least

$2 * cov$). While most low-coverage edges are unique (traversed only once in the genomic tour), some of them are repetitive since the coverage varies along the genome.

To improve the classification of unique and repetitive edges in the assembly graph, Flye reclassifies some edges using information about the read-paths. An edge e' in the assembly graph is a successor of an edge e if it follows e in one of the read-paths. A low-coverage edge is classified as unique if it has a single successor. All other edges (i.e., high-coverage edges and low-coverage edges with multiple successors) are classified as repetitive.

To avoid classifying chimeric connections in the assembly graph as successor edges and to minimize the influence of misaligned reads, Flye imposes an additional restriction on the edge to classify it as a successor: a fraction of the reads supporting a successor (among all reads contributing to the successor of a given edge) should exceed N percent of the fraction of the reads supporting the most frequent successor (the default value is $N = 20\%$).

We used the Flye *C. elegans* assembly to estimate the accuracy of Flye's classification of unique and repetitive edges. For each edge in the *C. elegans* assembly graph, we found whether it is unique or repetitive in the reference genome by aligning the edge to the entire reference genome and checking whether there exists a single alignment (unique edge) or multiple alignments (repetitive edge). This analysis revealed that the *C. elegans* assembly graph has 339 unique and 219 repetitive edges. Flye has misclassified 5 out of 219 repetitive edges as unique (2%) and 22 out of 339 unique edges as repetitive (6%). Note that only errors of the first type (misclassifying repeat edges as unique) lead to potential misassemblies during the repeat resolution step. Errors of the second type (classifying unique edges as repeat edges) do not lead to misassemblies but may potentially negatively affect the contiguity of the assembly since misclassified unique edges do not contribute to repeat resolution. This is however not a critical shortcoming in practice since long reads often bridge these misclassified edges.

2.7.13 Additional details on untangling assembly graphs

The maximum weight matching defines the set of edges in the transition graph; Flye additionally checks each of the inferred edges as follows. For each edge (u, v) from the matching, it computes the total weight *TotalWeight* of all edges in the transition graph adjacent to u or v. If $transition(u, v) < TotalWeight/2$, the edge is classified as weak and is consequently ignored. Weak edges typically arise from long repeats that may be bridged by a few reads in an ambiguous way.

Flye iteratively untangles edges and finds maximum weight matchings until no extra repeats can be resolved. Note that a repeat of multiplicity t may require less than t untangling operations to be completely resolved. For example, a repeat edge *REP* of multiplicity 2 in the assembly graph (with incoming edges IN_1 and IN_2 and outgoing edges OUT_1 and OUT_2) may only have bridging reads traversing IN_1, *REP*, and OUT_1 but not IN_2, *REP*, and OUT_2. However, using bridging reads to untangle IN_1 and OUT_1 (essentially forming a single edge from edges IN_1, *REP*, and OUT_1), turns the sequence of edges IN_2, *REP*, and OUT_2 into a non-branching path and thus completely untangles the repeat.

Note that some short edges are reclassified as long during this process and that some repetitive edges are reclassified as unique during the next iteration of the algorithm (for example, if they were a part of a bigger mosaic repeat that was partially resolved).

2.7.14 Revealing variable positions within repeats

To reveal the variable positions within a repeat (a repeat edge in the assembly graph), we map all reads to the consensus sequence of the repeat and generate a multiple alignment of all reads that are contained within or overlap with the repeat. Afterwards, we determine the second most frequent nucleotide in each column of the multiple alignment and define the substitution rate in this column as the number of occurrences of the second most frequent nucleotide divided by the total number of reads covering this column (note the difference between the concepts of substitution rate here and in [Lin et al., 2016]. We define the deletion and insertion rates in each column as in [Lin et al., 2016]. If the substitution, deletion, or insertion rate for a column exceeds a predefined threshold, the corresponding position is called a tentative divergent position. The repeat divergence is estimated by dividing the total number of tentative divergent positions by the length of the repeat. See Appendix 2.7.4 for a discussion on how repeat divergence can be affected by diploidy.

Below we construct the distribution of substitution, deletion, and insertion rates in non-divergent positions, compare it with the distribution of substitution, deletion, and insertion rates in divergent positions, and select a threshold that separates these two distributions. To construct these distributions, we selected the 22 kb long repeat (repeat *REP* in Figure 2.4) in the assembly graph of the EC9964 dataset (other repeats result in very similar distributions). Since this repeat has many variations between two repeat copies, (943 substitutions (4.3%), 346 deletions (1.6%), and 226 insertions (1.0%)), we manually resolved it with high confidence. A position in this repeat is classified as variable if it corresponds to a substitution, deletion, or insertion, and non-variable otherwise.

We mapped reads from the EC9964 dataset to the consensus of the *REP* repeat and calculated the substitution, deletion, and insertion rates. Figure 2.12 illustrates that variable positions feature higher substitution, deletion, and insertion rates than non-variable positions within a repeat. We thus identify tentative divergent positions based on mutation rates by selecting a mutation rate threshold that provides a good separation between the two distributions (0.1, 0.2, and 0.3 for substitutions, deletions, and insertions, respectively). This results in the identification of 924 out of 943 substitutions, 270 out of 346 deletions and 54 out of 226 insertions for the *REP* repeat. At the same time, we misclassified 81 non-variable positions as divergent (61 substitutions, 5 deletions, and 15 insertions), resulting in a false positive rate of 0.4%. In all, we identified 1329 tentative divergent positions, which leads to a divergence estimate of 6.0%, a slight underestimation of the true divergence rate of 6.9%.

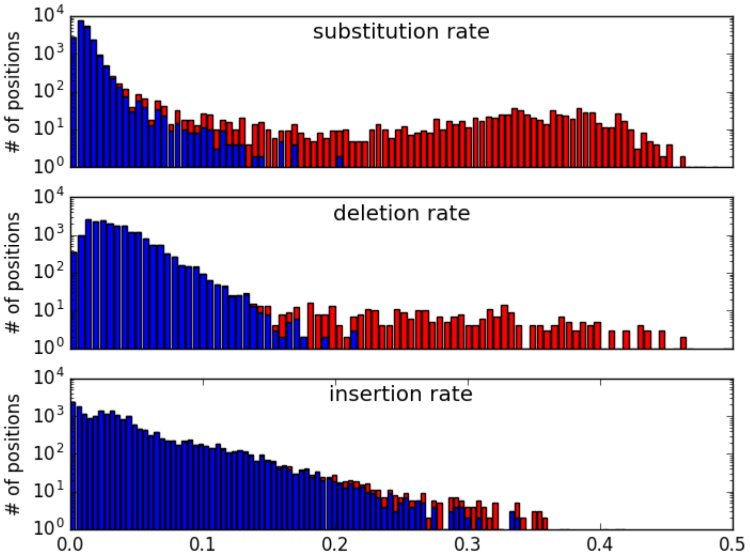


Figure 2.12: Separating variable and non-variable positions within repeats using substitution, deletion, and insertion rates computed for the REP repeat in the EC9964 dataset. Substitution (top), deletion (middle), and insertion (bottom) rates at each position in the multiple alignment of reads. Blue (red) bars represent mutation rates for non-variable (variable) positions. The number of positions with a given mutation rate (y-axis) is shown in a logarithmic scale. The cutoffs 0.1, 0.2, and 0.3 result in a good separation between variable and non-variable positions for substitutions, deletions, and insertions, respectively.

Table 2.7 provides details about Flye performance on the unbridged repeats from the Bacteria dataset.

2.7.15 Additional details on the unbridged repeat resolution approach

Initially, the unbridged repeat resolution algorithm recruits all reads traversing edges IN_1 and *REP* (IN_2 and *REP*) to the first (second) repeat copy and computes the consensus of each repeat copy using the recruited reads. Since the recruited reads do not span the entire edge *REP*, we only construct two consensus sequences corresponding to prefixes of *REP* where there is substantial read coverage by the recruited reads. We require at least *minCoverage* for

Table 2.7: Resolving unbridged repeats of multiplicity two in genomes from the Bacteria dataset. The results of repeat resolution after running Flye for 11 out of 21 genomes from the Bacteria datasets that contain repeats of multiplicity two. The label of each dataset denotes the bacterial species, its strain, and the ID number of the repeat edge found in the assembly graph (e.g. EC5052-7, EC5052-8, and EC5052-9 refer to 3 repeats with IDs "7", "8", and "9" present in the assembly graph for the *E. coli* NCTC5052 dataset). Bolded labels refer to repeats resolved by Flye. The * refers to a repeat of multiplicity 3. The "coverage" is calculated as the total read length divided by the repeat length, divided by the multiplicity of the repeat (comparable to the coverage of a normal genomic sequence of multiplicity one). The "divergence" is calculated based on the alignment of constructed repeat consensus sequences, dividing the total number of substitutions and indels by the total number of matches, substitutions, and indels (if the forward and reverse consensus sequences do not overlap, then the mean divergence of the forward and reverse sequences is calculated, weighted by the length of the sequences). "Maximal distance between divergent positions" refers to the maximal distance between adjacent confirmed divergent positions. "Remaining gap" refers to the length of the repeat remaining without separate consensus sequences for each copy after we have "moved into the repeat" from both the forward and reverse directions (note that it is 0 if the forward and reverse consensus sequences overlap).

repeat ID	length (kb)	coverage	divergence	tentative div. pos.	confirmed div. pos.	maximal gap (kb)	remaining gap (kb)	cis reads	trans reads
EC4450-29	11	159x	7.33	657	594	0.4	0	1219	42
KN5052-10	38	98x	1.12	376	250	20.8	0	0	0
KN5052-20	31	96x	2.67	826	676	17.3	0	1	0
EC7921-6	13	82x	11.51	1629	1338	0.3	0	215	814
EC9002-3	50	137x	5.91	3401	3064	0.9	0	2460	437
EC9006-8	22	94x	1.24	218	131	11.7	0	0	0
EC9006-9	14	78x	2.81	676	597	1.8	0	256	15
EC9006-10	16	93x	5.25	2843	2610	0.8	0	912	80
EC9007-5	24	140x	0.33	2467	37	14.6	5.0	0	0
EC9012-7	14	63x	19.22	2784	2552	1.3	0	599	42
EC9012-12	37	74x	3.12	1973	1601	2.0	0	1126	13
EC9016-4	17	47x	8.45	2586	2340	2.4	0	462	34
EC9016-5	24	58x	1.30	1210	203	21.5	0.3	0	0
EC9103-4	4	131x	6.62	340	314	0.4	0	135	87
KN9657-9	36	61x	0.08	186	3	35.7	4.3	0	0
EC9964-5	34	73x	6.20	2333	2179	0.9	0	64	892
EC9964-6	22	80x	4.17	1675	1522	1.7	0	12	649
EC11022-7	30	60x	1.44	1661	1491	6.3	0	2	602
EC11022-8	25	64x	0.37	165	17	10.1	0	0	0
SA11962-6	8	159x	11.39	613	562	0.5	0	1089	16
SA11962-8*	13	214x	0.77	154	100	4.1	0	40	42
KL12158-7	13	46x	0.06	50	0	12.7	0	0	0

each repeat copy to ensure that consensus sequences are sufficiently accurate (the default value of *minCoverage* = 10). Both consensus sequences are truncated to the length of the shortest consensus sequence to prevent bias in the read recruitment process in future iterations. In the case of the *REP* repeat in the EC9964 dataset, we constructed two consensus sequences corresponding to 8.6 kb long prefixes of *REP* with divergence 9.8% (Figure 2.4). As a result, we now have two consensus sequences for the entire edge *REP* that differ in some of the first 8.6 kb but coincide in the remaining part. The two constructed consensus sequences serve as two templates for recruiting reads to specific repeat copies in successive iterations. In this way, we gradually construct the consensus sequences from only reads that have been assigned to a specific repeat copy with high confidence.

This brief description hides some details, e.g., it is not clear why we identified the set of putative divergent positions since these positions have not been mentioned in the description of the algorithm. In reality, the constructed consensus sequences of prefixes of two repeat copies may have errors since the read coverage of these prefixes may be as low as the default parameter $minCoverage = 10\times$. Indeed, the consensus sequences are expected to have a high 7.5% error rate when the coverage is as low as $5-10\times$ [Lin et al., 2016]. Since these error-prone consensus sequences serve as two templates for recruiting reads to specific repeat copies in successive iterations, the read recruitment is compromised. We thus recruit reads to specific repeat copies based only on tentative divergent positions in the repeat. Since these positions were identified based on all reads (full coverage) rather than only reads contributing to a given template (coverage as low as *minCoverage*), they provide a more reliable standard for read recruitment.

