UNIVERSITY OF CALIFORNIA SAN DIEGO

Genome Assembly of Long Error-Prone Reads Using De Bruijn Graphs and Repeat Graphs

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

in

Bioinformatics and Systems Biology

by

Jeffrey Yuan

Committee in charge:

Professor Pavel Pevzner, Chair
Professor Bing Ren, Co-Chair
Professor Vineet Bafna
Professor Theresa Gaasterland
Professor Siavash Mirarab

2019

The Dissertation of Jeffrey Yuan is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Co-Chair

Chair

University of California San Diego

2019

EPIGRAPH

In theory, there is no difference
between theory and practice.
In practice, there is.

Benjamin Brewster

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank my advisor, Professor Pavel Pevzner for his indispensable support and guidance throughout my graduate career. His understanding of bioinformatics, his prominence in the bioinformatics world, and his intuition for the next big research topic are only matched by his kindness, his hospitality, and his dedication to his teaching and to his students.

I would like to thank Max Shen, Yu Lin, Misha Kolmogorov, Anton Bankevich, and Andrey Bzikadze for the many discussions we had, the assistance they gave, and the generally collaborative environment we were able to foster as we worked on our independent projects.

I would also like to acknowledge Professor Bing Ren and his former student Siddarth Selveraj for their support and assistance in my NSF Graduate Research Fellowship application in Fall 2013, despite my having only joined UC San Diego a few months prior.

I would also like to thank the many friends that I have made in my time at UC San Diego, whether I met them through the Bioinformatics & Systems Biology Program, Taiwanese American Professionals, or elsewhere. They gave me encouragement and support when I needed it.

I would also like to thank Professor Daniel Weinreich, my Honors Thesis advisor at Brown University, for inspiring me with his love for research and showing me that having passion for your work and keeping good company are the secrets to living a happy life.

I would also like to thank my family including my parents and my brother for believing in me and providing me support in various ways.

Chapter 1, in full, is a reformatted reprint of "Assembly of long error-prone reads using de Bruijn graphs" as it appears in *Proceedings of the National Academy of Sciences* 2016 by Yu Lin, Jeffrey Yuan, Mikhail Kolmogorov, Max W. Shen, Mark Chaisson, and Pavel A. Pevzner,

with some minor revisions and edits for improved readability. The dissertation author was a primary author of this material.

Chapter 2, in full, has been accepted for publication as "Assembly of long error-prone reads using repeat graphs" as it will appear in *Nature Biotechnology* by Mikhail Kolmogorov, Jeffrey Yuan, Yu Lin, and Pavel A. Pevzner. The material has been reformatted with some minor revisions and edits for improved readability. The dissertation author was a primary author of this material.

Chapter 3, in full, is currently being prepared for submission for publication as "DiploidFlye: haplotype phasing of long read assemblies using repeat graphs" by Jeffrey Yuan and Pavel A. Pevzner. The dissertation author is a primary author of this material.

VITA

2007-2011 Brown University
Bachelor of Science, Computational Biology, Magna cum Laude with Honors

2011-2012 Japan Exchange & Teaching Program
Assistant Language Teacher, Yokote City, Akita Prefecture, Japan

2013-2019 University of California San Diego
Doctor of Philosophy, Bioinformatics & Systems Biology

PUBLICATIONS

Kolmogorov, M., Yuan, J., Lin, Y. & Pevzner, P.A. Assembly of long error-prone reads using repeat graphs. *Nature Biotechnology*. In press.

Lin, Y.*, Yuan, J.*, Kolmogorov, M.*, Shen, M.W., Chaisson, M. & Pevzner, P.A. Assembly of long error-prone reads using de Bruijn graphs. *Proceedings of the National Academy of Sciences USA*. 2016; 113 (52): E8396-E8405.

Brandler, W.M.*, Antaki, D.*, Gujral, M.*, Noor, A., Rosanio, G., Chapman, T.R., Barrera, D.J., Lin, G.N., Malhotra, D., Watts, A.C., Wong, L.C., Estabillo, J.A., Gadomski, T.E., Hong, O., Fajardo, K.V.F., Bhandari, A., Owen, R., Baughn, M., Yuan, J., Solomon, T., Moyzis, A.G., Maile, M.S., Sanders, S.J., Reiner, G.E., Vaux, K.K., Strom, C.M., Zhang, K., Muotri, A.R., Akshoomoff, N., Leal, S.M., Pierce, K., Courchesne, E., Iakoucheva, L.M., Corsello, C. & Sebat, J. Frequency and complexity of de novo structural mutation in autism. *American Journal of Human Genetics*. 2016; 98 (4): 667-679.

Baker, C.W.*, Miller, C.R.*, Thaweethai, T., Yuan, J., Baker, M.H., Joyce, P. & Weinreich, D.M. Genetically determined variation in lysis time variance in the bacteriophage X174. *G3: Genes|Genomes|Genetics*. 2016; 6 (4): 939-955.

*These authors contributed equally to this work

ABSTRACT OF THE DISSERTATION

Genome Assembly of Long Error-Prone Reads Using De Bruijn Graphs and Repeat Graphs

by

Jeffrey Yuan

Doctor of Philosophy in Bioinformatics & Systems Biology

University of California San Diego, 2019

Professor Pavel Pevzner, Chair
Professor Bing Ren, Co-Chair

Genome assembly is the problem of reconstructing genomes from DNA sequence reads. Even the best assemblies are often fragmented due to the presence of repetitive regions in the genome. Using long, single molecule sequencing (SMS) reads can improve the contiguity of these assemblies, but still fail to resolve long repetitive regions. Furthermore, the high error rate of SMS reads poses additional difficulties for assembly, raising the question of whether the popular de Bruijn graph (DBG) approach to genome assembly can be applied to SMS reads.

First, I present ABruijn, the first genome assembler for SMS reads that follows the DBG approach. By modifying the DBG into an A-Bruijn graph, ABruijn is able to produce very polished assemblies for simple genomes such as *E. coli* and *S. cerevisiae*. However, ABruijn has some difficulties with processing very repetitive regions and very large genomes.

To address ABruijn's shortcomings, I helped to develop Flye, a DBG-based assembler for SMS reads that can be applied to large mammalian genomes such as the human genome. Flye features a much more efficient method for resolving highly repetitive regions and also generates a repeat graph, which offers a compact representation of all of the repeats in a genome. Flye further performs steps to resolve those repeats and improve the quality of the assembly, resulting in a more contiguous assembly of the human genome compared to other state-of-the-art assemblers.

Finally, I present diploidFlye, a haplotype-aware extension of Flye that is able to phase the contigs for assemblies of diploid organisms. diploidFlye takes advantage of the repeat graph generated by Flye to efficiently identify heterozygous variants and generate haplocontigs (haplotype-specific contigs) from the reads.

Overall, this dissertation presents several novel algorithms for improving the performance of the *de novo* genome assembly of long SMS reads, establishing the efficacy of the DBG approach even for error-prone SMS reads and developing a state-of-the-art assembler known as Flye with many novel features for improving the overall assembly.

INTRODUCTION

With the development of new sequencing technologies over the last two decades, there has been an explosion of sequencing data generated at lower and lower costs (Verma et al. 2016). In order to utilize this sequencing data, it must be either compared to a reference sequence—a method called *resequencing*—or assembled into longer sequences such as genomes, known as *de novo genome assembly*. Although resequencing is by far the dominant method, it is not possible without a reference sequence available and may perform poorly when there are significant variations between the query and the reference sequence or when the query maps to multiple repetitive loci in the reference (Chaisson et al. 2015; Mills et al. 2011). In contrast, *de novo* genome assembly does not require a reference sequence and would accurately capture any variations in the sequence data. In fact, genome assembly is often utilized to capture large structural variations, to reconstruct highly mutated sequences such as in cancer, or to discover unknown or very diverse datasets such as when sequencing new organisms or complex metagenomics datasets (Chaisson et al. 2015; Raphael 2012; Saxena et al. 2014). However, genome assembly is a much more difficult task than resequencing, requiring sophisticated algorithms, significant computational resources and a long runtime for large genomes. Furthermore, existing methods still often perform poorly, resulting in very fragmented assemblies or missing regions of the genome entirely (Chaisson et al. 2015; Raphael 2012). Thus, new algorithms that can improve on both the quality of the assembly and the time and memory cost for the assembly are required to improve on existing genome assembly techniques, which is the focus of this dissertation.

Most sequencing projects utilize Next Generation Sequencing (NGS) platforms, which have rapidly increased the throughput and lowered the cost of sequencing by leveraging

massively parallel sequencing techniques. These NGS platforms include Roche/454 pyrosequencing, Applied Biosciences SOLiD sequencing by ligation, Ion Torrent sequencing, and Illumina sequencing-by-synthesis (Harrington et al. 2013; Shendure et al. 2011; Verma et al. 2016). However, the DNA sequence outputs of these technologies, known as *reads*, are all short in length, between 100 – 400 base-pairs (bp) (Van Dijk et al. 2014). These short reads pose a problem for genome assembly because assemblies are usually limited by the length of genomic repetitive regions that can be resolved, which in turn depends on the read length; short read assemblies tend to result in very fragmented assemblies due to the presence of many repeats in the genome that are longer than the read length (Chaisson et al. 2015). Thus, one way to improve the quality of assembly is to start with longer reads.

In the past decade, single molecule sequencing (SMS) techniques have been developed that generate longer reads but have higher error rates. The two main platforms are Pacific Biosciences' Single Molecule Real-Time sequencing technology and Oxford Nanopore Technology's nanopore sequencing. Currently, SMS reads are on average longer than 10,000 bp but have very high error rates even exceeding 10% (Jain et al. 2016; Rhoads et al. 2015). Although these long reads would indeed be able to resolve longer repeats and thus improve genome assembly, the high error rate makes the assembly process more challenging.

Prior to the advent of SMS reads, there were two main paradigms for genome assembly: the Overlap-Layout-Consensus (OLC) approach and the de Bruijn graph (DBG) approach (Li et al. 2012). These approaches offer alternative strategies for performing assembly. The OLC approach finds all overlaps between reads, then it builds a network of overlapping reads called the overlap graph to find the structure of the genome, and then constructs the sequence by taking the consensus of the reads (Li et al. 2012; Pevzner et al. 2001). The DBG approach, on the other

hand, first breaks every read into smaller segments of length k called k-mers, then glues together identical k-mers to build a large network called the de Bruijn graph, and finally finds the sequence of the genome by tracing a path in the graph (Pevzner et al. 2001; Sohn et al. 2016).

Modern NGS short-read assemblers such as the Celera Assembler (Myers et al. 2000), the JR-Assembler (Chu et al. 2013), and SGA (Simpson et al. 2012) utilize the OLC approach; however, the DBG approach is much more popular, implemented by many more assemblers such as Velvet (Zerbino et al. 2008), ABySS (Simpson et al. 2009), AllPaths (Gnerre et al. 2011), SOAPdenovo (Luo et al. 2012), and SPAdes (Bankevich et al. 2012), to name a few. Therefore, the DBG approach must offer some advantages over the OLC approach. Indeed, the de Bruijn graph is simpler and less computationally expensive to construct than the overlap graph (exact k-mer matching is significantly easier than computing all pairwise alignments of reads); finding the path corresponding to the genome is easier in a DBG than in an overlap graph (since the genome appears as an Eulerian path rather than a Hamiltonian path); and the DBG offers an accurate representation of the repeat structure and complexity of the genome that the OLC approach does not (Kamath et al. 2017; Pevzner et al. 2004; Sohn et al. 2016).

Nevertheless, when it comes to long SMS reads, all existing assemblers, such as HGAP (Chin et al. 2013), FALCON (Chin et al. 2016) and Canu (Koren et al. 2017) rely on the OLC approach. This may be because the increased length of SMS reads renders the OLC approach more intuitive since longer reads lead to longer overlaps, but then these assemblers miss out on the advantages of the DBG approach, which has always been rather counter-intuitive. It may also be due to the commonly held belief that the high error rate of SMS reads renders the DBG approach infeasible, since the de Bruijn graph constructed from SMS reads would become very tangled by the presence of so many erroneous k-mers. However, Pevzner et al. 2004 showed that

the presence of any similarity between reads is enough to build an A-Bruijn graph, which can be used for genome assembly in the same way as a de Bruijn graph. The following chapters of this dissertation discuss how to apply the DBG approach for error-prone SMS reads and illustrate that the advantages offered by the DBG approach leads to improved genome assemblies.