Below we provide a description of various steps during the unbridged repeat resolution:

Evaluating the tentative divergent positions. We map all classified reads again, this time to two consensus copies of the repeat (rather than a single consensus copy as in the initial iteration) to construct a more accurate alignment. We further utilize the set of tentative divergent positions that were identified at the initial stage of the algorithm. We consider the consensus sequence of each repeat copy and compare the most frequent symbols ($A,C,G,T,-$ occurring in the set of already classified reads for each repeat copy at each tentative divergent position. If the most frequent symbol at a position differs for two repeat copies, then that position is called a confirmed divergent position. The most frequent symbols of all the confirmed divergent positions for a certain repeat copy represent a "signature" of this copy. Since some positions within a repeat may not have been reached by the two consensus sequences yet, they remain classified as tentative divergent positions.

Assigning reads to various repeat copies. We now map all unclassified reads to two consensus copies of the repeat and utilize the confirmed divergent positions to assign unclassified reads to a specific repeat copy. For each read, we compute its vote for each repeat copy as the number of confirmed divergent positions at which the symbol of the read agrees with the consensus of this repeat copy (all other positions are ignored). The read is assigned to a specific repeat copy if its vote for this copy is larger than the vote for another copy by a minimum threshold (the default value is three). The read remains unassigned in the case of ties.

Constructing new consensus sequences for each repeat copy. We use all reads that have been assigned to a specific repeat copy to construct a new consensus sequence for this copy. The consensus is only constructed up to where the coverage of the reads is at least *minCoverage* in both repeat copies to ensure that consensus sequences are accurate, and then both consensus sequences are truncated to the length of the shortest consensus sequence. The algorithm then proceeds to the next iteration unless no new reads mapping to the original repeat consensus were classified or all of the consensus sequences are identical to those in the previous iteration, in which cases it terminates.

Although we discussed the algorithm as "moving forward" into the repeat (e.g., moving ahead from edges IN_1 and IN_2 in Figure 2.4), the same procedure is performed by moving backward in the opposite direction (e.g., moving backwards from edges OUT_1 and OUT_2 in Figure 2.4), or equivalently, moving forward along the reverse complement of the repeat. There are two stopping rules for the described algorithm: (i) when the prefix of the repeat resulting from moving forward overlaps with the suffix of the repeat resulting from "moving backward" and (ii) when the prefix and the suffix both stop extending but still do not overlap. At this point, a consensus sequence has been constructed for both prefix and suffix of each repeat copy and a set of confirmed divergent positions for each repeat copy has been obtained.

As the repeat consensus sequences have been extended forward and backward (Figure 2.4), this procedure may result in the emergence of linking reads, i.e., reads that are assigned to both a repeat copy originating from one of the incoming edges (IN_1 or IN_2) and a repeat copy originating from one of the outgoing edges (OUT_1 or OUT_2). Linking reads are grouped depending on which incoming/outgoing edges they are assigned to: IN_1 and OUT_1, IN_2 and OUT_2, IN_1 and OUT_2, or IN_2 and OUT_1. We further classify all linking reads into one of two categories called cis (IN_1/OUT_1 and IN_2/OUT_2) and trans (IN_1/OUT_2 and IN_2/OUT_1) since there are only two ways to resolve the repeat.

If the number of linking reads in one of the categories exceeds a threshold (the default value is five) and exceeds the number of linking reads in another category by at least a factor of two, all reads in the “winning” category are assigned to the corresponding repeat copies and the consensus of each repeat copy is computed based on all reads assigned to this copy.

If our attempts to resolve the repeat did not result in the emergence of linking reads or if the conditions above on the number of linking reads do not hold, the repeat is classified as unresolved (note that some resolvable repeats may be classified as unresolved). Note that even in the case of unresolved repeats, our algorithm still finds more accurate consensus sequences for the prefixes and suffixes of the repeat.

Our analysis of the Bacteria dataset suggests that a repeat can usually be classified as resolvable based on the following two criteria:

The divergence rate exceeds a minimum divergence threshold. Based on simulated data, we set up a minimum 0.1% divergence threshold, i.e. at least one divergent position per each 1000 nucleotides on average. When the divergence rate falls below 0.1%, there is often a shortage of reads covering multiple divergent positions, which is necessary for successful repeat resolution. To determine the minimum divergence threshold for which the repeat resolution algorithm could be applied successfully, we simulated several repeats of multiplicity two of length 10 kb, 20 kb, and 40 kb, with divergence rates ranging from 0.01% to 0.45%. Variations between the different copies of these repeats were introduced by adding substitutions and indels randomly to both copies until the desired divergence rate was reached. Next, we simulated PB reads from these repeats with coverage 100X, mean error rates of 15%, and read lengths between 5 kb and 15 kb. When the repeat resolution algorithm was applied to these datasets, we found that all simulated repeats with divergence rate greater than 0.1% were successfully resolved. We thus chose 0.1% to be the minimum divergence threshold.

The distance between consecutive putative divergent positions does not exceed the maximum distance threshold. If consecutive divergent positions are 15 kb apart but the maximal read length is 10 kb, there will be no reads spanning these positions that can be used for repeat resolution. Moreover, it turns out that the maximal read length is too optimistic of a threshold, since the repeat may still be unresolvable even if consecutive divergent positions are less than the length of a read away. For example, although there are many divergent positions in a 24 kb long repeat of multiplicity two in the EC9007 dataset, there exists a 8 kb gap between consecutive divergent positions (located at positions 15,002 and 23,150 from the start of the repeat). The repeat is classified as unresolved since there is only one read spanning this gap, which does not provide a confident pairing of the incoming and outgoing edges for this repeat. On the other hand, we found that selecting the average read length as the threshold is too lenient. Based on our analysis of the Bacteria dataset, we set the default threshold for the maximal distance between consecutive divergent positions as twice the average read length, which varies from 12 kb to 20 kb in the Bacteria datasets.

If either of the above criteria does not hold, the repeat is classified as unresolvable.

Chapter 3

Metagenome assembly using long-reads

3.1 Abstract

Long-read sequencing technologies have substantially improved the assemblies of many isolate bacterial genomes as compared to the fragmented assemblies produced from short-read technologies. However, assembling complex metagenomic datasets remains a challenge even for state-of-the-art long-read assemblers. To address this gap, we present the metaFlye assembler and demonstrate that it generates contiguous and accurate assemblies for metagenomic datasets. In contrast to short-read metagenomics assemblers, metaFlye captures many full-length 16S RNA genes within long contigs, thus providing new opportunities for analyzing the microbial "dark matter of life". We also demonstrate that metaFlye improves full-length plasmid reconstructions as well as enables co-assembly of multiple metagenomes together for comprehensive analysis of metagenomic time series.

3.2 Introduction

Bacterial genome assemblies produced from long Single Molecule Sequencing reads (generated using Pacific Biosciences or Oxford Nanopore sequencing technologies) are substantially more contiguous compared to short-read assemblies [Phillippy, 2017, Jain et al., 2018, Schmid et al., 2018]. In contrast, early long-read metagenomic studies reported lower yields and reduced read lengths compared to isolate bacterial assemblies, which made it difficult to generate high-quality assemblies and suggested that sample preparation protocols have to be optimized to utilize long reads in metagenomic studies [Tsai et al., 2016, Driscoll et al., 2017]. However, the recent improvements in high molecular weight DNA extraction techniques have enabled the sequencing of complex metagenomes with deep coverage and increased read lengths [Moss and Bhatt, 2018, Nicholls et al., 2019, Bertrand et al., 2019, Kafetzopoulou et al., 2019, Charalampous et al., 2019, Somerville et al., 2018]. Although these improved protocols have already been used for analyzing complex bacterial communities [Stewart et al., 2018, Arumugam et al., 2019, Hiraoka et al., 2019, Lin et al., 2019, Bickhart et al., 2019], there is still no specialized long-read metagenomic assembler. Indeed, although some long-read assemblers [Chin et al., 2016, Li, 2016, Koren et al., 2017, Kamath et al., 2017, Kolmogorov et al., 2019, Ruan and Li, 2019] have been applied to metagenomic datasets, none of them were designed to handle the specific challenges of metagenome assembly. This is unfortunate since long-read metagenomic assemblies have the potential to greatly improve upon the contiguity of short-read assemblies and address their inherent limitations, such as strain resolution [Goltsman et al., 2018], detection of horizontal gene transfer [Guo et al., 2015], search for new candidate phyla [Eloe-Fadrosh et al., 2016], and the sequencing of novel plasmids and viruses [Arredondo-Alonso et al., 2017, Paez-Espino et al., 2016]. Metagenomic assembly presents additional computational challenges compared to the assembly of isolates due to the highly non-uniform coverage of the various species and strains comprising the sample, the presence of long intra-genomic and inter-genomic repeats [Li et al., 2015, Nurk et al., 2017], and additional difficulties involved in plasmid and virus reconstruction [Antipov et al., 2015, Page and Seemann, 2019]. We recently developed a fast long-read genome assembler, Flye, and showed that it produces accurate and contiguous assemblies [Kolmogorov et al., 2019]. Here we describe a metaFlye algorithms for long-read metagenome assembly, benchmark it using a diverse set of bacterial communities, and demonstrate that it improves over state-of-the-art long read assemblers that were not optimized for metagenome assembly.

3.3 Methods

3.3.1 Metagenomic long-read assembly challenges

The original Flye algorithm first attempts to approximate the set of *genomic* k-mers (k-mers that appear in the genome) by selecting *solid* k-mers (high-frequency k-mer in the read-set). It further uses solid k-mers to efficiently detect overlapping reads, and build contigs [Kolmogorov et al., 2019]. This approach excludes most erroneous k-mers (that appear in reads but not in the genome) from consideration and reduces the memory footprint of the k-mer index. However, in a metagenome setting, this approach would favor high-abundance species, while low-abundance species will have a reduced number of solid k-mers (if any), and thus will fail to be assembled. To address this limitation, we introduce a new approach to solid k-mer selection, which combines global k-mer counting with analyzing local k-mer distributions (see Methods).

Flye initially constructs the repeat graph from input reads, in which each family of long genomic repeats is collapsed into a single path in the graph [Kolmogorov et al., 2019]. It further classifies edges in the repeat graph as *unique* and *repetitive* and simplifies the graph by untangling most repeat edges using *bridging* reads. The original repeat edge classification algorithm assumes the uniform coverage of unique edges, and thus is not applicable to metagenome assembly. Here we introduce a new algorithm that reliably detects repeat edges in the metagenomic assembly graph in an iterative manner.

In addition to the challenges of assembling bacterial chromosomes, there are additional difficulties in assembling short plasmids that are typically covered only by a small number of reads. Below we show that such plasmids often remain undetected by existing assemblers and describe an algorithm that recovers unassembled plasmids from long-read sequencing data.

3.3.2 Solid *k*-mer selection in metagenome assemblies

The Flye algorithm [Kolmogorov et al., 2019] selects solid *k*-mers as follows (the typical *k*-mer size is 15 or 17 nucleotides for PacBio and ONT reads). In the first pass through all reads, the algorithm counts frequencies of *k*-mer hashes using a fixed-size array of counters. In the second pass, *k*-mers with pre-computed frequencies higher than a threshold (typically equal to 2 or 3) are counted using the cuckoo hash table [Li et al., 2014]. Given the computed *k*-mer frequency table and an estimated genome size $|G|$, the algorithm selects the $|G|$ most frequent *k*-mers, and sets a frequency threshold t as the minimum frequency among the selected *k*-mers. The selected threshold t separates solid *k*-mers (that are indexed) from erroneous ones (that are discarded).

This strategy typically results in a relatively small misclassification rate; e.g., in a typical isolate bacterial project only $\approx 5\%$ of unique *genomic* *k*-mers (true *k*-mers from the genome) are missing from the set of solid *k*-mers, and only $\approx 10\%$ of unique solid *k*-mers represent non-genomic *k*-mers. However, although it works well in genomic assemblies, it is not suitable for metagenomic assemblies, because there is no frequency threshold that robustly separates genomic from non-genomic *k*-mers (due to the uneven species coverage). Below, we describe an alternative strategy for solid *k*-mer selection and benchmark it using both isolate and metagenome datasets.

Similarly to the uniform coverage mode in Flye, metaFlye also starts with counting *k*-mers in all reads. Although high-frequency *k*-mers are still expected to represent genomic *k*-m ers, non-genomic *k*-mers arising from reads in high-abundance species often outnumber genomic

k-mers from low-abundance species. Given a per-nucleotide error rate ε in reads, we estimate the probability of a k- mer in a read to be error-free as $E = e^{-k\varepsilon}$, under a Poisson error distribution model. Thus, the expected number of solid k-mers in a read is $E \cdot |read|$. For each read, metaFlye selects a frequency threshold f, so that there are at least $E \cdot |read|$ k-mers in this read with frequency at least f and indexes k-mers above this threshold using a hash table. Similarly to other k-mer counting/indexing tools, metaFlye keeps the canonical representation of each k-mer, which is defined as the lexicographical minimum of the forward and reverse-complement of the k-mer.