The first chapter of this dissertation presents the ABruijn assembler, the first nonhybrid genome assembler for long SMS reads that utilizes the DBG approach. ABruijn builds an A-Bruijn graph on a subset of the k-mers in the reads (rather than on all k-mers as in the de Bruijn graph) and is thus able to avoid incorporating too many erroneous k-mers into the graph. Since ABruijn was developed as an initial proof-of-concept assembler, it is only applied to smaller genomes such as *E. coli*, *S. cerevisiae*, and *C. elegans*, and it is shown to obtain comparable results to other state-of-the-art assemblers such as Canu. ABruijn is also shown to perform well on complex bacterial genomes such as *Xanthomonas oryzae* as well as on lower coverage datasets. However, ABruijn runs into computational bottlenecks when it encounters highly repetitive regions, and its use is limited to relatively small organisms. To overcome these obstacles and increase the scale of genome sizes that can be assembled, Flye was developed following the same framework as ABruijn.

The second chapter describes Flye, which also utilizes an A-Bruijn graph to perform a DBG-based assembly of SMS reads. However, Flye also introduces a novel method for contig generation: rather than performing a large number of expensive calculations into resolving difficult repetitive regions like ABruijn, Flye greedily chooses an arbitrary path to produce *disjointigs*, which may contain misassemblies (falsely connected sequences). Then Flye constructs a repeat graph from these disjointigs, which is shown to be equivalent to the repeat graph constructed from the true genome or from correct contigs (pieces of assembled sequences).

This repeat graph provides a compact representation of all repetitive sequences in the genome, which is useful for visualizing the structure of the genome and for identifying the remaining unresolved repeats of the assembly. Flye produces both the repeat graph and a set of contigs generated from the repeat graph as the assembly output. The remaining unresolved regions can then be targeted for "finishing" the assembled genome by incorporating other technologies such as 10X Genomics or BioNano (Mostovoy et al. 2016). Note that the repeat graph is a natural product of the DBG approach to assembly, representing one of the major advantages of the DBG approach over the OLC approach.

In addition to constructing the repeat graph, Flye also performs additional repeat resolution steps, using spanning reads to resolve bridged repeats and using variations between different copies of the same repeat to resolve unbridged repeats. These additional steps help to simplify the repeat graph and result in more contiguous assemblies. Flye's many improvements (along with several other optimizations) allow it to assemble larger and more difficult datasets such as the human genome and complex metagenomes, producing comparable or better results than other SMS assemblers like Canu and FALCON.

Finally, in the third chapter, another method is presented for improving the assemblies produced by Flye, called diploidFlye. The genomes of more complex organisms are typically diploid, which means there are two, slightly different versions of the genome present in each organism, one from each parent (called haplotypes). Flye does not distinguish between the two parental haplotypes of the genome, but diploidFlye detects the variations between these haplotypes and constructs a separate sequence for each one, called *haplocontigs*. diploidFlye utilizes the structure of the repeat graph produced by Flye to simplify the process of phasing these haplotypes, producing improved, haplotype-aware assemblies.

References

Bankevich. A., Nurk, S., Antipov, D., Gurevich, A.A., Dvorkin, M., Kulikov, A.S., Lesin, V.M., Nikolenko, S.I., Pham, S., Prjibelski, A.D., Pyshkin, A.V., Sirotkin, A.V., Vyahhi, N., Tesler, G., Alekseyev, M.A. & Pevzner, P.A. SPAdes: A new genome assembly algorithm and its applications to single-cell sequencing. *Journal of Computational Biology*. 2012; 19 (5): 455-477.

Chaisson, M.J., Wilson, R.K. & Eichler, E.E. Genetic variation and the de novo assembly of human genomes. *Nature Reviews Genetics*. 2015; 16 (11): 627-40.

Chin, C.S., Alexander, D.H., Marks, P., Klammer, A.A., Drake, J., Heiner, C., Clum, A., Copeland, A., Huddleston, J., Eichler, E.E., Turner, S.W. & Korlach, J. Nonhybrid, finished microbial genome assemblies from long-read SMRT sequencing data. *Nature Methods*. 2013; 10 (6): 563-569.

Chin, C.S., Peluso, P., Sedlazeck, F.J., Nattestad, M., Concepcion, G.T., Clum, A., Dunn, C., O'Malley, R., Figueroa-Balderas, R, Morales-Cruz, A., Cramer, G.R., Delledonne, M., Luo, C., Ecker, J.R., Cantu, D., Rank, D.R., & Schatz, M.C. Phased diploid genome assembly with single-molecule real-time sequencing. *Nature Methods*. 2016; 13 (12): 1050-1054.

Chu, T.C., Lu, C.H., Liu, T., Lee, G.C., Li, W.H. & Shih, A.C. Assembler for de novo assembly of large genomes. *Proceedings of the National Academy of Sciences USA*. 2013; 110 (36): E3417-24.

Gnerre, S., Maccallum, I., Przybylski, D., Ribeiro, F.J., Burton, J.N., Walker, B.J., Sharpe, T., Hall, G., Shea, T.P., Sykes, S., Berlin, A.M., Aird, D., Costello, M., Daza, R., Williams, L., Nicol, R., Gnirke, A., Nusbaum, C., Lander, E.S. & Jaffe, D.B. High-quality draft assemblies of mammalian genomes from massively parallel sequence data. *Proceedings of the National Academy of Sciences USA*. 2011; 108 (4): 1513-8.

Gurevich, A., Saveliev, V., Vyahhi, N. & Tesler, G. QUAST: quality assessment tool for genome assemblies. *Bioinformatics*. 2013; 29 (8): 1072-1075.

Harrington, C.T., Lin, E.L., Olson, M.T. & Eshleman, J.R. Fundamentals of pyrosequencing. *Archives of Pathology & Laboratory Medicine*. 2013; 137 (9): 1296-303.

Jain, M., Olsen, H.E., Paten, B. & Akeson, M. The Oxford Nanopore MinION: delivery of nanopore sequencing to the genomics community. *Genome Biology*. 2016; 17 (1): 239.

Kamath, G.M., Shomorony, I., Xia, F., Courtade, T.A. & David, N.T. HINGE: long-read assembly achieves optimal repeat resolution. *Genome Research.* 2017; 27 (5): 747-756.

Koren, S., Walenz, B.P., Berlin, K., Miller, J.R., Bergman, N.H. & Phillippy, A.M. Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation. *Genome Research.* 2017; 27 (5): 722-736.

Kyriakidou, M., Tai, H.H., Anglin, N.L., Ellis, D. & Strömvik, M.V. Current Strategies of Polyploid Plant Genome Sequence Assembly. *Frontiers in Plant Science*. 2018; 9: 1660.

Li, Z., Chen, Y., Mu, D., Yuan, J., Shi, Y., Zhang, H., Gan, J., Li, N., Hu, X., Liu, B., Yang, B. & Fan, W. Comparison of the two major classes of assembly algorithms: overlap-layout-consensus and de-bruijn-graph. *Briefings in Functional Genomics*. 2012; 11 (1): 25-37.

Luo, R., Liu, B., Xie, Y., Li, Z., Huang, W., Yuan, J., He, G., Chen, Y., Pan, Q., Liu, Y., Tang, J., Wu, G., Zhang, H., Shi, Y., Liu, Y., Yu, C., Wang, B., Lu, Y., Han, C., Cheung, D.W., Yiu, S.M., Peng, S., Xiaoqian, Z., Liu, G., Liao, X., Li, Y., Yang, H., Wang, J., Lam, T.W. & Wang, J. SOAPdenovo2: an empirically improved memory-efficient short-read de novo assembler. 2012; 1 (1): 18.

Mills, R.E., Walter, K., Stewart, C., Handsaker, R.E., Chen, K., Alkan, C., Abyzov, A., Yoon, S.C., Ye, K., Cheetham, R.K., Chinwalla, A., Conrad, D.F., Fu, Y., Grubert, F., Hajirasouliha, I., Hormozdiari, F., Iakoucheva, L.M., Iqbal, Z., Kang, S., Kidd, J.M., Konkel, M.K., Korn, J., Khurana, E., Kural, D., Lam, H.Y., Leng, J., Li, R., Li, Y., Lin, C.Y., Luo, R., Mu, X.J., Nemesh, J., Peckham, H.E., Rausch, T., Scally, A., Shi, X., Stromberg, M.P., Stütz, A.M., Urban, A.E., Walker, J.A., Wu, J., Zhang, Y., Zhang, Z.D., Batzer, M.A., Ding, L., Marth, G.T., McVean, G., Sebat, J., Snyder, M., Wang, J., Ye, K., Eichler, E.E., Gerstein, M.B., Hurles, M.E., Lee, C., McCarroll, S.A. & Korbel, J.O.; 1000 Genomes Project. Mapping copy number variation by population-scale genome sequencing. *Nature*. 2011; 470: 59-65.

Mostovoy, Y., Levy-Sakin, M., Lam, J., Lam, E.T., Hastie, A.R., Marks, P., Lee, J., Chu, C., Lin, C., Džakula, Ž., Cao, H., Schlebusch, S.A., Giorda, K., Schnall-Levin, M., Wall, J.D. & Kwok, P.Y. A hybrid approach for de novo human genome sequence assembly and phasing. *Nature Methods*. 2016; 13 (7): 587-90.

Myers, E.W., Sutton, G.G., Delcher, A.L., Dew, I.M., Fasulo, D.P., Flanigan, M.J., Kravitz, S.A., Mobarry, C.M., Reinert, K.H., Remington, K.A., Anson, E.L., Bolanos, R.A., Chou, H.H., Jordan, C.M., Halpern, A.L., Lonardi, S., Beasley, E.M., Brandon, R.C., Chen, L., Dunn, P.J., Lai, Z., Liang, Y., Nusskern, D.R., Zhan, M., Zhang, Q., Zheng, X., Rubin, G.M., Adams, M.D., Venter, J.C. A whole-genome assembly of Drosophila. *Science*. 2000; 287 (5461): 2196-2204.

Pevzner, P.A., Tang, H. & Tesler, G. De novo repeat classification and fragment assembly. *Genome Research*. 2004; 14 (9): 1786-1796.

Pevzner, P.A., Tang, H. & Waterman, M.S. An Eulerian path approach to DNA fragment assembly. *Proceedings of the National Academy of Sciences USA*. 2001; 98 (17): 9748-9753.

Raphael, B. Chapter 6: Structural Variation and Medical Genomics. *PLoS Computational Biology*. 2012; 8 (12): e1002821.

Rhoads, A. & Au, K.F. PacBio Sequencing and Its Applications. *Genomics Proteomics Bioinformatics*. 2015; 13 (5): 278-89.

Saxena, R.K., Edwards, D. & Varshney, R.K. Structural variations in plant genomes. *Briefings in Functional Genomics*. 2014; 13 (4): 296-307.

Shendure, J.A., Porreca, G.J., Church, G.M., Gardner, A.F., Hendrickson, C.L., Kieleczawa, J. & Slatko, B.E. Overview of DNA sequencing strategies. *Current Protocols in Molecular Biology*. 2011; Chapter 7: Unit 7.1.

Simpson, J.T., Durbin, R. Efficient de novo assembly of large genomes using compressed data structures. *Genome Research*. 2012; 22 (3): 549-56.

Simpson, J.T., Wong, K., Jackman, S.D., Schein, J.E., Jones, S.J. & Birol, I. ABySS: A parallel assembler for short read sequence data. *Genome Research*. 2009; 19 (6): 1117-1123.

Sohn, J.I. & Nam, J.W. The present and future of de novo whole-genome assembly. *Briefings in Bioinformatics*. 2018; 19 (1): 23-40.

Van Dijk, E.L., Auger, H., Jaszczyszyn, Y. & Thermes, C. Ten years of next-generation sequencing technology. *Trends in Genetics*. 2014; 30 (9): 418-26.

Verma, M., Kulshrestha, S. & Puri, A. Genome Sequencing. In: Keith J. Ed. *Bioinformatics*. Series: *Methods in Molecular Biology*. Vol 1525. New York, NY: Humana Press; 2017.