We evaluated the uniform and metagenome k-mer selection modes using two bacterial datasets, for which true k-mers were extracted from the available references. The first set of PacBio reads from an *E. coli* isolate (at 50× coverage) contains 254.2M (million) k-mers, out of which 56.7M (22%) are genomic. In the uniform k-mer selection mode, Flye indexed 55.3M genomic k-mers (97% of all genomic k-mers) and 5.0M non-genomic (erroneous) k-mers. In the metagenome selection mode, metaFlye indexed 50.3M genomic k-mers (89%) and 22M non-genomic k-mers.

We further used the HMP dataset (described below) to evaluate the k-mer selection in metagenome mode. We focused on the two least abundant genomes in the mixture – *B. cereus* and *R. sphaeroides* – which had coverage 2-fold below the median species coverage. These two bacteria contributed to 83M genomic k-mers in the reads. In uniform coverage mode, Flye selected only 33.2M (40%) of their genomic k-mers. In contrast, metaFlye selected 71M (86%) of genomic k-mers in metagenome coverage mode.

3.3.3 Identifying repeats in the metagenome assembly graph

In difference from contigs (that are expected to represent contiguous segments of a genome), metaFlye first builds error-prone disjointigs that represent arbitrary paths in the assembly graph, but can be generated much faster than traditional contigs. To fix potential misassemblies within disjointigs, Flye constructs the repeat graph from disjointigs by collapsing each family of long repeats into a single path in the graph [Kolmogorov et al., 2019]. metaFlye further classifies each edge of the metagenome assembly graph as *unique* (its sequence appears only once in a single genome) or *repetitive* (the edge sequence appears multiple times in a single genome or is shared by multiple genomes). Flye uses this classification to identify *bridging* reads (that start and end at different unique edges) and resolves repeats using bridging reads [Kolmogorov et al., 2019]. Thus, the contiguity of Flye assemblies critically depends on its ability to correctly classify unique and repetitive edges of the assembly graph.

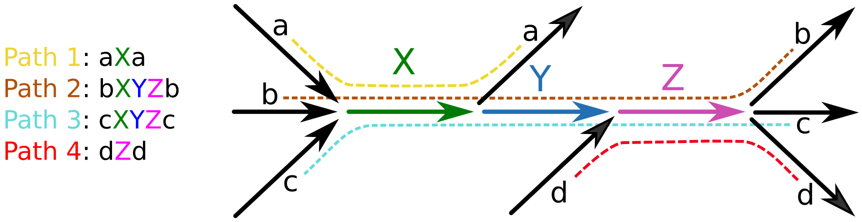


Figure 3.1: An example of a mosaic repeat. The subgraph of an assembly graph is formed by four distinct genome sub-paths. Edges are shown in color (for repeats of multiplicity 2 or 3), or in black (for unique edges of multiplicity 1). Although the edge Y is a part of the mosaic repeat, Flye may classify it as a unique edge since it has a single predecessor (X) and a single successor (Z). The metaFlye repeat detection algorithm will classify X and Z as repetitive on the first iteration (since they have three predecessors / successors). On the second iteration, Y will be classified as a repeat, since there exist reads that start at Y and continue into multiple predecessors / successors of X and Z , thus revealing that Y is a repeat.

The Flye algorithm first aligns all reads to the assembly graph, computes the mean coverage of each edge and represents all reads as *read-paths* (paths in the assembly graph). Afterwards, it captures the lion's share of the repeat edges by simply classifying all high-coverage edges (with coverage exceeding the mean coverage by a factor of 1.75) as repetitive. However, since there are possible variations in coverage along the genome, this procedure mis-classifies some repetitive edges as unique. To improve the classification of such edges, Flye additionally checks whether all read-paths through a unique edge continue into a single *successor* edge (a similar test is done for *predecessor* edges). If there are multiple successors or predecessors, the edge is re-classified as repetitive.

Although this approach works well in genomic assemblies, it is not suitable for metagenomic assemblies since the edge coverage is not a reliable predictor of the edge multiplicity. Without the coverage test, the read-paths criteria might fail to identify repetitive edges that belong to mosaic repeats, since it only checks one immediate predecessor and successor of each edge (Figure 3.1). To address this pitfall, we substitute the "diverged read-paths" approach in Flye by the "repeat detection" approach in metaFlye (described below) to identify repeat edges in the metagenome assembly graph without using coverage information.

Initially, all edges in the assembly graph are labeled as unique. The algorithm iterates through all edges and may change their classification into repetitive as described below. Thus, at each intermediate iteration, the assembly graph may contain both unique and repetitive edges.

Given a read-path through an edge *e*, metaFlye defines the next *unique* edge in this path as a *successor* of *e* (note that the original algorithm considers *any* edge as a successor). A set of all read-paths through an edge defines either a single or multiple successors. To account for chimeric reads, metaFlye filters out successors that are supported by less than $MaxSucc/delta$ reads, where *MaxSucc* is the number of reads for a successor with the highest support and *delta* is a threshold (the default value of $delta = 5$). If an edge has multiple successors or predecessors, it is classified as repetitive. The described test is performed iteratively on the entire set of edges, until no new edges are classified as repetitive.

Intuitively, in a mosaic repeat, the first iteration of the test will classify *some* of its edges as repetitive, but consecutive iterations extend the set of repeats (Figure 3.1). For a faster convergence of the algorithm, we traverse edges of the graph in the increasing order of their length, as short edges are more likely to be repetitive (two iterations are typically sufficient).

3.3.4 Assembling short plasmids

We distinguish between short (shorter than a threshold *L* with a default value of 10 kb) and long plasmids (of length at least *L*). Sequencing of short plasmids is an important task since they represent a large fraction ($\approx 30\%$) of all plasmids in the RefSeq database. However, although existing long-read assemblers perform well in assembling long circular plasmids (longer than the typical read length), our benchmarking revealed that they often miss short plasmids. Paradoxically, the longer the reads, the more plasmids remain unassembled.

To assemble short plasmids, metaFlye first aligns all reads to the assembled contigs and then extracts unaligned reads (reads with an aligned fraction below 50%). It further extracts single unaligned reads and pairs of unaligned reads that assemble into circular sequences. metaFlye focuses on single reads and pairs of reads because short plasmids are typically fully covered by a single read or a pair of reads.

Given the set of unaligned reads, metaFlye constructs a set of short *cyclocontigs* by first selecting all self-overlapping reads, i.e., reads that have overlapping prefixes and suffixes. To further extend the set of cyclocontigs, it considers all pairs of reads and selects pairs (A,B) such that B overlaps A and A overlaps B. This collection of constructed (unpolished) cyclocontigs may contain duplicates that represent the same circular plasmid. To extract unique sequences from this collection, metaFlye again performs an all-vs-all alignment of all the constructed cyclocontigs, finds similar ones and clusters them so that each cluster represents a unique circular sequence. metaFlye filters out single-read clusters (which are likely to represent artifacts from the extraction of unaligned reads). It then selects a representative for each cluster, polishes the representative using all reads contributing to the cluster as described in [Lin et al., 2016], and adds these sequences to the final assembly output.

3.4 Results

3.4.1 Benchmarking using mock metagenomic datasets

We benchmarked metaFlye, Canu [Koren et al., 2017], miniasm [Li, 2016] and wtdbg2 [Ruan and Li, 2019] using Pacific Biosciences (PacBio) and Oxford Nanopore Technology (ONT) mock metagenomic datasets, for which closely related reference genomes are available.

We also ran the FALCON assembler [Chin et al., 2016] on the PacBio datasets, but not on the ONT datasets (since FALCON requires PacBio-specific information as input). For each mock metagenome, we used metaQUAST [Mikheenko et al., 2018] to evaluate the statistics of the combined references (Table 3.1) as well as to compute the separate statistics for each species present in the sample (Figure 3.2). Figure 3.4 additionally shows NGAx plots for all datasets. Because miniasm outputs contigs with a high per-nucleotide error rate, we performed one round of contig polishing using Racon [Vaser et al., 2017].

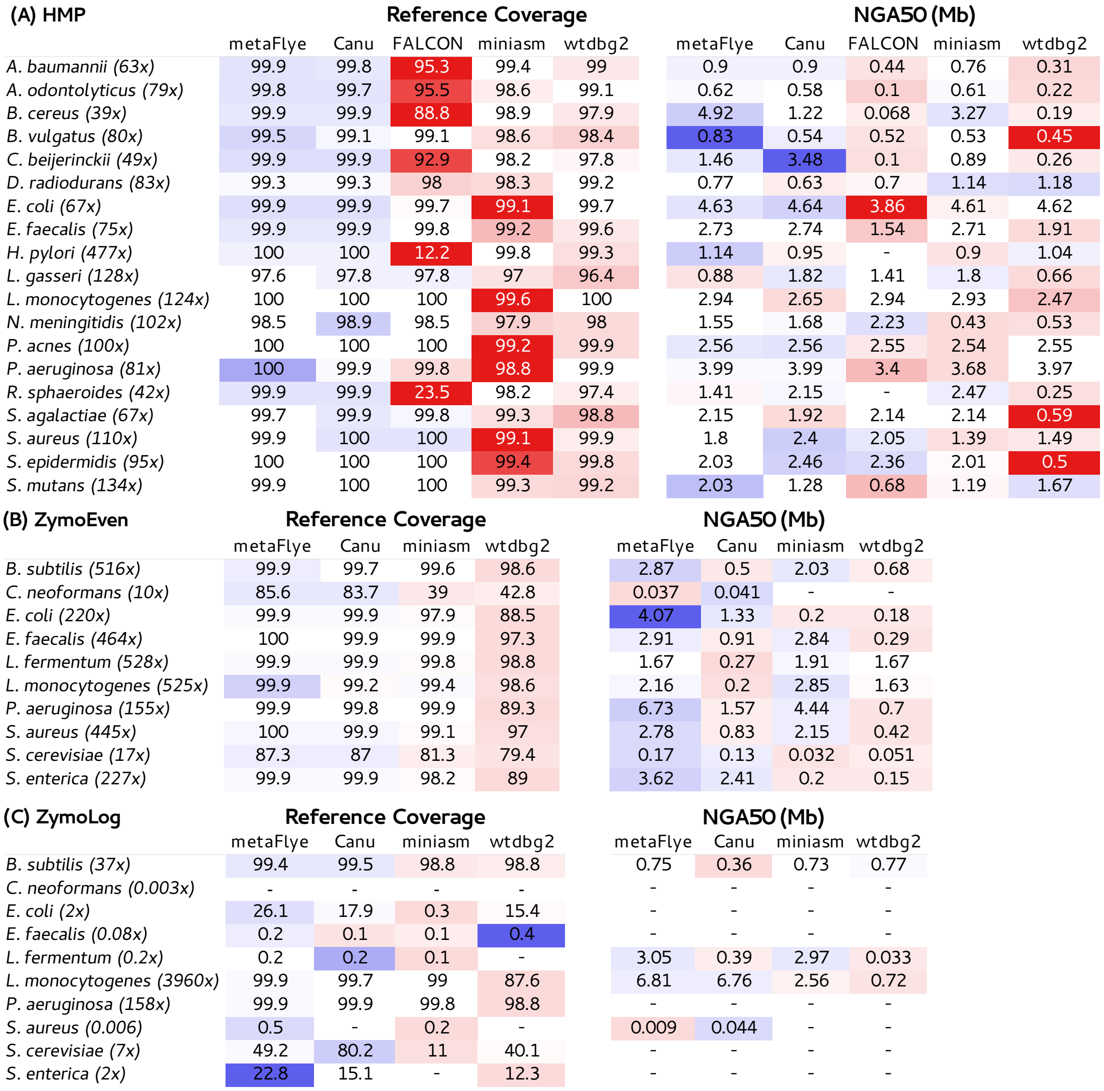
Generating assemblies. We used the following options to generate all assemblies:

- metaFlye was run using the "--meta --plasmids" option for all datasets. The minimum overlap parameter for metaFlye was manually set to 2 kb for the cow rumen assembly. We found that 13% of PacBio reads in the cow rumen dataset contained more than one PacBio subread (reads with multiple polymerase passes). To split those chimeric reads, we developed a small program called pbclip (https://github.com/fenderglass/pbclip) and applied it to the PacBio data before running metaFlye.

- Miniasm was run using its default parameters for all datasets.

- Canu was run using parameters recommended for metagenome assembly: "corOutCoverage=10000 corMhapSensitivity=high corMinCoverage=0 redMemory=32 oeaMemory=32 batMemory=200".

- Wtdbg2 was run using the default parameters for the HMP dataset. However, since the Zymo datasets had higher read coverage as well as low-abundance species, we increased the *k*-mer frequency coverage range using "--node-max 1000 -e 2" as suggested by the developers. This resulted in an increase in the total assembly length as compared to the default settings (from 28 Mb to 55 Mb for the ZymoEven dataset, and from 12.6 Mb to 23.4 Mb for the ZymoLog dataset).