Zerbino, D.R. & Birney, E. Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome Research*. 2008; 18 (5): 821-9.

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 the 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 OLC approaches and results in accurate genome reconstructions.

1.2 Significance Statement

When long reads generated using single-molecule sequencing (SMS) technology were first made available, most researchers were skeptical about the ability of existing algorithms to generate high-quality assemblies from long, error-prone reads. Nevertheless, recent algorithmic breakthroughs resulted in many successful SMS sequencing projects. However, as the recent assemblies of important plant pathogens illustrate, the problem of assembling long, error-prone reads is far from being resolved even in the case of relatively short bacterial genomes. We propose an algorithmic approach for assembling long error-prone reads and describe the ABruijn assembler, which results in accurate genome reconstructions.

1.3 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. Pacific Biosciences, known as the pioneer in long reads technologies can now produce accurate assemblies from these 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 even be obtained from less-accurate Oxford Nanopore reads. Advances in the assembly of long, error-prone reads also recently resulted in the accurate reconstructions of many different genomes (Koren et al. 2013; Koren et al. 2015; Lam et al. 2015; Chaisson et al. 2015; Huddleston et al. 2014; Ummat et al. 2014). However, as illustrated in Booher et al. (2015), the problem of assembling long, error-prone reads is far from being completely solved even in the case of relatively small bacterial genomes.

Previous studies of SMS assemblies were based on the overlap-layout-consensus (OLC) approach (Kececioglu et al. 1995) or the similar string graph approach (Myers 2005), which requires an all-against-all comparison of reads (Myers 2014) and remains computationally challenging (see Idury et al. 2014, Li et al. 2012, and 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 originally 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 error-prone reads. Although this is true for the original de Bruijn graph approach to assembly (Idury et al. 1995; 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 the amino acid sequences of peptides (Bandeira et al. 2007; Bandeira et al. 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 et al. 1995; Butler et al. 2008; Simpson et al. 2009; Zerbino et al. 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 et al. 2010), genotyping (Iqbal et al. 2012), and Ig classification (Bonissone et al. 2016). A-Bruijn graphs are even more general than de Bruijn graphs; for example, they also encompass breakpoint graphs, which is 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.

1.4 Methods

1.4.1 The Key Idea of the ABruijn Algorithm

The Challenge of Assembling Long Error-Prone Reads.

Given the high error rates of SMS technologies, the accurate assembly of long repeats remains challenging. Also, frequent *k*-mers dramatically increase the number of candidate overlaps, thus complicating how to choose 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 in Falcon (Chin et al. 2016). However, such masking may lead to the loss of some correct overlaps. Below, we illustrate these challenges using the *Xanthomonas* genomes as an example.

Booher et al. (2015) recently sequenced several different 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 lead to the development of novel measures for plant disease control (Schornack et al. 2013; Doyle et al. 2013). However, assembling *Xanthomonas* genomes using SMS reads remains challenging (let alone using short reads).

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 also extends beyond these *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 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 several *Xanthomonas* genomes as well as 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 $Path(String, k)$ (see Figure 1.1). Note that this somewhat unusual definition results in exactly the same de Bruijn graph as the standard definition (see Compeau et al. 2014 for details).

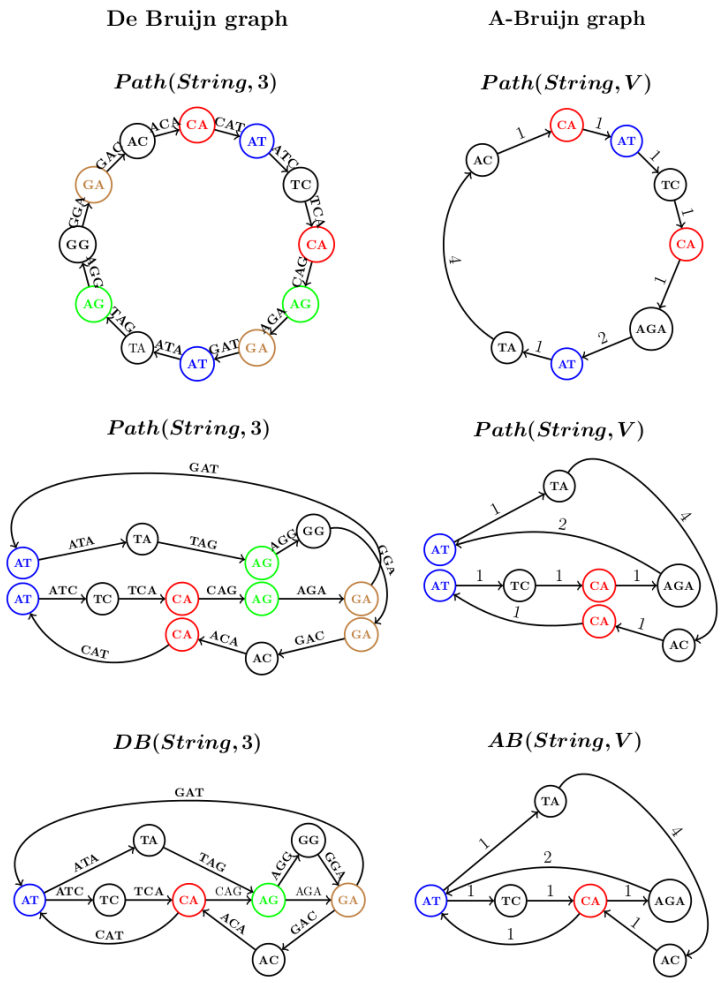

Figure 1.1: Constructing the de Bruijn graph (Left) and the A-Bruijn graph (Right) for a circular $String$ = CATCAGATAGGA . (Left) From $Path(String, 3)$ to $DB(String, 3)$. **(Right)** From $Path(String, V)$ to $AB(String, V)$ for $V = \{\text{CA}, \text{AT}, \text{TC}, \text{AGA}, \text{TA}, \text{AC}\}$. 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., it is 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 a new concept $Path(String, V)$ is defined as the path through all words from V appearing in $String$ (in order) as shown in Figure 1.1. Afterward, 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 $AB(String, \sum^{k-1})$, where $\sum^{k-1}$ stands for the set of all $(k-1)$-mers in alphabet $\sum$.

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 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 $V = \sum^{k-1}$ consists of all $(k-1)$-mers, $AB(Reads, \sum^{k-1})$ 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 of k such that one can apply various graph simplification procedures (e.g., bubble and tip removal; Pevzner et al. 2004; Zerbino et al. 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 for 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 non-genomic otherwise. Ideally, we would like to select a set of solid strings containing all genomic k-mers and no non-genomic 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 non-genomic 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 (other values make it 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 the reads), overlapping reads typically share many solid k-mers (as opposed to 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 following the path extension paradigm borrowed from some 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 base-calling errors. The described approach minimizes the number of misassemblies but results in an

inaccurate draft genome with many base-calling errors. We later describe an error-correction approach, which results in accurate genome reconstructions.

1.4.2 Assembling Long Error-Prone Reads

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 if it appears once in the genome and repeated if it appears multiple times. Figure 1.2 shows a histogram of the number of unique, repeated, and non-genomic 15-mers for a range of frequencies for the ECOLI SMS dataset (details for this dataset can be found in the "Datasets" paragraph of the Results section of this chapter). As Figure 1.2 illustrates, the lion's share of 15-mers with frequencies above a threshold 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 relatively few non-genomic k-mers while capturing most genomic k-mers.

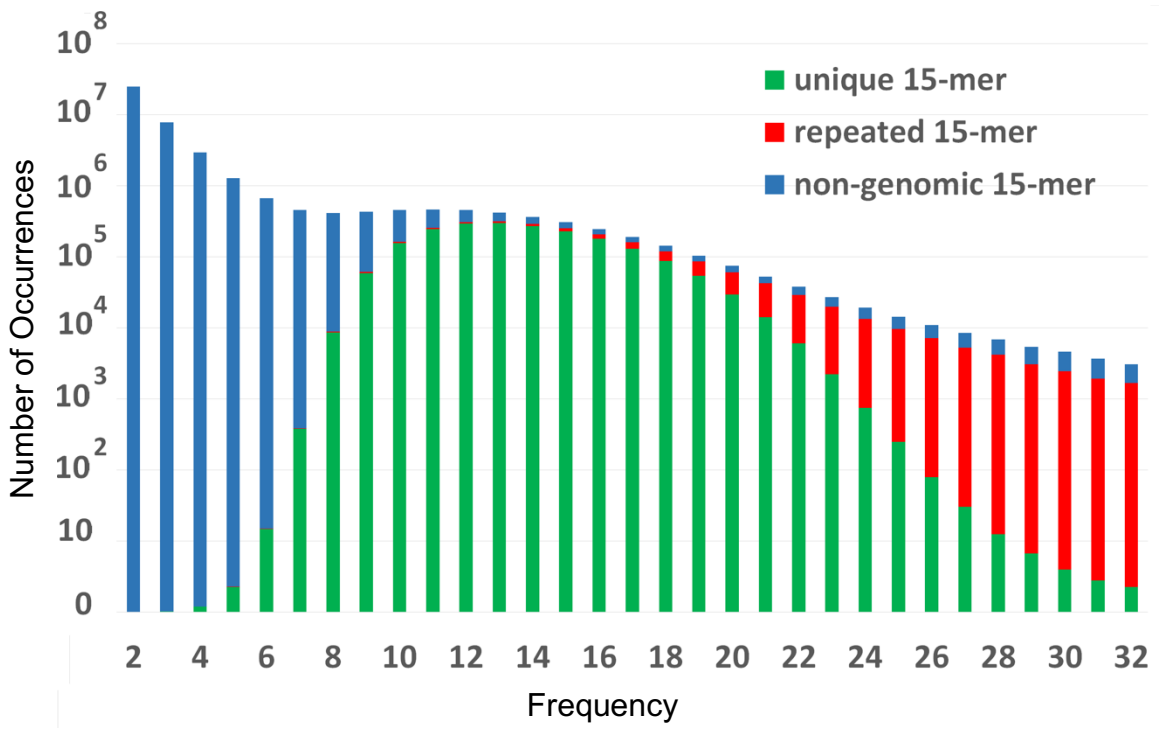

Figure 1.2: A histogram of the number of 15-mers vs frequency for the ECOLI dataset. The bars for unique, repeated, and non-genomic 15-mers for the E. coli genome are stacked and shown in green, red, and blue according to their fractions. ABruijn automatically selects the parameter t and defines solid 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 unique, repeated, and non-genomic 15-mers, respectively, for ECOLI at $t = 7$ (and 3.9, 0.1, and 0.3 million for $t = 8$). Although larger values of k (e.g. $k = 25$) also produce high-quality SMS assemblies, we found that selecting smaller, rather than larger, values for k results in slightly better performance.

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 on read-paths derived from SMS reads in the case of hybridSPAdes (Antipov et al. 2015). Recent studies of bacterial plankton (Labont 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 (24). 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 a consistent set of paths is that they are sampled from a single segment of the genomic path in the assembly graph (as opposed to an inconsistent set of paths that are sampled from multiple segments of the genomic path; see 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 (Figure 1.3) 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).

```
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
        OverlapPaths ← ReadPaths_minOverlap(GrowingPath)
        if the set OverlapPaths is consistent
          if ConsensusPath(OverlapPaths) contains InitialPath
              return the string spelled by GrowingPath (as the complete genome)
          if ConsensusPath(OverlapPaths) overlaps with GrowingPath
                extend GrowingPath by ConsensusPath(OverlapPaths)
        else
          return the string spelled by GrowingPath (as one of the contigs)
```

Figure 1.3: The pseudocode for hybridSPAdes.
This simplified sketch of the hybridSPAdes pseudocode shows how to construct contigs starting with an assembly graph. It is important to check that $OverlapPaths$ is consistent to ensure that they correspond to a single genomic path segment before including them in the resulting contigs.

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 (Figure 1.4).

```
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
```

Figure 1.4: The starting three lines of the pseudocode for longSPAdes.
These are the only lines that differ between the pseudocode for longSPAdes and that of hybridSPAdes. The rest of the pseudocode is identical to the remaining lines shown in Figure 1.3.

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 required before performing assembly using 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 (see Figure 1.5 for an example of bubble in the A-Bruijn graph). 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.

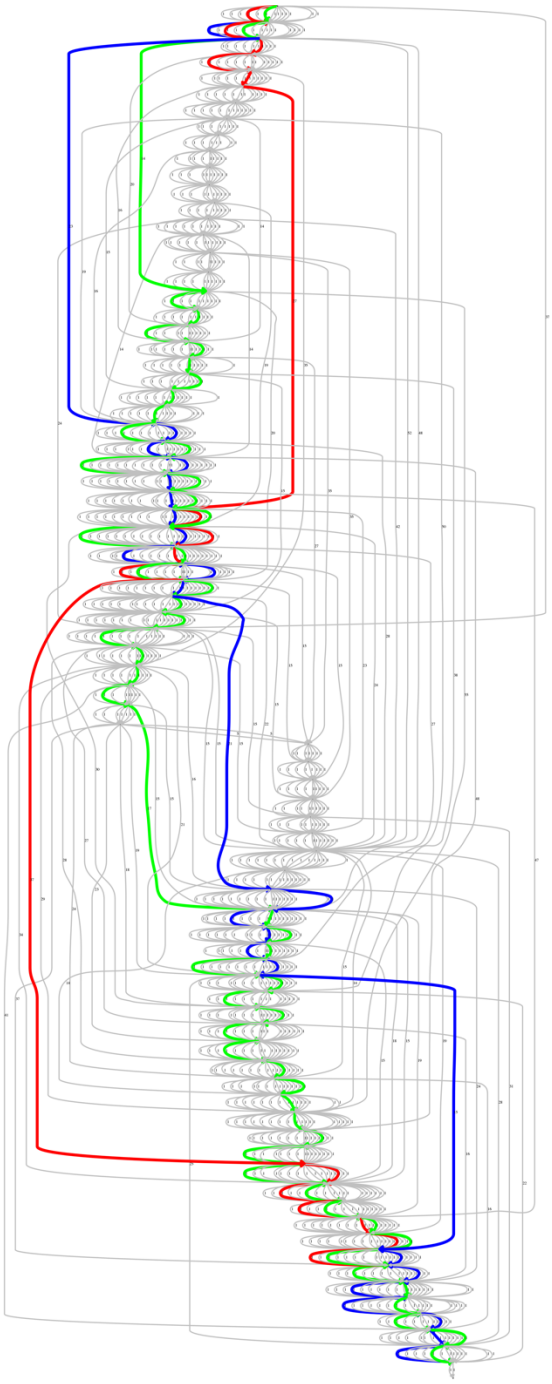

Figure 1.5: A bubble in the A-Bruijn graph of (15, 7)-mers for the ECOLI dataset.
The A-Bruijn graph was constructed using (15, 7)-mers for the ECOLI dataset. This small subgraph of the A-Bruijn graph shows 82 (15, 7)-mers appearing in segments of 61 reads covering a short 100-nucleotide region (starting at position 2,000,000 in the E. coli genome). Three out of 61 read-paths are highlighted in blue, red, and green. The complexity of this small subgraph illustrates how difficult it would be to correct the A-Bruijn graph.

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 (Figure 1.6) uses the same terms "overlapping" and "consistent" as longSPAdes, these notions are defined differently in the context of the A-Bruijn graph. These new notions (as well as the parameters $jump$ and $maxOverhang$) are described below.

```
ABruijn(LongReads, k, t, minOverlap, jump, maxOverhang)
  construct the A-Bruijn 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
    OverlapPaths ← all paths in ReadPaths overlapping ReadPath
                   (w.r.t. minOverlap, jump and maxOverhang)
    if the set OverlapPaths is consistent
      if InitialPath is a consistent path in OverlapPaths
        return the string spelled by GrowingPath (as the complete genome)
      ConsensusPath ← a most-consistent path in OverlapPaths
      extend GrowingPath by ConsensusPath
      ReadPath ← ConsensusPath
    else
      return the string spelled by GrowingPath (as one of the contigs)
```

Figure 1.6: The pseudocode for ABruijn.
The pseudocode for ABruijn is similar to that of longSPAdes, but there are significant differences in the definitions for the terms "consistent" and "overlapping." Furthermore, there are additional parameters $jump$ and $maxOverhang$ specific to ABruijn.

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 above 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 iteratively construct other contigs. Also, ABruijn attempts to extend the path to the "left" if the path extension to the "right" halts. Other complications such as removing chimeric reads are also omitted.

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 $d_P(v, w)$ 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 $v_1 \ldots v_t$ in P such that $d_P(v_i, v_{i+1}) \leq jump$ 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 1.7). 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 non-symmetric 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).

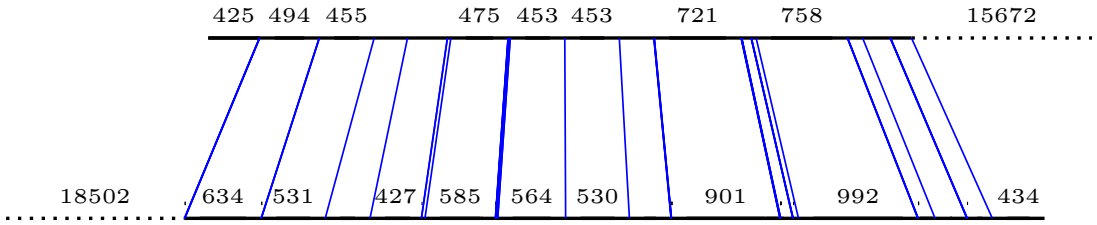

Figure 1.7: An example of a common *jump*-subpath from the ECOLI dataset.
Two overlapping reads from the ECOLI dataset and their common $jump$-subpath with maximum span that contains 50 vertices and has span 6,714 with respect to the bottom read (for $jump = 1{,}000$). The left and right overhangs for 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 $d_P(w, v) \leq jump$.

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:

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

Given all paths sharing vertices with a path P, common $jump$-subpaths with maximum span with P for all of them can be computed using a single scan of P. Below we describe a fast heuristic for this procedure.

A Fast Heuristic for Finding a Common $jump$-Subpath with Maximum Span.

We define $Predecessors_{jump}(v)$ as the set of all jump-predecessors of a vertex v in paths P_1 and P_2. A vertex w in $Predecessors_{jump}(v)$ is called dominant if it is not a $jump$-predecessor of any other vertex in $Predecessors_{jump}(v)$. If paths P_1 and P_2 traverse $Predecessors_{jump}(v)$ in the same order, then there is one dominant vertex in $Predecessors_{jump}(v)$, 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 $Predecessors_{jump}(v)$.

Our use of k-mers to identify overlapping reads has similarities with MHAP (Berlin et al. 2015), which 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.

($jump$, Δ)- Subpaths.

ABruijn uses a more restricted notion of the common $jump$-subpath described below. Given a parameter Δ (the default value is $jump/2$), a sequence of vertices $v_1 \dots v_t$ in a weighted directed graph is called a common $(jump, \Delta)$-subpath of paths P_1 and P_2 if it is a $jump$-subpath of both P_1 and P_2, and $\left|d_{P_1}(v_i, v_{i+1}) - d_{P_2}(v_i, v_{i+1})\right| \leq \Delta$ for $1 \leq i < t$. The concept $Path_{jump,\Delta}(P_1, P_2)$ is defined similarly to the concept $Path_{jump}(P_1, P_2)$.

We found that using $(jump, \Delta)$-subpaths results in slightly more accurate assemblies of highly repetitive genomes. Common $(jump, \Delta)$-subpaths with maximum span can be computed using a recurrence that is similar to the one for common $jump$-subpaths. Below we will revert to

using common $jump$-subpaths rather than common $(jump, \Delta)$-subpaths for the sake of simplicity.

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.

Choice of Parameters in the ABruijn Algorithm.

Given parameters k, t, $jump$ and Δ, we define the following statistics (Table 1.1):

- $Pr^+(k, t, jump)$: the probability that two overlapping reads share a (k, t)-mer along a region of length $jump$ in their overlap. To ensure that the notion of a common $jump$-

subpath indeed detects overlapping reads, ABruijn selects parameters k, t, and $jump$ in such a way that $Pr^{+}(k, t, jump)$ is large.

- $Pr^{-}(k, t, jump)$: the probability that two regions of length jump from two non-overlapping reads share a (k, t)-mer. To ensure that the notion of the common $jump$-subpath does not detect non-overlapping reads, ABruijn selects parameters k, t, and $jump$ in such a way that $Pr^{-}(k, t, jump)$ is small.
- $Pr^{*}(jump, \Delta)$: the probability that two overlapping reads differ by at most Δ in length along a region of length $jump$ in their overlap.

ABruijn uses the default parameters $k = 15$, $jump = 1500$, $\Delta = jump/2$, $maxOverhang = 1500$ and $minOverlap = 5000$ for all datasets and automatically selects parameter t. For the ECOLI dataset, it results in $Pr^{+}(15, 7, 1500) = 0.98$, $Pr^{-}(15, 7, 1500) = 0.002$, and $Pr^{*}(jump, \Delta) = 0.97$. For the ECOLI$_{nano}$ dataset, it results in $t = 4$, $Pr^{+}(15, 4, 1500) = 0.97$, $Pr^{-}(15, 4, 1500) = 0.002$, and $Pr^{*}(jump, \Delta) = 1.00$. Increasing the default parameter $k = 15$ to 17 and 19 result in assemblies of similar quality (with the exception of sequencing projects with low coverage).

Table 1.1: The empirical estimates of $Pr^{+}(k,t,jump)$ and $Pr^{-}(k,t,jump)$ under different choices of parameters k, t, and $jump$. The estimates are based on statistics from 10,000 pairs of overlapping reads (to estimate $Pr^{+}(k,t,jump)$) and 10,000 pairs of non-overlapping reads (to estimate $Pr^{-}(k,t,jump)$) from the ECOLI dataset and the ECOLI$_{nano}$ dataset.

	k	t	$jump$	$Pr^{+}(k,t,jump)$	$Pr^{-}(k,t,jump)$
ECOLI	15	7	1500	0.98	0.002
	17	6	1500	0.98	0.002
	19	5	1500	0.98	0.002
	21	4	1500	0.98	0.002
ECOLI$_{nano}$	15	4	1500	0.97	0.002
	17	4	1500	0.97	0.001
	19	3	1500	0.97	0.001
	21	3	1500	0.96	0.001

Additional Complications with the Implementation of the Path Extension Paradigm.

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 (Table 1.1 describes statistics on spurious overlaps). Below we describe how ABruijn addresses these complications.

Detecting chimeric reads.

The traditional way to identify a chimeric read in the de Bruijn graph framework (when the reference genome is not known) is to detect a chimeric junction in this read, i.e., a junction that improperly connects two non-adjacent segments of the genome. The existing assembly

algorithms often classify a position in the read as a chimeric junction if it is not covered by (or poorly covered by) alignments of this read with other reads. However, while this approach works for accurate reads, it needs to be modified for inaccurate reads since alignment artifacts make it difficult to identify chimeric junctions.

Traditional de Bruijn graph assemblers classify a read as chimeric if one of the edges in its read-path in the assembly graph has low coverage. They further remove the chimeric reads and corresponding edges from the assembly graph (see Nurk et al. 2013 for more advanced approaches to the detection of chimeric reads). To generalize this approach to A-Bruijn graphs, we need to redefine the notion of coverage for A-Bruin graphs.

An edge (v, w) in a path P is called internal if the distances from v to the start of P and from w to the end of P exceed $jump$, and strongly internal if those distances exceed $jump + maxOverhang$. Given overlapping paths P and P', we define the *P-spread* of P' as the sub-path of P starting and ending at the first and last vertices of $Path_{jump}(P, P')$.

To check if a path P in the A-Bruijn graph is chimeric, we consider all paths $Paths$ that overlap with this path and further trim the non-internal edges of these paths, resulting in a set of paths that we refer to as $TrimmedPaths$. The coverage of an edge in path P is defined as the number of paths in $TrimmedPaths$ whose P-spread contain this edge. A path is called chimeric if one of its strongly internal edges has coverage below 10% of the coverage of its neighboring edge.