(A) HMP

	Reference Coverage					NGA50 (Mb)				
	metaFlye	Canu	FALCON	miniasm	wtdbg2	metaFlye	Canu	FALCON	miniasm	wtdbg2
A. baumannii (63x)	99.9	99.8	95.3	99.4	99	0.9	0.9	0.44	0.76	0.31
A. odontolyticus (79x)	99.8	99.7	95.5	98.6	99.1	0.62	0.58	0.1	0.61	0.22
B. cereus (39x)	99.9	99.9	88.8	98.9	97.9	4.92	1.22	0.068	3.27	0.19
B. vulgatus (80x)	99.5	99.1	99.1	98.6	98.4	0.83	0.54	0.52	0.53	0.45
C. beijerinckii (49x)	99.9	99.9	92.9	98.2	97.8	1.46	3.48	0.1	0.89	0.26
D. radiodurans (83x)	99.3	99.3	98	98.3	99.2	0.77	0.63	0.7	1.14	1.18
E. coli (67x)	99.9	99.9	99.7	99.1	99.7	4.63	4.64	3.86	4.61	4.62
E. faecalis (75x)	99.9	99.9	99.8	99.2	99.6	2.73	2.74	1.54	2.71	1.91
H. pylori (477x)	100	100	12.2	99.8	99.3	1.14	0.95	-	0.9	1.04
L. gasseri (128x)	97.6	97.8	97.8	97	96.4	0.88	1.82	1.41	1.8	0.66
L. monocytogenes (124x)	100	100	100	99.6	100	2.94	2.65	2.94	2.93	2.47
N. meningitidis (102x)	98.5	98.9	98.5	97.9	98	1.55	1.68	2.23	0.43	0.53
P. acnes (100x)	100	100	100	99.2	99.9	2.56	2.56	2.55	2.54	2.55
P. aeruginosa (81x)	100	99.9	99.8	98.8	99.9	3.99	3.99	3.4	3.68	3.97
R. sphaeroides (42x)	99.9	99.9	23.5	98.2	97.4	1.41	2.15	-	2.47	0.25
S. agalactiae (67x)	99.7	99.9	99.8	99.3	98.8	2.15	1.92	2.14	2.14	0.59
S. aureus (110x)	99.9	100	100	99.1	99.9	1.8	2.4	2.05	1.39	1.49
S. epidermidis (95x)	100	100	100	99.4	99.8	2.03	2.46	2.36	2.01	0.5
S. mutans (134x)	99.9	100	100	99.3	99.2	2.03	1.28	0.68	1.19	1.67

(B) ZymoEven

	Reference Coverage				NGA50 (Mb)			
	metaFlye	Canu	miniasm	wtdbg2	metaFlye	Canu	miniasm	wtdbg2
B. subtilis (516x)	99.9	99.7	99.6	98.6	2.87	0.5	2.03	0.68
C. neoformans (10x)	85.6	83.7	39	42.8	0.037	0.041	-	-
E. coli (220x)	99.9	99.9	97.9	88.5	4.07	1.33	0.2	0.18
E. faecalis (464x)	100	99.9	99.9	97.3	2.91	0.91	2.84	0.29
L. fermentum (528x)	99.9	99.9	99.8	98.8	1.67	0.27	1.91	1.67
L. monocytogenes (525x)	99.9	99.2	99.4	98.6	2.16	0.2	2.85	1.63
P. aeruginosa (155x)	99.9	99.8	99.9	89.3	6.73	1.57	4.44	0.7
S. aureus (445x)	100	99.9	99.1	97	2.78	0.83	2.15	0.42
S. cerevisiae (17x)	87.3	87	81.3	79.4	0.17	0.13	0.032	0.051
S. enterica (227x)	99.9	99.9	98.2	89	3.62	2.41	0.2	0.15

(C) ZymoLog

	Reference Coverage				NGA50 (Mb)			
	metaFlye	Canu	miniasm	wtdbg2	metaFlye	Canu	miniasm	wtdbg2
B. subtilis (37x)	99.4	99.5	98.8	98.8	0.75	0.36	0.73	0.77
C. neoformans (0.003x)	-	-	-	-	-	-	-	-
E. coli (2x)	26.1	17.9	0.3	15.4	-	-	-	-
E. faecalis (0.08x)	0.2	0.1	0.1	0.4	-	-	-	-
L. fermentum (0.2x)	0.2	0.2	0.1	-	3.05	0.39	2.97	0.033
L. monocytogenes (3960x)	99.9	99.7	99	87.6	6.81	6.76	2.56	0.72
P. aeruginosa (158x)	99.9	99.9	99.8	98.8	-	-	-	-
S. aureus (0.006)	0.5	-	0.2	-	0.009	0.044	-	-
S. cerevisiae (7x)	49.2	80.2	11	40.1	-	-	-	-
S. enterica (2x)	22.8	15.1	-	12.3	-	-	-	-

Figure 3.2: Per-species statistics for the mock metagenomic datasets. Reference coverage and NGA50 statistics were computed using metaQUAST. The read coverage for each species are given in the brackets after the species name. NGA50 values are not reported for assemblies with reference coverage below 50%.

- FALCON was run using a configuration file recommended for bacterial assemblies.

3.4.2 Analyzing HMP assemblies

The Human Microbiome Project (HMP) mock dataset represents a mock human gut microbiome formed by 22 bacteria with known reference genomes sequenced using PacBio reads (total length 6.8 Gb and N50 = 6.7 kb). Nineteen of these bacteria have read coverages ranging from 39x (*B. cereus*) to 477× (*H. pylori*). Since the remaining three genomes (*M. smithii, C. albicans*, and *S. pneumoniae*) have low coverage (below 1×), they were excluded from further analysis.

Table 3.1: Assembly statistics and benchmarks for the mock metagenomic datasets. To evaluate the HMP assemblies, we excluded three references with coverage below 1x from the metaQUAST analysis. In the case of the ZymoEven datasets, all ten reference genomes were used. In the case of the ZymoLog GridION dataset, only four reference genomes with read coverages above 3x (*L. monocytogenes, P. aeruginosa , B. subtilis*, and *S. cerevisiae*) were used. In the case of the ZymoLog GridION dataset, five bacteria and one yeast references were used. Two yeast genomes (*S. cerevisiae* and *C. neoformans*) were excluded from the misassembly counts analysis in all Zymo datasets because of the many apparent differences between the reference and the assembled strains. Miniasm contigs were polished using Racon. Statistics were computed using metaQUAST 5.0.2 with the minimum contig length set to 5 kb. All tools were benchmarked on a computational node with 52 Intel Xeon 8164 CPUs. Reference coverage is the percentage of the reference genome covered by assembled contigs. NGA50 is the NG50 statistic computed for contigs that are broken at their misassembly breakpoints.

Dataset description	Assembler	Total length	Contigs	Reference coverage	NGA50	Mis-assemblies	CPU hours
HMP	metaFlye	66.2 Mb	85	99.8%	1.77 Mb	67	45
6.8 Gb PacBio,	Canu	68.2 Mb	180	99.7%	1.82 Mb	122	756
19 bacterial references	FALCON	60.0 Mb	388	90.3%	0.76 Mb	116	150
	miniasm	66.5 Mb	95	99.6%	1.48 Mb	74	11
	wtdbg2	65.5 Mb	187	98.7%	0.66 Mb	104	4
ZymoEven GridION	metaFlye	65.5 Mb	637	94.6%	526 kb 2	9	90
14 Gb ONT,	Canu	65.7 Mb	807	95.6%	272 kb	36	4,590
8 bacterial &	miniasm	51.9 Mb	998	80.4%	83 kb	26	67
2 yeast references	wtdbg2	54.2 Mb	1101	75.9%	75 kb	11	5
ZymoLog GridION	metaFlye	23.6 Mb	363	75.2%	407 kb	10	112
16 Gb ONT,	Canu	26.9 Mb	433	90.2%	187 kb	57	38,800
3 bacterial &	miniasm	15.6 Mb	122	56.5%	209 kb	43	299
1 yeast references	wtdbg2	23.0 Mb	668	56.6%	14 kb	20	13
ZymoLog PromethION	metaFlye	70.3 Mb	527	94.6%	647 kb	11	1,000
146 Gb ONT	wtdgb2	25.7 Mb	313	39.4%	-	30	12
8 bacterial							
2 yeast references							
ZymoLog PromethION	metaFlye	38.4 Mb	284	95.4%	3 Mb	34	4,500
148 Gb ONT,	wtdgb2	17.3 Mb	385	32.8%	-	31	16
5 bacterial							
1 yeast references							

The metaFlye, Canu and miniasm assemblies resulted in high reference coverage (ranging from 99.6% for miniasm to 99.8% for metaFlye) and NGA50 (ranging from 1.48 Mb for miniasm to 1.82 Mb for Canu). The number of misassemblies varied from 67 for metaFlye to 122 for Canu. The wtdbg2 and FALCON assemblies had reduced reference coverage (98.6% and 89.9%, respectively) and lower contiguity(NGA50 = 0.66 Mb and 0.76 Mb, respectively). The reduced coverage and contiguity were mainly associated with bacteria with abundances that deviated from the median dataset coverage the most (*B.cereus, R. shaeroides, C. beijerinckii* and *H. pylori*; see Figure 3.2), highlighting the challenge of assembling metagenomics datasets with uneven species abundance.

metaFlye assembled all 14 known plasmids that have been previously identified in the HMP dataset [Antipov et al., 2019]. Miniasm, Canu, FALCON and wtdbg2 missed one, two, four, and four plasmids, respectively. Most of the missed plasmids were shorter than 5 kb and were fully covered by a single read, illustrating the additional complications in reconstructing short plasmids. Overall, plasmid reconstruction using long reads showed substantial improvement over short-read metagenomic assemblers: the metaplasmidSPAdes short-read plasmid assembler reconstructed only seven out of the 14 plasmids from the same sample [Antipov et al., 2019].

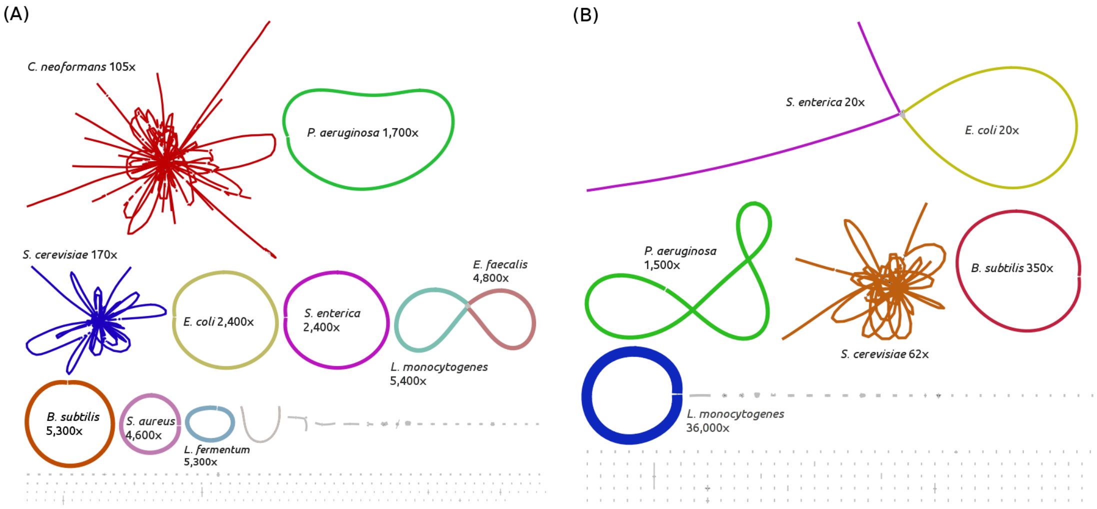


Figure 3.3: Bandage visualizations of the Zymo assemblies using metaFlye. Graph edges are colored according to their reference alignments. (a) The ZymoEven PromethION assembly. (b) The ZymoLog PromethION assembly.

3.4.3 Analyzing Zymo assemblies

The ZymoBIOMICS Microbial Community Standards datasets represent mock metagenomic datasets generated using ONT reads with an N50 of 5 kb [Nicholls et al., 2019]. The ZymoEven mock community consists of eight bacteria with abundance 12% and two yeast species with abundance 2%. The ZymoLog dataset represents the same microbial community with abundances distributed as a log scale from 89.1% (*Listeria monocytogenes*) to 0.000089% (*Staphylococcus aureus*). Each of the two communities were sequenced using GridION (total read lengths of 14 Gb and 16 Gb for the ZymoEven and ZymoLog datasets, respectively) and PromethION (total read lengths of 146 Gb and 148 Gb for the ZymoEven and ZymoLog datasets, respectively). Since the provided reference assemblies of the two yeast species (*S. cerevisiae* and *C. neoformans*) were highly fragmented, we substituted them with the closest complete reference strains from NCBI (YJM1307 and JEC21, respectively). Because of the structural differences between the references and the assembled strains, we ignored misassemblies from these yeast genomes in the total count of misassemblies.