Most-Consistent Paths.

Given a path P in a set of paths $Paths$, we define $rightSupport_{Paths}(P)$ as the number of paths in $Paths$ that support P. $leftSupport_{Paths}(P)$ is defined as the number of paths

in $Paths$ that are supported by P. We also define $Support_{Paths}(P)$ as the minimum of $rightSupport_{Paths}(P)$ and $leftSupport_{Paths}(P)$. A path P is most-consistent if it maximizes $Support_{Paths}(P)$ among all paths in $Paths$ (Figure 1.8, 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 $Paths$.

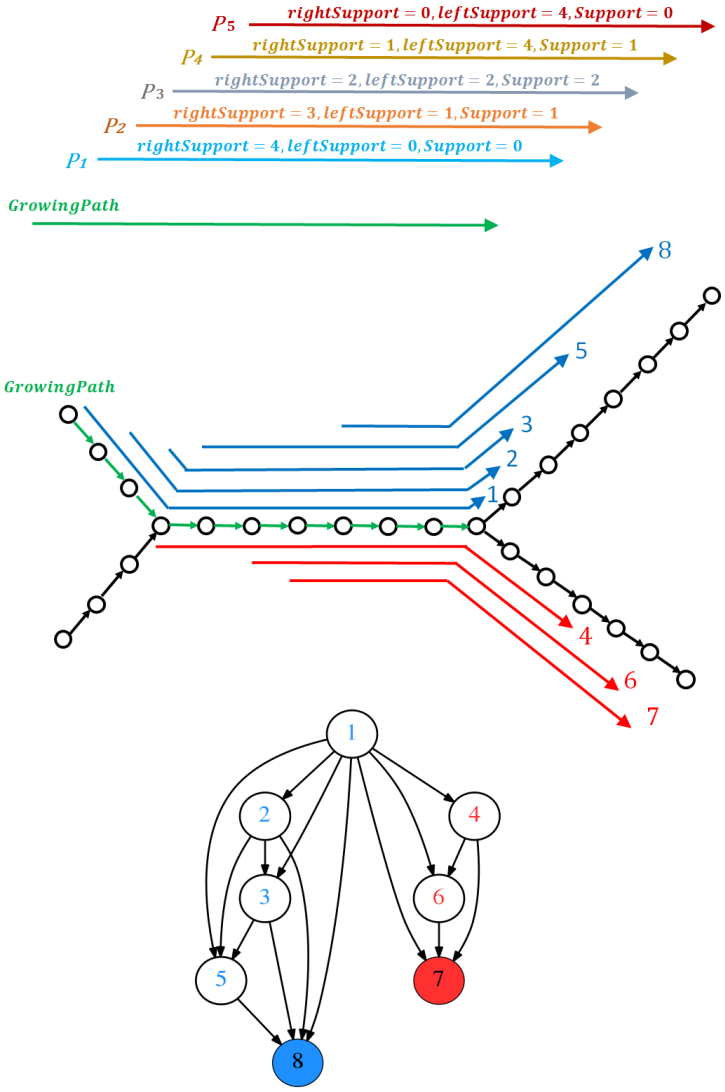

Figure 1.8: Path support and most-consistent paths.
(Top) A growing path (shown in green) and a set of five paths $Paths$ above it (extending this path). The gray path with $Support_{Paths}(P) = 2$ is the most-consistent path in the set $Paths$. **(Middle)** A growing path (shown in green) ending in 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).

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 when the growing path contains a repeat (Prjibelski et al. 2014; Vasilinetc et al. 2015). Figure 1.8, 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 1.8, 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 1.8, Bottom shows a support graph with eight vertices (each vertex corresponds to a read-path in Figure 1.8, Middle). There is an edge from a vertex v to a vertex w in this graph if 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 (the cluster of blue vertices in Figure 1.8, 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 (the cluster of red vertices in Figure 1.8, 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 (which would lead to multiple clusters) or not (which would lead to a single cluster).

Although exSPAnder and ABruijn face a similar challenge when analyzing repeats, the A-Bruijn 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 1.8. 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 $G*(Reads)$, using the Floyd–Warshall algorithm. Figure 1.9 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.

ABruijn partitions the set of vertices in the graph $G*(Reads)$ into non-overlapping clusters as follows. It selects a vertex v with maximum indegree in $G*(Reads)$ and, if this indegree exceeds a threshold (the default value is 1), it 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 1.9 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 $G^*(Reads)$ 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 the ECOLI and BLS datasets, respectively, end in repeats.

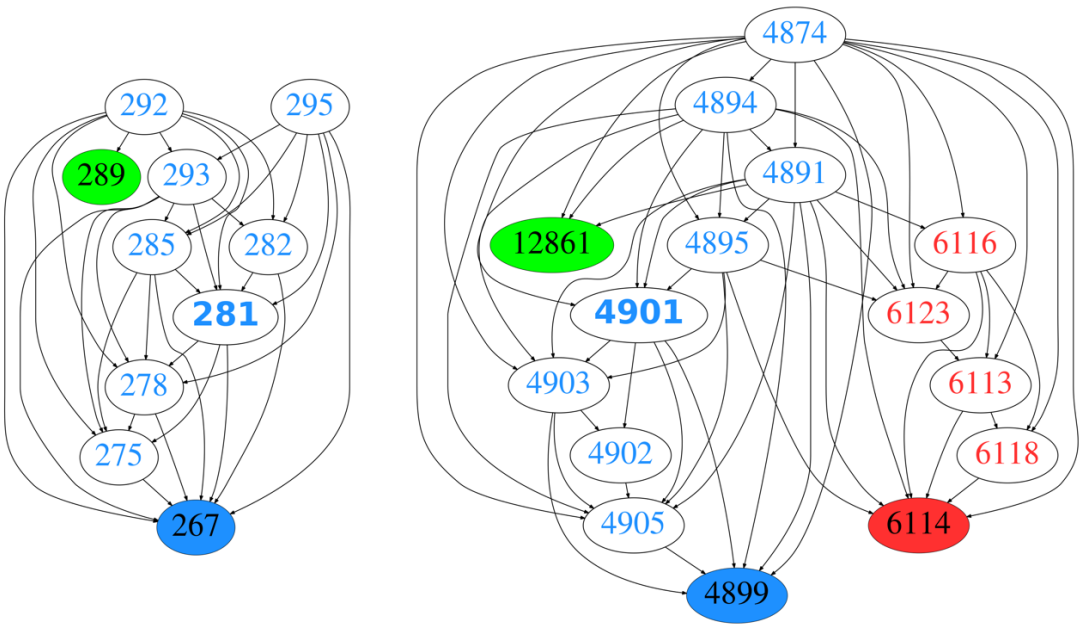

Figure 1.9: Support graph examples revealing the absence and presence of repeats. **(Left)** Support graph $G(Reads)$ for a read in the BLS dataset that does not end in a long repeat (details for this dataset can be found in the "Datasets" paragraph of the Results section of this chapter). Reads in the 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 $G^*(Reads)$ 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 $G^*(Reads)$ is a most-consistent read-path, and it is selected for path extension (unless it ends in a repeat). **(Right)** Support graph $G^*(Reads)$ 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 $G^*(Reads)$ and reveals a cluster consisting of nine blue vertices. The vertex 4901 with large indegree (4) and large outdegree (4) in $G^*(Reads)$ 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 $G^*(Reads)$.

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 1.8, 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.

1.4.3 Correcting Errors in the Draft Genome

Matching Reads Against the Draft Genome.

ABruijn uses BLASR (Chaisson et al. 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 (misaligned segments)]. 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 (or column) of the alignment with the space symbol "-" in the reference sequence as a non-reference position (or column) and to all other positions as a reference position (or 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 1.10).

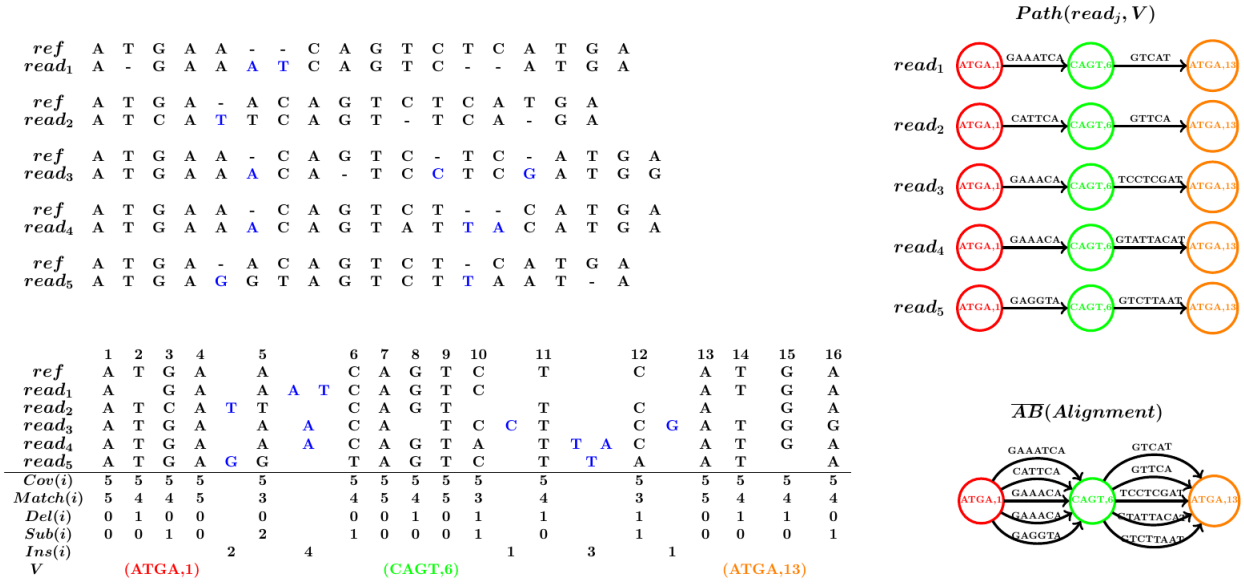

Figure 1.10: Decomposing a multiple alignment into necklaces.
(Top Left) The pairwise alignments between a reference region ref in the draft genome and five reads $Reads = \{read_1, read_2, read_3, read_4, read_5\}$. All inserted symbols in these reads with respect to the region ref are colored in blue. **(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 nonreference 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 made up of segments {GAATCA,GATTCA,GAAACA,GAAACA,GAGGTA} and {GTCAT,GTTCA,TCCTCGAT,GTATTACAT,GTCTTAAT}.

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 1.10, 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. There are 139,585 solid regions in the draft assembly of the ECOLI dataset (for $l = 10$). 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.

Because (α, β)-solid l-mers are very accurate (for appropriate choices of α, β and l), we use them to construct yet another A-Bruijn graph with much simpler bubbles. Because analyzing errors in homonucleotide runs is a difficult problem (Chin et al. 2013), we select landmarks outside of homonucleotide runs.

A 4-mer is called simple if all its consecutive nucleotides are different. For example, CAGT and ATGA are simple 4-mers, and GTTC is not a simple 4-mer. We select simple 4-mers that are at least l positions away from each other within solid regions as landmarks. We introduce multiple landmarks (rather than a single one) 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 (the average length of these segments is only ≈30 nucleotides). Figure 1.11 shows the distribution of the lengths of necklaces constructed by aligning all reads in the ECOLI dataset to the draft genome.