The Canu and metaFlye assemblies of the ZymoEven GridION dataset covered 95.6% and 94.6% of the combined reference length (in contigs of length 5 kb and higher), and improved over the miniasm and wtdbg2 assemblies (80% and 76%, respectively). metaFlye, compared to Canu, showed better contiguity (NGA50 = 526 kb and 272 kb, respectively) and accuracy (29 and 36 misassemblies, respectively). Figure 3.2 illustrates that metaFlye and miniasm produced similar assemblies for most of the bacterial species; however, miniasm produced more fragmented assemblies for the yeast species.

The ZymoLog GridION dataset contains only four species with read coverage above 3×: *L. monocytogenes* (3960×), *P. aeruginosa* (158×), *B. subtilis* (38×) and *S. cerevisiae* (7×). metaFlye reconstructed over 99% of the three most abundant bacteria and 49% of *S.cerevisiae*. Miniasm assembled a smaller fraction of *S.cerevisiae* (10%), and wtdbg2 generated a highly fragmented assembly of the abundant *L. monocytogenes*. Canu produced the best coverage of the

S.cerevisiae genome (80%), however the assembly was highly fragmented. Overall, metaFlye showed the best contiguity, followed by miniasm and Canu (NGA50 = 407 Kb, 209 Kb and 187 Kb, respectively; see Figure 3.2). The number of misassemblies varied from 10 for metaFlye to 57 for Canu (Table 3.1).

metaFlye assembled PromethION runs of both ZymoEven and ZymoLog communities in 1,000 and 4,500 CPU hours, respectively (Table 3.1). In the ZymoEven dataset, all bacterial genomes but two were assembled into single circular contigs (the assemblies of *L. monocytogenes* and *E. faecalis* resulted in three contigs since they share an unresolved repeat of length 35 kb). The contiguity of the *C. neoformans* and *C. cerevisiae* assemblies (NGA50) increased by a factor of 2 as compared to the GridION assembly. For the ZymoLog dataset, the cumulative reference coverage of all species in metaFlye assembly increased from 38% to 58%. In particular, *S.cerevisiae* coverage increased from 49% to 87% and the previously unassembled *E. coli* and *S. enterica* genomes had over 99% coverage in the PromethION assembly. This improvement in the reconstruction of underrepresented species highlights the benefits of generating deep coverage datasets for long-read metagenome sequencing. Bandage visualizations [Wick et al., 2015] of the metaFlye GridION assemblies are shown in Figure 3.3.

The wtdbg2 assembly of the ZymoEven PromethION dataset was smaller than its GridION assembly (26 Mb vs 54.2 Mb, respectively), which might be a result of read subsampling procedures implemented in this assembler. Similarly, the ZymoLog assembly size was reduced from 23.4 Mb for GridION to 17.3 Mb for PromethION. Since the Canu running time on the Zymo PromethION datasets was estimated as 50,000+ CPU hours, it was impractical to run it on the available hardware. We also did not attempt to run miniasm, since the estimated size of the minimap2 alignment in PAF format was 20 Tb (projected from 200 Gb for the GridION ZymoEven dataset).

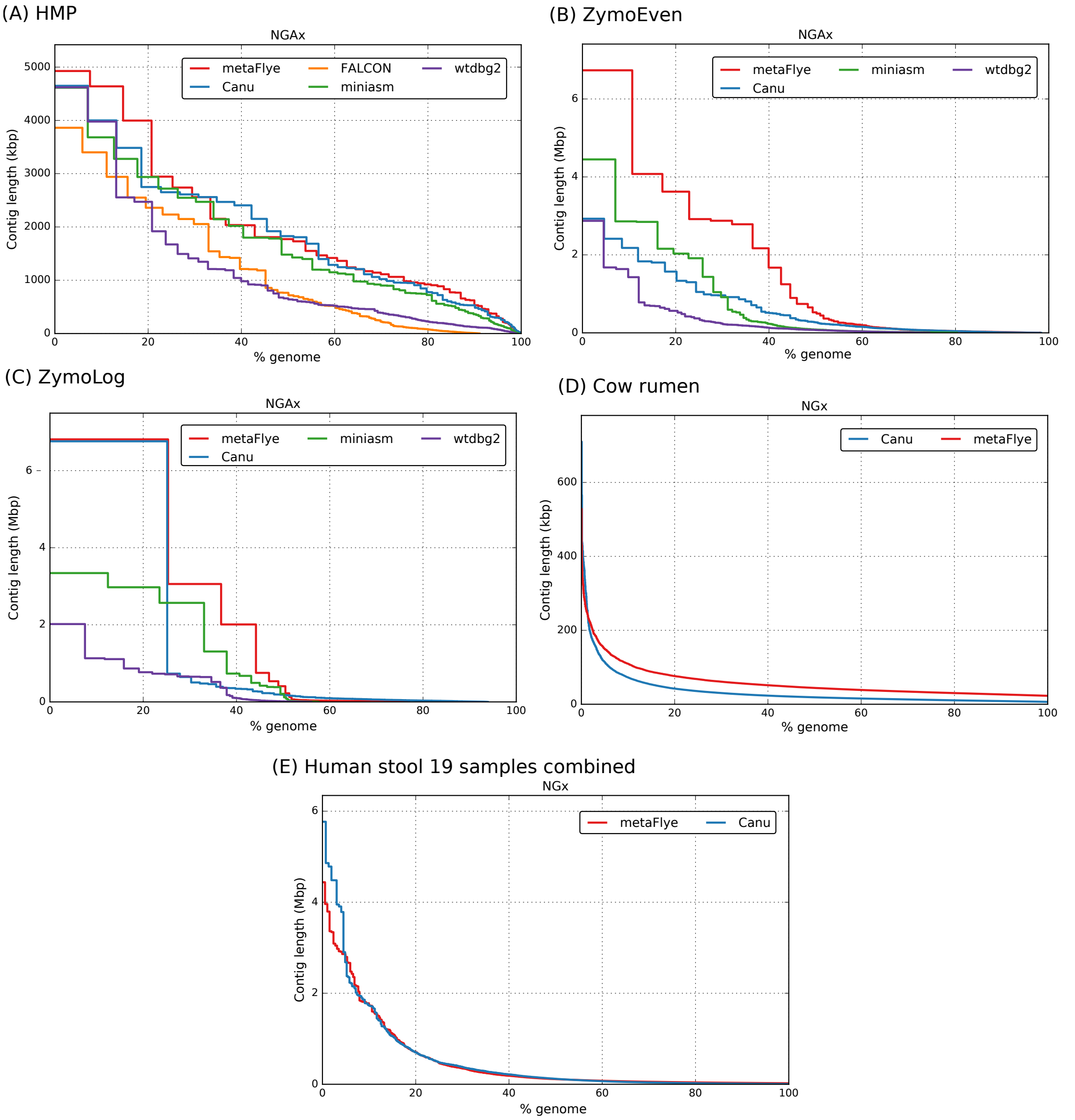


Figure 3.4: NGAx / NGx plots for different datasets generated using metaQUAST. (A-C) NGA statistics were computed for the datasets with available references. (D) NG plot of the cow rumen dataset with the genome size set to 1 Gb. (E) NG plot of the 19 human stool samples combined, with genome size set to 800 Mb.

3.4.4 Analyzing cow rumen assemblies

We assembled a cow rumen dataset consisting of PacBio reads (total read length 52.2 Gb with N50 9 kb) and compared the metaFlye assembly against the Canu assembly that was generated in the original study [Bickhart et al., 2019]. Both assemblies were polished using short reads with two rounds of Pilon polishing in indel correction mode [Walker et al., 2014]. The metaFlye and Canu assemblies had total lengths of 1,260 Mb and 1,035 Mb contained in contigs longer than 5 kb, respectively. Prodigal [Hyatt et al., 2010] predicted 1,431,527 full (38,376 partial) genes in the metaFlye assembly, and 1,191,681 full (95,679 partial) genes in the Canu assembly. The NG50 was 44 kb for metaFlye and 19 kb for Canu for a hypothetical metagenome size of 1 Gb (Figure 3.4d). The NGx statistics should be evaluated with caution since (in difference from NGAx statistics) it does not account for possible misassemblies (NGA50 statistics cannot be computed since the reference cow rumen genomes are unknown). However, our analysis of mock datasets suggest that metaFlye is more accurate than Canu.

We used Barrnap (https://github.com/tseemann/barrnap) to identify 581 and 422 full-length 16S rRNA genes in metaFlye and Canu assemblies, respectively. We further clustered these genes at 95% identity using vsearch 2.13.4 [Rognes et al., 2016] to reveal the fine-grained taxonomic composition of the microbial community. Singletons were removed, because they likely represent poorly polished copies of 16S genes rather than separate 16S genes. This clustering resulted into 105 and 78 OTUs for metaFlye and Canu assemblies, respectively (with 71 OTUs in common). Taxonomic assignment was performed against the SILVA132 full-length rRNA database [Quast et al., 2012] using the SILVA ACT service. 25 representative sequences showed $< 90\%$ alignment identity to the closest hit in the Silva database, suggesting that they likely represent novel members of *Synergistetes, Kiritimatiellaeota, Tenericutes* and *Bacteroidetes*. This analysis demonstrates that, in contrast to short-read assemblers that typically fail to reconstruct full-length 16S RNAs due to the assembly fragmentation, long-read assemblers show a great potential in analyzing the microbial “dark matter of life” [Lloyd et al., 2018] by capturing many 16S RNA genes within long contigs (up to 374 kb long in the metaFlye cow rumen assembly).

plasmidVerify [Antipov et al., 2019] and VirSorter [Roux et al., 2015] identified 52 putative plasmids and 37 viruses from the circular contigs in the metaFlye assembly. Among them, seven plasmids and two viruses were identified using only the plasmid detection algorithm aimed at short plasmids and described in the Methods section. All but three of the identified plasmids and viruses did not have significant BLAST matches (E-value < 0.001) against the NCBI database, thus potentially representing novel sequences. The Canu assembly had fewer putative plasmids among its circular contigs (39 versus 52) but more viruses (87 versus 37) as compared to the metaFlye assembly. Interestingly, there was little overlap between the plasmids (only 8) and viruses (only 10) identified in the metaFlye and Canu assemblies, suggesting that there may be a synergy between these two tools.

3.4.5 Analyzing human microbiome assemblies

[Bertrand et al., 2019] introduced a metagenome assembly pipeline OPERA-MS that combines short- and long-read assembly with MAG clustering using the available bacterial reference databases. The authors showed that OPERA-MS improves assembly contiguity by an order of magnitude as compared to short read-only methods. Below we benchmark the performance of long-read assemblers (metaFlye and Canu) using the 19 human gut metagenomic datasets generated in this study. We also demonstrate that metaFlye enables co-assembly of multiple metagenomics datasets together for analyzing metagenomic time series sampled across space and samples. We show that co-assembly increases the total assembly length while preserving its contiguity as compared to the assemblies of individual samples.

We analyzed all available samples from the ENA database (project ID: PRJEB29152) and excluded samples with low coverage depth, which resulted in 19 datasets (Table 3.2) with varying total read lengths (from 1.6 Gb to 8.0 Gb) and assembly sizes (from 5 Mb to 114 Mb in contigs that had read coverage above 5x). We used Pilon to polish each of the 19 assemblies separately using Illumina reads. In sum, metaFlye and Canu assembled 999 Mb and 835 Mb of sequence; Prodigal predicted 1,446,584 and 1,215,605 full-length genes in these assemblies, respectively. metaFlye assembled more sequence (compared to Canu) in 15 out of 19 samples (Figure 3.5a),

Table 3.2: metaFlye and Canu assemblies of 19 human stool samples. Only contigs with ONT read coverage above 5x were retained for both the metaFlye and Canu assemblies. Contigs were further polished using Pilon in indel correction mode. The NG50 statistic was calculated based on a genome size equal to the minimum of the metaFlye and Canu total assembly lengths. Genes were predicted using Prodigal.

Sample	Total read	Assembly size (Mb)		NG50 (kb)		Longest contig (Mb)		Predicted genes	
ID	length (Gb)	metaFlye	Canu	metaFlye	Canu	metaFlye	Canu	metaFlye	Canu
01	3.18	32	29	29	54	1.8	1.7	51,417	46,242
02	2.66	17	14	175	193	3.4	4.8	30,355	26,231
03	5.37	104	75	417	201	4.0	4.5	174,904	131,270
05	4.22	79	64	69	62	1.8	1.4	111,519	92,408
06	1.99	5	5	1,230	891	1.8	1.2	6,081	6,393
07	7.99	66	64	286	286	3.1	2.0	77,477	72,531
08	2.53	35	37	44	147	0.45	1.4	61,802	61,830
09	1.00	32	25	60	48	0.65	1.1	44,940	36,995
10	5.86	87	70	119	110	3.0	2.9	116,611	94,394
11	4.25	75	53	611	475	4.4	5.8	110,900	75,748
14	4.30	53	43	109	149	1.5	1.7	65,841	55,274
15	1.63	57	45	73	68	3.8	1.7	81,672	65,642
16	2.57	20	35	25	75	0.62	1.3	30,396	53,033
17	4.93	56	45	126	208	1.8	1.9	74,711	58,382
18	2.52	35	28	57	115	0.42	1.0	57,320	46,043
19	5.17	45	35	129	145	1.8	3.9	78,523	60,235
21	7.25	40	54	25	14	0.34	0.13	60,235	75,161
22	2.67	32	27	663	794	3.3	4.9	47,686	38,476
23	5.23	113	74	105	64	2.4	2.4	169,168	119,317

and the NG50 statistic was comparable for both assemblers in most of the datasets (Figure 3.5b; Figure 3.4e). metaFlye processed all samples in 1,020 CPU hours, while the Canu assembly took 15,200 CPU hours. Similarly to the cow rumen metagenome analysis, we extracted and clustered full-length 16S RNAs and revealed 109 and 84 OTUs (excluding singletons) for the metaFlye and Canu assemblies, respectively (all 84 OTUs from the Canu assembly were also recovered by metaFlye).