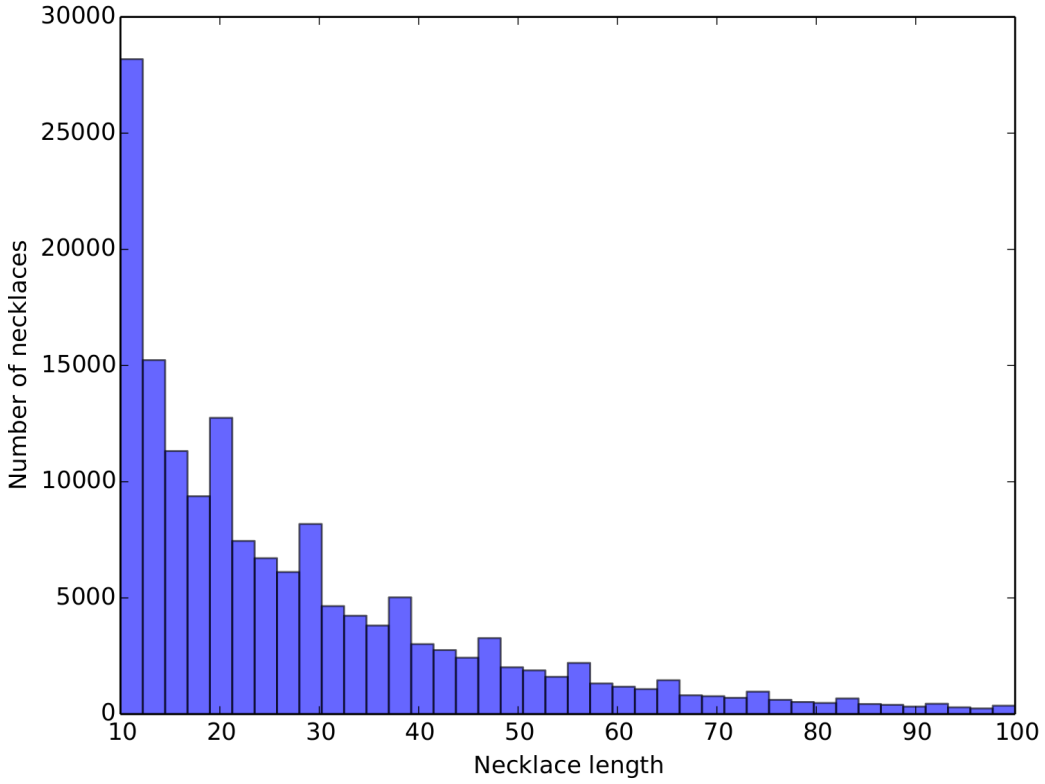

Figure 1.11: A histogram of necklace lengths.
A histogram of the lengths of 159,142 necklaces formed by aligning all reads in the ECOLI dataset to the draft genome and constructing the A-Bruijn graph for this alignment. 3,271 necklaces that are longer than 100 bp are not shown.

To evaluate how errors in the draft genome affect alignments of long error-prone reads, we corrupted the reference E. coli genome by introducing random single-nucleotide errors at randomly chosen positions (10,000 mismatches, deletions, and insertions) and aligned all reads against the corrupted genome. A segment in the corrupted genome is called corrupted if it has been changed by an error and correct otherwise. Figure 1.12 shows the distribution of the local

match and insertion rates (for both corrupted and correct simple 4-mers) and illustrates that 77% of all correct simple 4-mers are (0.8, 0.2)-solid. Remarkably, none of the corrupted simple 4-mers are (0.8, 0.2)-solid.

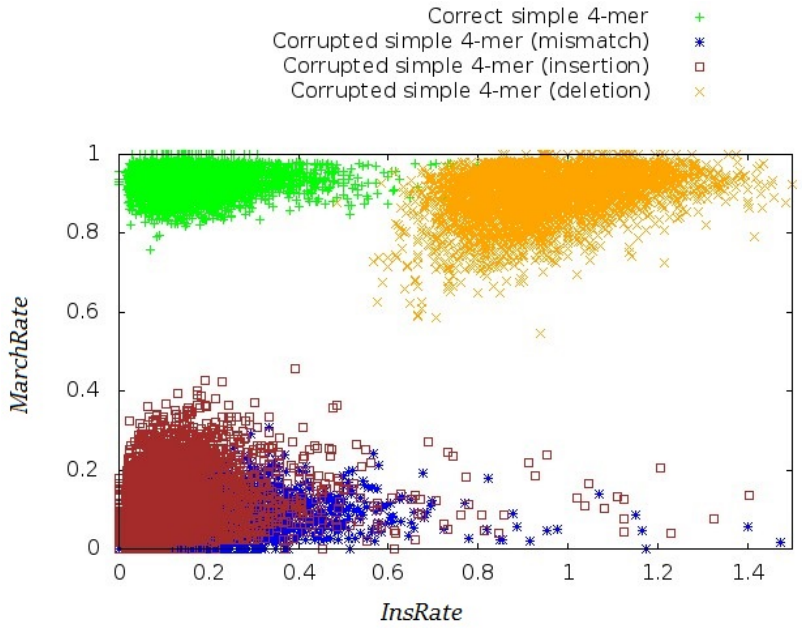

Figure 1.12: Match and insertion rate distribution for a simulated corrupted genome. Distribution of local match and insertion rates as a 2-D plot for correct simple 4-mers (green), corrupted simple 4-mers with mismatches (blue), corrupted simple 4-mers with insertions (red) and corrupted simple 4-mers with deletions (orange).

ABruijn finds all maximal (α, β)-solid l-mers (the default value of l is 10) and treats them as solid regions. It further uses the landmarks (the middle points of simple 4-mers) within the solid regions as the boundaries of necklaces to ensure that single homonucleotide runs in reads do not split into two consecutive necklaces and are not adjacent to the boundaries of necklaces. This condition is important for the subsequent genome polishing step.

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 1.10, 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 1.10, 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).

A Probabilistic Model for Necklace Polishing.

Each necklace contains read-segments $Segments = \{seg_1, seg_2, \ldots, seg_m\}$ and our goal is to find a consensus sequence $Consensus$ maximizing $Pr(Segments|Consensus) = \prod_{i=1}^{m} Pr\ (seg_i|Consensus)$, 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. Table 1.2 shows the values for these rates for three different protocols of Pacific Biosciences reads: P6-C4, P5-C3, and P4-C2. Note that the parameters for P6-C4 and P5-C3 are nearly identical.

Table 1.2: Match, mismatch, insertion, and deletion rates for various Pacific Biosciences protocols.
The match, mismatch, insertion, and deletion rates obtained by aligning datasets from different protocols against the reference genome. The statistical parameters of the P6-C4 protocol were compared with the statistical parameters of the older P5-C3 and P4-C2 protocol (derived from the P5-C3 and P4-C2 Pacific Biosciences datasets in Kim et al. 2014).

	P6-C4	P5-C3	P4-C2
$Match(A)$	0.958	0.962	0.953
$Match(C)$	0.944	0.935	0.946
$Match(G)$	0.950	0.946	0.920
$Match(T)$	0.956	0.958	0.936
$Sub(A \to C)$	0.005	0.004	0.006
$Sub(A \to G)$	0.002	0.002	0.002
$Sub(A \to T)$	0.002	0.002	0.003
$Sub(C \to A)$	0.008	0.012	0.010
$Sub(C \to G)$	0.004	0.006	0.002
$Sub(C \to T)$	0.004	0.004	0.003
$Sub(G \to A)$	0.004	0.004	0.006
$Sub(G \to C)$	0.003	0.003	0.006
$Sub(G \to T)$	0.004	0.004	0.006
$Sub(T \to A)$	0.004	0.004	0.006
$Sub(T \to C)$	0.003	0.002	0.007
$Sub(T \to G)$	0.004	0.003	0.004
$Del(A)$	0.032	0.030	0.036
$Del(C)$	0.041	0.043	0.038
$Del(G)$	0.039	0.043	0.062
$Del(T)$	0.033	0.033	0.047
$Ins(A)$	0.027	0.028	0.024
$Ins(C)$	0.019	0.016	0.031
$Ins(G)$	0.022	0.024	0.016
$Ins(T)$	0.021	0.020	0.016
$NoIns$	0.912	0.912	0.913

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 drops to 0.003% after this step 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. 2015). 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.

Furthermore, for each run LZ ... ZR, where a nucleotide Z in the genome is flanked by the nucleotides L (on the left) and R (on the right) distinct from Z, we limit our analysis to only reads that are well-aligned against LZ ... ZR. A read is well-aligned against LZ ... ZR if the flanking L and R nucleotides both form either a match with the read or is aligned against a nucleotide Z in the read (see Figure 1.13). The counts of all read segments well-aligned to each homonucleotide region are used to calculate the error distributions for all homonucleotide runs.

read genome	CAAT–AG | | | * | | | CAAAAAG	CAAT–AA | | | * | | * CAAAAAG	CAAAAAT | | | | | | * CAAAAAG
	3A1X	4A1X	(not counted)

Figure 1.13: Examples of read well-aligned to homonucleotide regions.
Well-aligned reads (the first two examples) and a poorly aligned read (the last example). The well-aligned reads are represented as 3A1X and 4A1X in the likelihood estimate (X stands for an arbitrary nucleotide).

Table 1.3 presents the frequencies for all read segments covering the homonucleotide runs AAAAAA and AAAAAAA in the ECOLI dataset. Table 1.4 presents the frequencies for all read segments covering the homonucleotide runs AAAA and AAAAA for ECOLI$_{nano}$. Interestingly, when we apply the statistical parameters derived from the older P5-C3 protocol to our P6-C4 ECOLI dataset, the number of ABruijn errors remains small, illustrating that our

probabilistic framework is not subject to over-training. The frequencies in the resulting tables hardly change when one changes the dataset of reads or the reference genome either.

Table 1.3: AAAAAA and AAAAAAA error distributions for ECOLI. The frequencies of segments from reads spanning 6-nucleotide runs AAAAAA **(Left)** and 7-nucleotide runs AAAAAAA **(Right)** in the ECOLI dataset. Only combinations with frequencies exceeding 0.001 are shown. X stands for an arbitrary nucleotide.

AAAAAA	Frequency	AAAAAAA	Frequency
6A	0.439	7A	0.385
5A	0.156	6A	0.156
7A	0.115	8A	0.119
6A1X	0.074	7A1X	0.071
5A1X	0.041	5A	0.049
4A	0.039	6A1X	0.045
7A1X	0.021	8A1X	0.024
8A	0.018	9A	0.024
6A2X	0.015	7A2X	0.016
4A1X	0.010	5A1X	0.014
3A	0.009	4A	0.013
5A2X	0.009	6A2X	0.011
7A2X	0.005	8A2X	0.006
6A3X	0.005	9A1X	0.006
8A1X	0.004	10A	0.006
4A2X	0.003	7A3X	0.005
5A3X	0.003	5A2X	0.004
9A	0.003	3A	0.003
6A4X	0.003	4A1X	0.003
3A1X	0.002	6A3X	0.003
7A3X	0.002	7A4X	0.002
2A	0.002	8A3X	0.002
6A5X	0.001	6A4X	0.002
8A2X	0.001	9A2X	0.001
4A3X	0.001	5A3X	0.001
6A6X	0.001	2A	0.001
9A1X	0.001	6A5X	0.001
		10A1X	0.001
		8A4X	0.001
		7A5X	0.001
		4A2X	0.001

Table 1.4: AAAA and AAAAA error distributions for ECOLI$_{nano}$. The frequencies of segments from Oxford Nanopore reads spanning 4-nucleotide runs AAAA **(Left)** and 5-nucleotide runs AAAAA **(Right)** in the ECOLI$_{nano}$ dataset. Only combinations with frequencies exceeding 0.001 are shown. X stands for an arbitrary nucleotide.

AAAA	Frequency	AAAAA	Frequency
4A	0.359	4A	0.310
3A	0.351	3A	0.256
3A1X	0.063	5A	0.126
2A	0.043	4A1X	0.059
4A1X	0.035	3A1X	0.047
3A2X	0.019	2A	0.028
2A1X	0.017	5A1X	0.025
4A2X	0.016	3A2X	0.023
5A	0.015	4A2X	0.022
5A1X	0.012	5A2X	0.012
2A2X	0.011	2A1X	0.011
3A3X	0.006	4A3X	0.008
4A3X	0.006	3A3X	0.007
5A2X	0.005	2A2X	0.007
1A1X	0.004	6A1X	0.007
1A	0.004	5A3X	0.006
1A2X	0.004	2A3X	0.005
2A3X	0.003	1A	0.004
1A3X	0.003	1A1X	0.004
4A4X	0.003	6A2X	0.003
5A3X	0.002	4A4X	0.003
3A4X	0.002	5A4X	0.003
6A1X	0.001	1A2X	0.002
2A4X	0.001	3A4X	0.002
4A5X	0.001	6A3X	0.002
5A4X	0.001	5A5X	0.001
		1A3X	0.001
		2A4X	0.001
		4A5X	0.001
		1A4X	0.001

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). To decide on the length of a homonucleotide run, we simply select the length of the run that maximizes the likelihood function. For example, using the frequencies from Table 1.3, if $Segments = \{5A, 6A, 6A, 7A, 6A1C\}$, then

$Pr(Segments|6A) = 0.156\times0.4392\times0.115\times0.0740.156\times0.4392\times0.115\times0.074 >$

$Pr(Segments|7A) = 0.049\times0.1562\times0.385\times0.0450.049\times0.1562\times0.385\times0.045$

and we select AAAAAA over AAAAAAA as the necklace consensus.

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). Further improvements on correcting errors was explored by considering the lengths of Open Reading Frames (ORFs).

ORF-Based Error-Correction of Bacterial Genomes.

While the likelihood-based approaches to error-correction (described in the main text) corrects the lion's share of errors in the draft genomes, some errors remain uncorrected, particularly with respect to the errors in estimating the lengths of homonucleotide runs. We thus complement the likelihood-based approaches with an ORF-based error-correction approach that analyzes Open Reading Frames (ORFs).

Note that while the average length of a protein-coding gene in most bacterial genomes exceeds 800 bp (Brocchieri et al. 2005), the average ORF length in a randomly generated string of nucleotides is only 64 bp. Thus, every error that represents an indel within a gene (a frameshift) may introduce a premature stop codon and has the potential to significantly reduce the length of the ORF corresponding to this gene.

If we are deciding between two alternative lengths of a homonucleotide run within a gene (correct and incorrect), the correct choice results in an ORF that corresponds to the gene length while the incorrect choice results in a frameshift that may introduce a premature stop codon.

Such frameshift mutations usually shorten the length of the longest ORF that spans over the homonucleotide run with incorrectly defined length.

Given a position in the genome, we compute its ORF-length as the maximum length of all six ORFs covering this position. If the genome is assembled without errors, then ORF-lengths are large for most positions that belong to genes. Since genes typically cover over 85% of bacterial genomes, most positions in the entire genome have large ORF-lengths. However, if a genome is assembled with errors, the ORF-lengths for positions with indels are typically smaller than the ORF-length of this position in the error-free genome (see Figure 1.14).

Since in some cases, the likelihood values for alternative choices for the length of a homonucleotide run are nearly the same, we develop an additional decision rule that analyzes the ORF-lengths between two alternatives and gives preference to the choice that results in a significantly longer ORF-length.

Given two candidate lengths of a homonucleotide run with a small difference in their homonucleotide likelihood score (smaller than a threshold Δ), we compute the difference between their ORF-lengths and select the candidate with larger ORF-length if the difference between ORF-lengths exceeds a threshold (the default value is 128 bp). If the difference between the ORF-lengths is smaller than the threshold, we retain the length of the run that maximizes the homonucleotide likelihood score described in the main text.

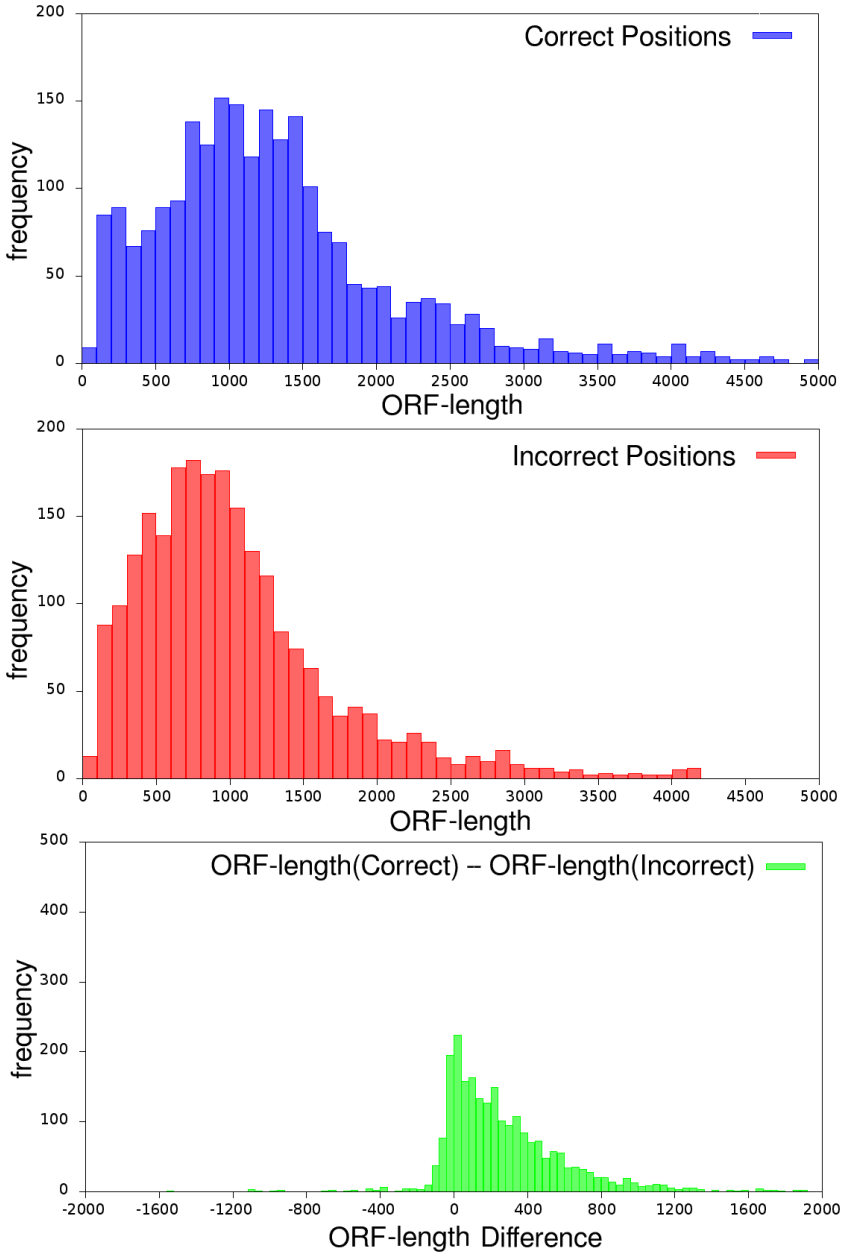

Figure 1.14: ORF-length histograms for correct and incorrect positions.
Distribution of ORF-lengths for correct positions in the error-free E. coli genome (top) and incorrect positions in the error-prone E. coli genome (middle), and the difference between the ORF-lengths of corresponding correct and incorrect positions (bottom). The error-prone E. coli genome was generated by deleting or inserting a single (randomly chosen) nucleotide with probability 0.0005 at each position. The vast majority of indels in the error-prone genome result in a significant reduction of ORF lengths. On average, there is a 276-nucleotide reduction in the ORF-length for the error-prone genome.

1.5 Results

Because CANU (Berlin et al. 2015) improved on PBcR (Koren et al. 2012) with respect to both speed and accuracy, we limited our benchmarking to ABruijn and CANU v1.2 using the following datasets.

Datasets.

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

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

The BLS and PXO datasets were derived from *X. 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 *B. neritina* dataset (referred to as BNE) contains 1,127,494 reads (estimated coverage ≈ 25×) generated using the P6-C4 Pacific Biosciences technology. *B. neritina* is a microscopic marine eukaryote that forms colonies attached to the wetted surfaces and forms symbiotic communities with various bacteria. *B. neritina* is the source of bryostatin, an anticancer and memory-enhancing compound (Trost et al. 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.

Symbiotic bacteria live inside of *B. neritina*, making it impossible to isolate the *B. neritina* DNA from the bacterial DNA when performing genome sequencing. As a 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.

We have also assembled the *S. cerevisiae* W303 genome (referred to as SCE), which contains 232,230 reads with $\approx 117\times$ coverage generated using the P5-C3 Pacific Biosciences technology (Kim et al. 2014).

The Challenge of Benchmarking SMS Assemblies.

High-quality short-read bacterial assemblies typically have error-rates on the order of 10^{-5}, 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).

Assembling 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 1.5 illustrates that, in contrast to ABruijn, CANU generates rather inaccurate assemblies without Quiver, a tool that uses raw machine-level signals saved in HDF5 files for polishing: 637 errors (160 insertions and 477 deletions) and 19 errors (12 insertions and 7 deletions) remain 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 1.5: Total errors remaining for CANU and ABruijn assemblies.
Summary of errors for the CANU and ABruijn assemblies of the ECOLI, BLS, and PXO datasets as well as for the down-sampled ECOLI datasets with coverage varying from 50× to 25×. To offset CANU assembly errors in the case of 30× coverage, we provided the average number of errors for the four replicas with best results (out of five).

Coverage	CANU	ABruijn	CANU+ Quiver	ABruijn+ Quiver
BLS	73	5	51	31
PXO	1,162	21	130	15
ECOLI	637	19	14	15
ECOLI 50×	703	33	20	18
ECOLI 45×	829	45	29	29
ECOLI 40×	1,158	84	45	45
ECOLI 35×	1,541	153	88	84
ECOLI 30×	2,470	291	175	154
ECOLI 25×	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 the four out of five best assemblies for each value of coverage. Table 1.5 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. Table 1.6 illustrates that the lion's share of ABruijn errors occur in the low-coverage regions. When the coverage of bubbles drops to at most 15×, the fraction of bubbles with errors goes up to 1% and then for coverage at most 10× up to 5%.

Table 1.6: Analysis of errors in down-sampled datasets.
The total number of bubbles/necklaces with errors in them increases as the coverage of those bubbles/necklaces decreases. Very few errors occur when the coverage is at above 20×.

coverage	#bubbles	#bubbles with errors	fraction of erroneous bubbles
6-10X	4360	205	0.047
11-15X	103522	1357	0.013
16-20X	498494	1725	0.0035
21-25X	973452	1149	0.0012
26-30X	1219376	675	0.0006
31-35X	1248166	333	0.0003
36-40X	1046376	174	0.0002

Assembling the ECOLI$_{nano}$ Dataset.

Both the Nanocorrect assembler described in Loman et al. (2015) and ABruijn assembled the ECOLI$_{nano}$ dataset into a single circular contig structurally concordant with the *E. coli K12* genome. Nanocorrect and ABruijn runs resulted in assemblies 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 co-assembly with short reads to generate finished genomes (Antipov et al. 2015; Labont 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 machine-level 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 (see Table 1.5) 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.

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 long 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 the collapsed 212-kb repeat (generated by both CANU and ABruijn). The ABruijn+Quiver assembly of the PXO dataset resulted in only 15 errors whereas the CANU+Quiver assembly resulted in 130 errors, including one insertion of 100 nucleotides.

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 1.15 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× 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.

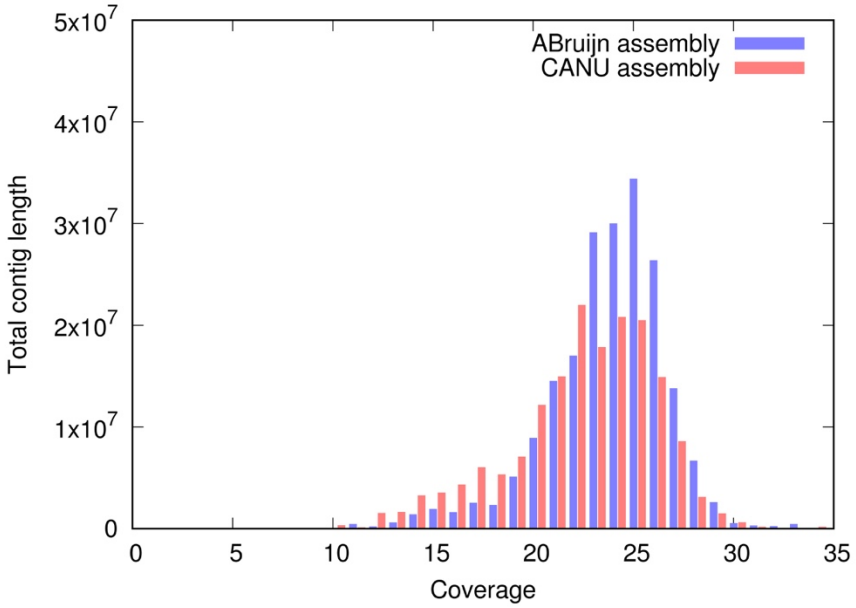

Figure 1.15: A comparison between ABruijn and CANU assemblies for *B. neritina*. This histogram shows how the total length of contigs varies with coverage for CANU (red) and ABruijn (blue) assemblies. ABruijn contig lengths are shown on the left of the corresponding CANU contig length for each coverage value.