To investigate the sequence overlap between the samples, we used SibeliaZ [Minkin and Medvedev, 2019] to produce multi-way whole genome alignments between all samples. A contig region is called *unique* if it does not align to any other contig region in another sample, and *shared*, otherwise. The percentage of unique regions varied from 7% to 73% in various samples (Figure 3.5a), highlighting the differences in sample compositions. This analysis revealed that 510 Mb out of a total of 999 Mb in the metaFlye assembly represent non-redundant *core* sequences (400 Mb out of 835 Mb in Canu assemblies also represent core sequences).

3.4.6 Co-assembly of multiple metagenomes

To investigate how the difficulty of metagenome assembly scales with the increasing complexity of the sample, we compared *separate* metaFlye assemblies of all 19 samples with the multi-metagenomic *mixed* metaFlye assembly of all reads from all samples (comprising a total length of 75.6 Gb) that took 570 CPU hours and resulted in 549 Mb of total contig length. The mixed assembly was slightly larger than the total core sequence length of the 19 separate assemblies (510 Mb), likely due to (i) the recovery of species whose coverage was too low in the separate samples but had sufficient depth of coverage in the mixed sample, and (ii) the presence of related bacterial strains that were collapsed in the whole genome alignment. To compare the sequence content, we aligned the reads from each sample to the mixed and separate metaFlye contigs using minimap2 [Li, 2018]. The fraction of reads aligned to both the mixed and separate assemblies ranged from 42% to 80% across the 19 separate assemblies, while the fraction of reads aligned to the separate, but not the mixed assemblies ranged from 2% to 8% (Figure 3.5c). This confirms that the mixed and all of the separate assemblies (combined together) had similar sequence content. For each long contig from each separate assembly (longer than 50 Kb), we selected a contig from the mixed assembly with the best alignment to evaluate change in contiguity. Within each sample, we computed the ratio between the lengths of separate and mixed contigs for those pairs (Figure 3.5d). For all samples, the contig length rate distribution was centered around one, suggesting that on average the contiguity of the mixed assembly is comparable to the (easier) assemblies of the separate samples.

3.5 Discussion

Although long-read metagenomics is a promising direction for untangling complex bacterial communities, it faces difficult algorithmic challenges. We developed the long-read metagenomic assembler metaFlye and benchmarked it using both mock and real metagenomic communi-

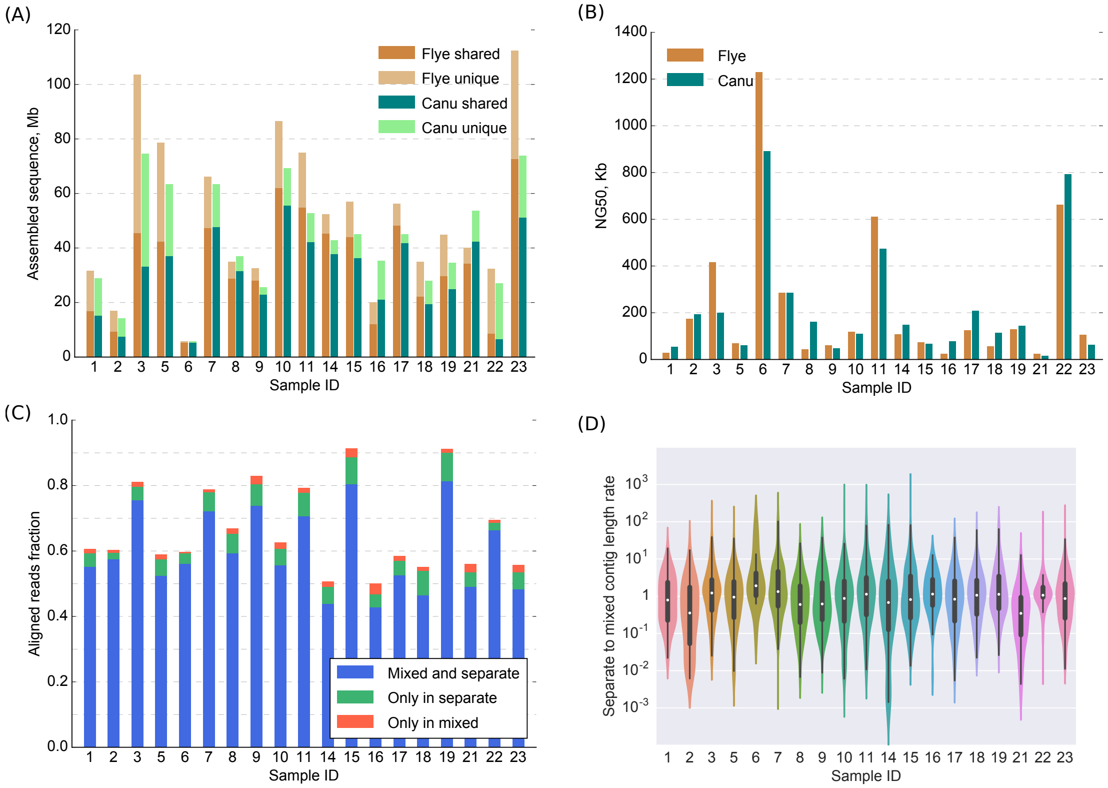


Figure 3.5: Analysis of 19 human stool metagenomic samples. The numbering of the 19 samples follows the numbering of the samples analyzed in [Bertrand et al., 2019]. (a) Total contig lengths for the metaFlye and Canu assemblies (only contigs with coverage above 5x were considered). (b) The NG50 length distribution. For each sample, the minimum of the metaFlye and Canu assembly sizes was selected as the genome size for calculating the NG50. (C) The fraction of aligned reads (with minimum alignments of at least 75% of the reads length) in metaFlye assemblies. Each read was aligned against a separate and a mixed assembly. (D) The length ratios of the contigs from the separate assemblies to the corresponding contigs from the mixed assembly. The distributions of the length ratios within each sample are centered around 1.

ties. Most long-read assemblers generated contigs covering a large fraction of the reference for the HMP mock dataset, with the metaFlye, Canu and miniasm assemblies being the most contiguous. However, miniasm and wtdbg2 tended to have difficulty assembling species with large deviations in coverage (as in the Zymo datasets), while the Canu assemblies showed reduced contiguity and an increased number of assembly errors. With respect to the running time, metaFlye was 10- to 300-fold faster than Canu on the various datasets. Only metaFlye and wtdgb2 were able to scale to the 150 Gb PromethION runs, but the wtdbg2 PromethION assemblies were less complete and more fragmented compared to its corresponding GridION assemblies.

Our analysis of the cow rumen dataset revealed that long-read assemblers greatly improve on short-read assemblers with respect to the full-length sequencing of 16S RNA genes, plasmids and viruses. The metaFlye and Canu assemblies of this dataset confirmed the trend that we observed with the mock metagenomes: that the metaFlye assembly tends to be more contiguous (in terms of the NGx statistics). Since the number of assembly errors is not known, it remains unclear what the gap is between the NGAx and NGx statistics for both assemblers.

It was recently demonstrated [Bertrand et al., 2019] that combining short and long read assembly techniques with the information from bacterial reference databases provide an order of magnitude improvement in contiguity as compared to de novo short-read assemblies. We arrived at the same conclusion for long-read-only assemblies. It is possible that combining metaFlye with an external MAG assembly pipeline (such as OPERA-MS) will further improve the assembly quality. We demonstrated that co-assembly of multiple microbiome samples improves on separate sample-by-sample assemblies with respect to total assembly length, while having comparable contiguity. This enables new approaches to long-read-based analysis of metagenomic time series.

Although metaFlye is currently limited to assembling long reads only, we plan to extend it to assembling hybrid datasets that combine long and short reads. The existing bacterial and metagenomic hybrid assemblers, such as hybridSPAdes [Antipov et al., 2015], Unicycler [Wick et al., 2017], and Opera-MS [Bertrand et al., 2019] first assemble short reads using SPAdes or metaSPAdes and further scaffold the resulting contigs by overlaying either individual or multiple long reads to resolve repeats in the short-read assembly graph. All these approaches are based on constructing the short-read metagenomic assembly graph that is significantly more fragmented than the long-read assembly graph generated by metaFlye. Thus, it seems more logical to implement an alternative hybrid metagenomic approach based on (i) assembling long-read into contigs, (ii) assembling short reads into contigs, and (iii) combining long-read and short-read contigs and assembling them together using Flye.

Software versions used.

- Flye: 2.4.2
- Canu: 1.8
- FALCON: pb-falcon 0.2.5
- Miniasm: 0.3
- Wtdbg2: 2.3
- QUAST: 5.0.2

Data availability. The described datasets are available from the corresponding locations:

- HMP mock dataset: https://github.com/PacificBiosciences/DevNet/wiki/Human_Microbiome_Project_MockB_Shotgun
- Zymo datasets: https://github.com/LomanLab/mockcommunity
- Cow rumen dataset : NCBI SRA repository under Bioproject PRJNA507739
- Human stool samples: ENA project PRJEB29152
- The assemblies and metaQUAST evaluations used in this study are available at: https://doi.org/10.5281/zenodo.2801953

Code availability. metaFlye is freely available as a part of the Flye package at: https://github.com/fenderglass/Flye. The pbclip tool for PacBio subread splitting is available from: https://github.com/fenderglass/pbclip.

3.6 Acknowledgements.

We are grateful to Denis Bertrand, Derek Bickhart, Niranjan Nagarajan, and Tim Smith for sharing the metagenomic datasets prior to journal publication.

This chapter, in full, has been submitted for a journal publication, and also appears as a bioRxiv pre-print: M. Kolmogorov, M. Rayko, J. Yuan, E. Polevikov and P. Pevzner, “metaFlye: scalable and accurate long-read metagenome assembler”, bioRxiv (2019) doi: https://doi.org/10.1101/637637. The dissertation author was the primary developer of metaFlye and first author of this paper.

Bibliography

Dmitry Antipov, Anton Korobeynikov, Jeffrey S McLean, and Pavel A Pevzner. hybridspades: an algorithm for hybrid assembly of short and long reads. *Bioinformatics*, 32(7):1009–1015, 2015.

Dmitry Antipov, Mikhail Raiko, Alla Lapidus, and Pavel A Pevzner. Plasmid detection and assembly in genomic and metagenomic data sets. *Genome research*, 29(6):961–968, 2019.

Sergio Arredondo-Alonso, Rob J Willems, Willem van Schaik, and Anita C Schürch. On the (im) possibility of reconstructing plasmids from whole-genome short-read sequencing data. *Microbial genomics*, 3(10), 2017.

Krithika Arumugam, Caner Bağcı, Irina Bessarab, Sina Beier, Benjamin Buchfink, Anna Gorska, Guanglei Qiu, Daniel H Huson, and Rohan BH Williams. Annotated bacterial chromosomes from frame-shift-corrected long-read metagenomic data. *Microbiome*, 7(1):61, 2019.

Philip M Ashton, Satheesh Nair, Tim Dallman, Salvatore Rubino, Wolfgang Rabsch, Solomon Mwaigwisya, John Wain, and Justin O'grady. Minion nanopore sequencing identifies the position and structure of a bacterial antibiotic resistance island. *Nature biotechnology*, 33(3): 296, 2015.

Nuno Bandeira, Karl R Clauser, and Pavel A Pevzner. Shotgun protein sequencing: assembly of peptide tandem mass spectra from mixtures of modified proteins. *Molecular & Cellular Proteomics*, 6(7):1123–1134, 2007.

Nuno Bandeira, Victoria Pham, Pavel Pevzner, David Arnott, and Jennie R Lill. Automated de novo protein sequencing of monoclonal antibodies. *Nature biotechnology*, 26(12):1336, 2008.

Anton Bankevich and Pavel A Pevzner. Truspades: barcode assembly of truseq synthetic long reads. *Nature methods*, 13(3):248, 2016.

Anton Bankevich, Sergey Nurk, Dmitry Antipov, Alexey A Gurevich, Mikhail Dvorkin, Alexander S Kulikov, Valery M Lesin, Sergey I Nikolenko, Son Pham, Andrey D Prjibelski, et al. Spades: a new genome assembly algorithm and its applications to single-cell sequencing. *Journal of computational biology*, 19(5):455–477, 2012.

Zhirong Bao and Sean R Eddy. Automated de novo identification of repeat sequence families in sequenced genomes. *Genome research*, 12(8):1269–1276, 2002.

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, 2015.

Denis Bertrand, Jim Shaw, Manesh Kalathiyappan, Amanda Hui Qi Ng, M Senthil Kumar, Chenhao Li, Mirta Dvornicic, Janja Paliska Soldo, Jia Yu Koh, Chengxuan Tong, et al. Hybrid metagenomic assembly enables high-resolution analysis of resistance determinants and mobile elements in human microbiomes. *Nature biotechnology*, page 1, 2019.

Derek M Bickhart, Mick Watson, Sergey Koren, Kevin Panke-Buisse, Laura M Cersosimo, Maximilian O Press, Curtis P Van Tassell, Jo Ann S Van Kessel, Bradd J Haley, Seon Woo Kim, et al. Assignment of virus and antimicrobial resistance genes to microbial hosts in a complex microbial community by combined long-read assembly and proximity ligation. *Genome Biology*, 20(1):1–18, 2019.

Adam J Bogdanove, Ralf Koebnik, Hong Lu, Ayako Furutani, Samuel V Angiuoli, Prabhu B Patil, Marie-Anne Van Sluys, Robert P Ryan, Damien F Meyer, Sang-Wook Han, et al. Two new complete genome sequences offer insight into host and tissue specificity of plant pathogenic xanthomonas spp. *Journal of Bacteriology*, 193(19):5450–5464, 2011.

Sébastien Boisvert, Frédéric Raymond, Élénie Godzaridis, François Laviolette, and Jacques Corbeil. Ray meta: scalable de novo metagenome assembly and profiling. *Genome biology*, 13 (12):R122, 2012.

Stefano R Bonissone and Pavel A Pevzner. Immunoglobulin classification using the colored antibody graph. In *International Conference on Research in Computational Molecular Biology*, pages 44–59. Springer, 2015.

Nicholas J Booher, Sara CD Carpenter, Robert P Sebra, Li Wang, Steven L Salzberg, Jan E Leach, and Adam J Bogdanove. Single molecule real-time sequencing of xanthomonas oryzae genomes reveals a dynamic structure and complex tal (transcription activator-like) effector gene relationships. *Microbial genomics*, 1(4), 2015.

Jonathan Butler, Iain MacCallum, Michael Kleber, Ilya A Shlyakhter, Matthew K Belmonte, Eric S Lander, Chad Nusbaum, and David B Jaffe. Allpaths: de novo assembly of whole-genome shotgun microreads. *Genome research*, 18(5):810–820, 2008.

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):238, 2012.

Mark JP Chaisson, John Huddleston, Megan Y Dennis, Peter H Sudmant, Maika Malig, Fereydoun Hormozdiari, Francesca Antonacci, Urvashi Surti, Richard Sandstrom, Matthew Boitano, et al. Resolving the complexity of the human genome using single-molecule sequencing. *Nature*, 517(7536):608, 2015.

Themoula Charalampous, Gemma L Kay, Hollian Richardson, Alp Aydin, Rossella Baldan, Christopher Jeanes, Duncan Rae, Sara Grundy, Daniel J Turner, John Wain, et al. Nanopore metagenomics enables rapid clinical diagnosis of bacterial lower respiratory infection. *Nature Biotechnology*, page 1, 2019.

Chen-Shan Chin, David H Alexander, Patrick Marks, Aaron A Klammer, James Drake, Cheryl Heiner, Alicia Clum, Alex Copeland, John Huddleston, Evan E Eichler, et al. Nonhybrid, finished microbial genome assemblies from long-read smrt sequencing data. *Nature methods*, 10(6):563, 2013.

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, et al. Phased diploid genome assembly with single-molecule real-time sequencing. *Nature methods*, 13(12):1050, 2016.

Phillip Compeau and Pavel Pevzner. *Bioinformatics algorithms: an active learning approach*, volume 1. Active Learning Publishers La Jolla, California, 2015.

Erin L Doyle, Barry L Stoddard, Daniel F Voytas, and Adam J Bogdanove. Tal effectors: highly adaptable phytobacterial virulence factors and readily engineered dna-targeting proteins. *Trends in cell biology*, 23(8):390–398, 2013.

Connor B Driscoll, Timothy G Otten, Nathan M Brown, and Theo W Dreher. Towards long-read metagenomics: complete assembly of three novel genomes from bacteria dependent on a diazotrophic cyanobacterium in a freshwater lake co-culture. *Standards in genomic sciences*, 12(1):9, 2017.

Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of mathematics*, 17:449–467, 1965.

Jack Edmonds and Ellis L Johnson. Matching, euler tours and the chinese postman. *Mathematical programming*, 5(1):88–124, 1973.

Emiley A Eloe-Fadrosh, David Paez-Espino, Jessica Jarett, Peter F Dunfield, Brian P Hedlund, Anne E Dekas, Stephen E Grasby, Allyson L Brady, Hailiang Dong, Brandon R Briggs, et al. Global metagenomic survey reveals a new bacterial candidate phylum in geothermal springs. *Nature communications*, 7:10476, 2016.

Jay Ghurye, Mihai Pop, Sergey Koren, Derek Bickhart, and Chen-Shan Chin. Scaffolding of long read assemblies using long range contact information. *BMC genomics*, 18(1):527, 2017.

Adrian J Gibbs and George A McIntyre. The diagram, a method for comparing sequences: Its use with amino acid and nucleotide sequences. *European journal of biochemistry*, 16(1):1–11, 1970.

Francesca Giordano, Louise Aigrain, Michael A Quail, Paul Coupland, James K Bonfield, Robert M Davies, German Tischler, David K Jackson, Thomas M Keane, Jing Li, et al. De novo yeast genome assemblies from minion, pacbio and miseq platforms. *Scientific reports*, 7 (1):3935, 2017.

Daniela S Aliaga Goltsman, Christine L Sun, Diana M Proctor, Daniel B DiGiulio, Anna Robaczewska, Brian C Thomas, Gary M Shaw, David K Stevenson, Susan P Holmes, Jillian F Banfield, et al. Metagenomic analysis with strain-level resolution reveals fine-scale variation in the human pregnancy microbiome. *Genome research*, 28(10):1467–1480, 2018.

Sara Goodwin, James Gurtowski, Scott Ethe-Sayers, Panchajanya Deshpande, Michael C Schatz, and W Richard McCombie. Oxford nanopore sequencing, hybrid error correction, and de novo assembly of a eukaryotic genome. *Genome research*, 25(11):1750–1756, 2015.

Jiangtao Guo, Qi Wang, Xiaoqi Wang, Fumeng Wang, Jinxian Yao, and Huaiqiu Zhu. Horizontal gene transfer in an acid mine drainage microbial community. *BMC genomics*, 16(1):496, 2015.

Alexey Gurevich, Vladislav Saveliev, Nikolay Vyahhi, and Glenn Tesler. Quast: quality assessment tool for genome assemblies. *Bioinformatics*, 29(8):1072–1075, 2013.

Satoshi Hiraoka, Yusuke Okazaki, Mizue Anda, Atsushi Toyoda, Shin-ichi Nakano, and Wataru Iwasaki. Metaepigenomic analysis reveals the unexplored diversity of dna methylation in an environmental prokaryotic community. *Nature communications*, 10(1):159, 2019.

John Huddleston, Swati Ranade, Maika Malig, Francesca Antonacci, Mark Chaisson, Lawrence Hon, Peter H Sudmant, Tina A Graves, Can Alkan, Megan Y Dennis, et al. Reconstructing complex regions of genomes using long-read sequencing technology. *Genome research*, 24(4): 688–696, 2014.

Doug Hyatt, Gwo-Liang Chen, Philip F LoCascio, Miriam L Land, Frank W Larimer, and Loren J Hauser. Prodigal: prokaryotic gene recognition and translation initiation site identification. *BMC bioinformatics*, 11(1):119, 2010.

Ramana M Idury and Michael S Waterman. A new algorithm for dna sequence assembly. *Journal of computational biology*, 2(2):291–306, 1995.

Zamin Iqbal, Mario Caccamo, Isaac Turner, Paul Flicek, and Gil McVean. De novo assembly and genotyping of variants using colored de bruijn graphs. *Nature genetics*, 44(2):226, 2012.

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, et al. Nanopore sequencing and assembly of a human genome with ultra-long reads. *Nature biotechnology*, 36(4):338, 2018.

Zhaoshi Jiang, Haixu Tang, Mario Ventura, Maria Francesca Cardone, Tomas Marques-Bonet, Xinwei She, Pavel A Pevzner, and Evan E Eichler. Ancestral reconstruction of segmental duplications reveals punctuated cores of human genome evolution. *Nature genetics*, 39(11): 1361, 2007.

LE Kafetzopoulou, ST Pullan, P Lemey, MA Suchard, DU Ehichioya, M Pahlmann, A Thielebein, J Hinzmann, L Oestereich, DM Wozniak, et al. Metagenomic sequencing at the epicenter of the nigeria 2018 lassa fever outbreak. *Science*, 363(6422):74–77, 2019.

Govinda M Kamath, Ilan Shomorony, Fei Xia, Thomas A Courtade, and N Tse David. Hinge: long-read assembly achieves optimal repeat resolution. *Genome research*, 27(5):747–756, 2017.

John D Kececioglu and Eugene W Myers. Combinatorial algorithms for dna sequence assembly. *Algorithmica*, 13(1-2):7, 1995.

Kristi E Kim, Paul Peluso, Primo Babayan, P Jane Yeadon, Charles Yu, William W Fisher, Chen-Shan Chin, Nicole A Rapicavoli, David R Rank, Joachim Li, et al. Long-read, whole-genome shotgun sequence data for five model organisms. *Scientific data*, 1:140045, 2014.

Mikhail Kolmogorov, Jeffrey Yuan, Yu Lin, and Pavel A Pevzner. Assembly of long, error-prone reads using repeat graphs. *Nature biotechnology*, 37(5):540, 2019.

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.

Sergey Koren, Michael C Schatz, Brian P Walenz, Jeffrey Martin, Jason T Howard, Ganeshkumar Ganapathy, Zhong Wang, David A Rasko, W Richard McCombie, Erich D Jarvis, et al. Hybrid error correction and de novo assembly of single-molecule sequencing reads. *Nature biotechnology*, 30(7):693, 2012.

Sergey Koren, Gregory P Harhay, Timothy PL Smith, James L Bono, Dayna M Harhay, Scott D Mcvey, Diana Radune, Nicholas H Bergman, and Adam M Phillippy. Reducing assembly complexity of microbial genomes with single-molecule sequencing. *Genome biology*, 14(9): R101, 2013.

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, 2017.

Jessica M Labonté, Brandon K Swan, Bonnie Poulos, Haiwei Luo, Sergey Koren, Steven J Hallam, Matthew B Sullivan, Tanja Woyke, K Eric Wommack, and Ramunas Stepanauskas. Single-cell genomics-based analysis of virus–host interactions in marine surface bacterioplankton. *The ISME journal*, 9(11):2386, 2015.

Ka-Kit Lam, Kurt LaButti, Asif Khalak, and David Tse. Finishersc: a repeat-aware tool for upgrading de novo assembly using long reads. *Bioinformatics*, 31(19):3207–3209, 2015.

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, 2015.

Heng Li. Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences. *Bioinformatics*, 32(14):2103–2110, 2016.

Heng Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34(18): 3094–3100, 2018.

Xiaozhou Li, David G Andersen, Michael Kaminsky, and Michael J Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the Ninth European Conference on Computer Systems*, page 27. ACM, 2014.

Zhenyu Li, Yanxiang Chen, Desheng Mu, Jianying Yuan, Yujian Shi, Hao Zhang, Jun Gan, Nan Li, Xuesong Hu, Binghang Liu, et al. Comparison of the two major classes of assembly algorithms: overlap–layout–consensus and de-bruijn-graph. *Briefings in functional genomics*, 11(1):25–37, 2012.

Jyun-Hong Lin, Zong-Yen Wu, Liang Gong, Chee-Hong Wong, Wen-Cheng Chao, Chun-Ming Yen, Ching-Ping Wang, Chia-Lin Wei, Yao-Ting Huang, and Po-Yu Liu. Complex microbiome in brain abscess revealed by whole-genome culture-independent and culture-based sequencing. *Journal of clinical medicine*, 8(3):351, 2019.

Yu Lin, Sergey Nurk, and Pavel A Pevzner. What is the difference between the breakpoint graph and the de bruijn graph? *BMC genomics*, 15(6):S6, 2014.

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.

Karen G Lloyd, Andrew D Steen, Joshua Ladau, Junqi Yin, and Lonnie Crosby. Phylogenetically novel uncultured microbial cells dominate earth microbiomes. *MSystems*, 3(5):e00055–18, 2018.

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, 2015.

Nicole B Lopanik, Jennifer A Shields, Tonia J Buchholz, Christopher M Rath, Joanne Hothersall, Margo G Haygood, Kristina Håkansson, Christopher M Thomas, and David H Sherman. In vivo and in vitro trans-acylation by bryp, the putative bryostatin pathway acyltransferase derived from an uncultured marine symbiont. *Chemistry & biology*, 15(11):1175–1186, 2008.

Marnix H Medema, Kai Blin, Peter Cimermancic, Victor de Jager, Piotr Zakrzewski, Michael A Fischbach, Tilmann Weber, Eriko Takano, and Rainer Breitling. antismash: rapid identification, annotation and analysis of secondary metabolite biosynthesis gene clusters in bacterial and fungal genome sequences. *Nucleic acids research*, 39(suppl_2):W339–W346, 2011.

Alla Mikheenko, Andrey Prjibelski, Vladislav Saveliev, Dmitry Antipov, and Alexey Gurevich. Versatile genome assembly evaluation with quast-lg. *Bioinformatics*, 34(13):i142–i150, 2018.

Ilia Minkin and Paul Medvedev. Scalable multiple whole-genome alignment and locally collinear block construction with sibeliaz. *BioRxiv*, page 548123, 2019.

Eli L Moss and Ami S Bhatt. Generating closed bacterial genomes from long-read nanopore sequencing of microbiomes. *bioRxiv*, page 489641, 2018.

Eugene W Myers. The fragment assembly string graph. *Bioinformatics*, 21(suppl_2):ii79–ii85, 2005.

Eugene W Myers, Granger G Sutton, Art L Delcher, Ian M Dew, Dan P Fasulo, Michael J Flanigan, Saul A Kravitz, Clark M Mobarry, Knut HJ Reinert, Karin A Remington, et al. A whole-genome assembly of drosophila. *Science*, 287(5461):2196–2204, 2000.

Gene Myers. Efficient local alignment discovery amongst noisy long reads. In *International Workshop on Algorithms in Bioinformatics*, pages 52–67. Springer, 2014.

Maria Nattestad, Sara Goodwin, Karen Ng, Timour Baslan, Fritz J Sedlazeck, Philipp Rescheneder, Tyler Garvin, Han Fang, James Gurtowski, Elizabeth Hutton, et al. Complex rearrangements and oncogene amplifications revealed by long-read dna and rna sequencing of a breast cancer cell line. *Genome research*, 28(8):1126–1135, 2018.

Samuel M Nicholls, Joshua C Quick, Shuiquan Tang, and Nicholas J Loman. Ultra-deep, long-read nanopore sequencing of mock microbial community standards. *Gigascience*, 8(5):giz043, 2019.

Sergej Nowoshilow, Siegfried Schloissnig, Ji-Feng Fei, Andreas Dahl, Andy WC Pang, Martin Pippel, Sylke Winkler, Alex R Hastie, George Young, Juliana G Roscito, et al. The axolotl genome and the evolution of key tissue formation regulators. *Nature*, 554(7690):50, 2018.

Sergey Nurk, Dmitry Meleshko, Anton Korobeynikov, and Pavel A Pevzner. metaspades: a new versatile metagenomic assembler. *Genome research*, 27(5):824–834, 2017.

Yukiteru Ono, Kiyoshi Asai, and Michiaki Hamada. Pbsim: Pacbio reads simulatortoward accurate genome assembly. *Bioinformatics*, 29(1):119–121, 2012.

David Paez-Espino, Emiley A Eloe-Fadrosh, Georgios A Pavlopoulos, Alex D Thomas, Marcel Huntemann, Natalia Mikhailova, Edward Rubin, Natalia N Ivanova, and Nikos C Kyrpides. Uncovering earths virome. *Nature*, 536(7617):425, 2016.

Andrew Page and Torsten Seemann. Tiptoft: detecting plasmids contained in uncorrected long read sequencing data. *The Journal of Open Source Software*, 4:1021, 2019.

Genis Parra, Keith Bradnam, and Ian Korf. Cegma: a pipeline to accurately annotate core genes in eukaryotic genomes. *Bioinformatics*, 23(9):1061–1067, 2007.

Pavel A Pevzner. 1-tuple dna sequencing: computer analysis. *Journal of Biomolecular structure and dynamics*, 7(1):63–73, 1989.

Pavel A Pevzner, Haixu Tang, and Michael S Waterman. An eulerian path approach to dna fragment assembly. *Proceedings of the national academy of sciences*, 98(17):9748–9753, 2001.

Pavel A Pevzner, Haixu Tang, and Glenn Tesler. De novo repeat classification and fragment assembly. *Genome research*, 14(9):1786–1796, 2004.

Son K Pham and Pavel A Pevzner. Drimm-synteny: decomposing genomes into evolutionary conserved segments. *Bioinformatics*, 26(20):2509–2516, 2010.

Adam M Phillippy. New advances in sequence assembly, 2017.

Andrey D Prjibelski, Irina Vasilinetc, Anton Bankevich, Alexey Gurevich, Tatiana Krivosheeva, Sergey Nurk, Son Pham, Anton Korobeynikov, Alla Lapidus, and Pavel A Pevzner. Exspander: a universal repeat resolver for dna fragment assembly. *Bioinformatics*, 30(12):i293–i301, 2014.

Lianrong Pu, Yu Lin, and Pavel A Pevzner. Detection and analysis of ancient segmental duplications in mammalian genomes. *Genome research*, 28(6):901–909, 2018.

Christian Quast, Elmar Pruesse, Pelin Yilmaz, Jan Gerken, Timmy Schweer, Pablo Yarza, Jörg Peplies, and Frank Oliver Glöckner. The silva ribosomal rna gene database project: improved data processing and web-based tools. *Nucleic acids research*, 41(D1):D590–D596, 2012.

Judith Risse, Marian Thomson, Sheila Patrick, Garry Blakely, Georgios Koutsovoulos, Mark Blaxter, and Mick Watson. A single chromosome assembly of bacteroides fragilis strain be1 from illumina and minion nanopore sequencing data. *Gigascience*, 4(1):60, 2015.

Torbjørn Rognes, Tomáš Flouri, Ben Nichols, Christopher Quince, and Frédéric Mahé. Vsearch: a versatile open source tool for metagenomics. *PeerJ*, 4:e2584, 2016.

Roy Ronen, Christina Boucher, Hamidreza Chitsaz, and Pavel Pevzner. Sequel: improving the accuracy of genome assemblies. *Bioinformatics*, 28(12):i188–i196, 2012.

Simon Roux, Francois Enault, Bonnie L Hurwitz, and Matthew B Sullivan. Virsorter: mining viral signal from microbial genomic data. *PeerJ*, 3:e985, 2015.

Jue Ruan and Heng Li. Fast and accurate long-read assembly with wtdbg2. *BioRxiv*, page 530972, 2019.

Steven L Salzberg, Daniel D Sommer, Michael C Schatz, Adam M Phillippy, Pablo D Rabinowicz, Seiji Tsuge, Ayako Furutani, Hirokazu Ochiai, Arthur L Delcher, David Kelley, et al. Genome sequence and rapid evolution of the rice pathogen xanthomonas oryzae pv. oryzae pxo99 a. *BMC genomics*, 9(1):204, 2008.

Michael Schmid, Daniel Frei, Andrea Patrignani, Ralph Schlapbach, Jürg E Frey, Mitja NP Remus-Emsermann, and Christian H Ahrens. Pushing the limits of de novo genome assembly for complex prokaryotic genomes harboring very long, near identical repeats. *Nucleic acids research*, 46(17):8953–8965, 2018.

Sebastian Schornack, Matthew J Moscou, Eric R Ward, and Diana M Horvath. Engineering plant disease resistance based on tal effectors. *Annual Review of Phytopathology*, 51:383–406, 2013.

Itai Sharon, Michael Kertesz, Laura A Hug, Dmitry Pushkarev, Timothy A Blauwkamp, Cindy J Castelle, Mojgan Amirebrahimi, Brian C Thomas, David Burstein, Susannah G Tringe, et al. Accurate, multi-kb reads resolve complex populations and detect rare microorganisms. *Genome research*, 25(4):534–543, 2015.

Jared T Simpson, Kim Wong, Shaun D Jackman, Jacqueline E Schein, Steven JM Jones, and Inanç Birol. Abyss: a parallel assembler for short read sequence data. *Genome research*, 19(6): 1117–1123, 2009.

Jared T Simpson, Rachael E Workman, PC Zuzarte, Matei David, LJ Dursi, and Winston Timp. Detecting dna cytosine methylation using nanopore sequencing. *Nature methods*, 14(4):407, 2017.

Vincent Somerville, Stefanie Lutz, Michael Schmid, Daniel Frei, Aline Moser, Stefan Irmler, Juerg E Frey, and Christian H Ahrens. Long read-based de novo assembly of low complex metagenome samples results in finished genomes and reveals insights into strain diversity and an active phage system. *bioRxiv*, page 476747, 2018.

Rob D Stewart, Marc D Auffret, Amanda Warr, Alan W Walker, Rainer Roehe, and Mick Watson. The genomic and proteomic landscape of the rumen microbiome revealed by comprehensive genome-resolved metagenomics. *bioRxiv*, page 489443, 2018.

Todd J Treangen, Sergey Koren, Daniel D Sommer, Bo Liu, Irina Astrovskaya, Brian Ondov, Aaron E Darling, Adam M Phillippy, and Mihai Pop. Metamos: a modular and open source metagenomic assembly and analysis pipeline. *Genome biology*, 14(1):R2, 2013.

Barry M Trost and Guangbin Dong. Total synthesis of bryostatin 16 using atom-economical and chemoselective approaches. *Nature*, 456(7221):485, 2008.

Yu-Chih Tsai, Sean Conlan, Clayton Deming, Julia A Segre, Heidi H Kong, Jonas Korlach, Julia Oh, NISC Comparative Sequencing Program, et al. Resolving the complexity of human skin metagenomes using single-molecule sequencing. *MBio*, 7(1):e01948–15, 2016.

Ajay Ummat and Ali Bashir. Resolving complex tandem repeats with long reads. *Bioinformatics*, 30(24):3491–3498, 2014.

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, 2017.

Irina Vasilinetc, Andrey D Prjibelski, Alexey Gurevich, Anton Korobeynikov, and Pavel A Pevzner. Assembling short reads from jumping libraries with large insert sizes. *Bioinformatics*, 31(20):3262–3268, 2015.

Nikolay Vyahhi, Alex Pyshkin, Son Pham, and Pavel A Pevzner. From de bruijn graphs to rectangle graphs for genome assembly. In *International Workshop on Algorithms in Bioinformatics*, pages 249–261. Springer, 2012.

Bruce J Walker, Thomas Abeel, Terrance Shea, Margaret Priest, Amr Abouelliel, Sharadha Sakthikumar, Christina A Cuomo, Qiandong Zeng, Jennifer Wortman, Sarah K Young, et al. Pilon: an integrated tool for comprehensive microbial variant detection and genome assembly improvement. *PloS one*, 9(11):e112963, 2014.

Matthias H Weissensteiner, Andy WC Pang, Ignas Bunikis, Ida Höijer, Olga Vinnere-Petterson, Alexander Suh, and Jochen BW Wolf. Combination of short-read, long-read, and optical mapping assemblies reveals large-scale tandem repeat arrays with population genetic implications. *Genome research*, 27(5):697–708, 2017.

Ryan R Wick, Mark B Schultz, Justin Zobel, and Kathryn E Holt. Bandage: interactive visualization of de novo genome assemblies. *Bioinformatics*, 31(20):3350–3352, 2015.

Ryan R Wick, Louise M Judd, Claire L Gorrie, and Kathryn E Holt. Unicycler: resolving bacterial genome assemblies from short and long sequencing reads. *PLoS computational biology*, 13(6): e1005595, 2017.

Margaret M Williams, Kathryn Sen, Michael R Weigand, Tami H Skoff, Victoria A Cunningham, Tanya A Halse, M Lucia Tondella, CDC Pertussis Working Group, et al. Bordetella pertussis strain lacking pertactin and pertussis toxin. *Emerging infectious diseases*, 22(2):319, 2016.

Daniel R Zerbino and Ewan Birney. Velvet: algorithms for de novo short read assembly using de bruijn graphs. *Genome research*, 18(5):821–829, 2008.

Aleksey V Zimin, Daniela Puiu, Ming-Cheng Luo, Tingting Zhu, Sergey Koren, Guillaume Marçais, James A Yorke, Jan Dvořák, and Steven L Salzberg. Hybrid assembly of the large and highly repetitive genome of aegilops tauschii, a progenitor of bread wheat, with the masurca mega-reads algorithm. *Genome research*, 27(5):787–792, 2017.