We attribute the large difference in the total contig length to fragmentation in CANU assemblies for low-coverage datasets, which we had already observed in our analysis of the down-sampled 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 ultra-conserved 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.

Assembling the *S. cerevisiae* W303 Genome.

Since the *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 estimating 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. This high similarity between the assemblies suggests that many of the differences between these assemblies and chromosome IV represent structural variations rather than assembly errors.

Considering only long contigs (longer than 50 Kb), both the PBcR-MHAP assemblies and ABruijn assemblies of the SCE dataset were largely structurally concordant with the sixteen chromosomes of the *S. cerevisiae* S288C genome (Kim et al. 2014). Although QUAST with default parameters reported 77 and 72 misassemblies for the 20 long contigs in the PBcR-MHAP

assembly and the 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, so 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 the *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.

Running Time and Memory Footprint.

For the *Xanthomonas* genomes, which have complex repeat structure and high coverage, the assembly time and memory footprint increased compared to the ECOLI dataset: 48 minutes for the assembly step, 125 minutes for the polishing step, and 15 Gb of memory for the PXO dataset, and 26 minutes for the assembly step, 90 minutes for the polishing step, and 21 Gb of 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 48 minutes (with a memory footprint of 2 Gb) for the ECOLI$_{nano}$ 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 less than 2 minutes).

In contrast, the running time for the SCE dataset was dominated by the assembly step (8 hours and 44 minutes for the assembly step and 2 hours and 30 minutes for the polishing step). The increase in the running time of the assembly step is explained by the presence of many long and highly conserved Ty1 - Ty 5 repeats and long segmental duplications.

For the BNE metagenome, the assembly step took 9 hours and 10 minutes, the polishing step took 19 hours and 21 minutes, and the memory footprint was 278 Gb (64 cores, AMD Opteron 6376 2.30 GHz, 512 Gb of RAM).

1.6 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. Since 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.

1.7 Additional Information

Author contributions.

Yu Lin (Y.L.), Jeffrey Yuan (J.Y.), Mikhail Kolmogorov (M.K.), and Pavel A. Pevzner (P.A.P.) designed research; Y.L., J.Y., M.K., Max W. Shen (M.W.S.), and P.A.P. performed research; Y.L., M.K., and Mark Chaisson (M.C.) analyzed data; and Y.L., M.K., and P.A.P. wrote the paper. Y.L., J.Y., and M.K. contributed equally to this work.

Conflict of Interests.

The authors declare no conflict of interest.

Code Availability.

The ABruijn assembler is freely available from *https://sites.google.com/site/abruijngraph*.

1.8 Acknowledgements

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.

Chapter 1, in full, is a reformatted reprint of "Assembly of long error-prone reads using de Bruijn graphs" as it appears in *Proceedings of the National Academy of Sciences USA* 2016 by Yu Lin, Jeffrey Yuan, Mikhail Kolmogorov, Max W. Shen, Mark Chaisson, and Pavel A. Pevzner, with some minor revisions and edits for improved readability. The dissertation author was a primary author of this material.

1.9 References

Antipov, D., Korobeynikov, A. & Pevzner, P.A. hybridSPAdes: An algorithm for hybrid assembly of short and long reads. *Bioinformatics*. 2015; 32 (7): 1009-1115.

Ashton, P.M., Nair, S., Dallman, T., Rubino, S., Rabsch, W., Mwaigwisya, S., Wain, J. & O'Grady, J. Minion nanopore sequencing identifies the position and structure of a bacterial antibiotic resistance island. *Nature Biotechnology*. 2015; 33 (3): 296-300.

Bandeira, N., Clauser, K.R. & Pevzner, P.A. Shotgun protein sequencing: Assembly of peptide tandem mass spectra from mixtures of modified proteins. *Molecular & Cellular Proteomics*. 2007; 6 (7): 1123-1134.

Bandeira, N., Pham, V., Pevzner, P., Arnott, D. & Lill, J.R. Automated de novo protein sequencing of monoclonal antibodies. *Nature Biotechnol*ogy. 2008; 26 (12): 1336-1338.

Bankevich. A., Nurk, S., Antipov, D., Gurevich, A.A., Dvorkin, M., Kulikov, A.S., Lesin, V.M., Nikolenko, S.I., Pham, S., Prjibelski, A.D., Pyshkin, A.V., Sirotkin, A.V., Vyahhi, N., Tesler, G., Alekseyev, M.A. & Pevzner, P.A. SPAdes: A new genome assembly algorithm and its applications to single-cell sequencing. *Journal of Computational Biology*. 2012; 19 (5): 455-477.

Berlin, K., Koren, S., Chin, C.S., Drake, J.P., Landolin, J.M. & Phillippy, A.M. Assembling large genomes with single-molecule sequencing and locality-sensitive hashing. *Nature Biotechnology*. 2015; 33 (6): 623-630.

Bogdanove, A.J., Koebnik, R., Lu, H., Furutani, A., Angiuoli, S.V., Patil, P.B., Van Sluys, M.A., Ryan, R.P., Meyer, D.F., Han, S.W., Aparna, G., Rajaram, M., Delcher, A.L., Phillippy, A.M., Puiu, D., Schatz, M.C., Shumway, M., Sommer, D.D., Trapnell, C., Benahmed, F., Dimitrov, G., Madupu, R., Radune, D., Sullivan, S., Jha, G., Ishihara, H., Lee, S.W., Pandey, A., Sharma, V., Sriariyanun, M., Szurek, B., Vera-Cruz, C.M., Dorman, K.S., Ronald, P.C., Verdier, V., Dow, J.M., Sonti, R.V., Tsuge, S., Brendel, V.P., Rabinowicz, P.D., Leach, J.E., White, F.F. & Salzberg, S.L. Two new complete genome sequences offer insight into host and tissue specificity of plant pathogenic Xanthomonas spp. *Journal of Bacteriology*. 2011; 193 (19): 5450–5464.

Boisvert, S., Raymond, F., Godzaridis, E., Laviolette, F. & Corbeil, J. Ray meta: Scalable de novo metagenome assembly and profiling. *Genome Biology*. 2012; 13 (12): R122.

Bonissone, S.R. & Pevzner, P.A. Immunoglobulin classification using the colored antibody graph. *Journal of Computational Biology*. 2016; 23 (6): 483-494.

Booher, N.J., Carpenter, S.C., Sebra, R.P., Wang, L., Salzberg, S.L., Leach, J.E. & Bogdanove, A.J. Single molecule real-time sequencing of Xanthomonas oryzae genomes reveals a dynamic structure and complex TAL (transcription activator-like) effector gene relationships. *Microbial Genomics*. 2015; 1 (4): 1-22.

Brocchieri, L. & Karlin, S. Protein length in eukaryotic and prokaryotic proteomes. *Nucleic Acids Research*. 2005; 33 (10): 3390-400.

Butler, J., MacCallum, I., Kleber, M., Shlyakhter, I.A., Belmonte, M.K., Lander, E.S., Nusbaum, C. & Jaffe, D.B. ALLPATHS: De novo assembly of whole-genome shotgun microreads. *Genome Res*earch. 2008; 18 (5): 810-820.

Chaisson, M.J., Huddleston, J., Dennis, M.Y., Sudmant, P.H., Malig, M., Hormozdiari, F., Antonacci, F., Surti, U., Sandstrom, R., Boitano, M., Landolin, J.M., Stamatoyannopoulos, J.A., Hunkapiller, M.W., Korlach, J. & Eichler, E.E. Resolving the complexity of the human genome using single-molecule sequencing. *Nature*. 2015; 517 (7536): 608-611.

Chaisson, M.J. & Tesler, G. Mapping single molecule sequencing reads using basic local alignment with successive refinement (BLASR): Application and theory. *BMC Bioinformatics*. 2012; 13: 238.

Chin, C.S., Alexander, D.H., Marks, P., Klammer, A.A., Drake, J., Heiner, C., Clum, A., Copeland, A., Huddleston, J., Eichler, E.E., Turner, S.W. & Korlach, J. Nonhybrid, finished microbial genome assemblies from long-read SMRT sequencing data. *Nature Methods*. 2013; 10 (6): 563-569.

Chin, C.S., Peluso, P., Sedlazeck, F.J., Nattestad, M., Concepcion, G.T., Clum, A., Dunn, C., O'Malley, R., Figueroa-Balderas, R, Morales-Cruz, A., Cramer, G.R., Delledonne, M., Luo, C., Ecker, J.R., Cantu, D., Rank, D.R., & Schatz, M.C. Phased diploid genome assembly with single molecule real-time sequencing. *bioRxiv*. 2016; doi: 056887.

Compeau, P.E.C. & Pevzner, P.A. *Bioinformatics Algorithms: An Active-Learning Approach*. Victoria, BC, Canada: Active Learning Publishers; 2014.

Doyle, E.L., Stoddard, B.L., Voytas, D.F., & Bogdanove, A.J. TAL effectors: Highly adaptable phytobacterial virulence factors and readily engineered DNA-targeting proteins. *Trends in Cell Biology*. 2013; 23 (8): 390-398.

Goodwin, S., Gurtowski, J., Ethe-Sayers, S., Deshpande, P., Schatz, M.C. & McCombie, W.R. Oxford nanopore sequencing and de novo assembly of a eukaryotic genome. *Genome Research*. 2015; 25 (11): 1758-1756.

Gurevich, A., Savaliev, V., Vyahhi, N. & Tesler, G. QUAST: Quality assessment tool for genome assemblies. *Bioinformatics.* 2013; 29 (8): 1072-1075.

Huddleston, J., Ranade, S., Malig, M., Antonacci, F., Chaisson, M., Hon, L., Sudmant, P.H., Graves, T.A., Alkan, C., Dennis, M.Y., Wilson, R.K., Turner, S.W., Korlach, J. & Eichler, E.E. Reconstructing complex regions of genomes using long-read sequencing technology. *Genome Research*. 2014; 24 (4): 688-696.

Idury, R.M. & Waterman, M.S. A new algorithm for DNA sequence assembly. *Journal of Computational Biology*. 1995; 2 (2): 291-306.

Iqbal, Z., Caccamo, M., Turner, I., Flicek, P. & McVean, G. De novo assembly and genotyping of variants using colored de Bruijn graphs. *Nature Genetics*. 2012; 44 (2): 226-232.

Kececioglu, J.D. & Myers, E.W. Combinatorial algorithms for DNA sequence assembly. *Algorithmica*. 1995; 13: 7-51.

Kim, K.E., Peluso, P., Babayan, P., Yeadon, P.J., Yu, C., Fisher, W.W., Chin, C.S., Rapicavoli, N.A., Rank, D.R., Li, J., Catcheside, D.E., Celniker, S.E., Phillippy, A.M., Bergman, C.M. & Landolin, J.M. Long-read, whole-genome shotgun sequence data for five model organisms. *Scientific Data*. 2014; 1: 140045.

Koren, S., Harhay, G.P., Smith, T.P., Bono, J.L., Harhay, D.M., Mcvey, S.D., Radune, D., Bergman, N.H. & Phillippy, A.M. Reducing assembly complexity of microbial genomes with single-molecule sequencing. *Genome Biology*. 2013; 14 (9) 101.

Koren, S. & Phillippy, A.M. One chromosome, one contig: Complete microbial genomes from long-read sequencing and assembly. *Current Opinion in Microbiology*. 2015; 23: 110-120.

Koren, S., Schatz, M.C., Walenz, B.P., Martin, J., Howard, J.T., Ganapathy, G., Wang, Z., Rasko, D.A., McCombie, W.R., Jarvis, E.D. & Phillippy, A.M. Hybrid error correction and de novo assembly of single-molecule sequencing reads. *Nature Biotechnology*. 2012; 30 (7): 693–700.

Labonté, J.M., Swan, B.K., Poulos, B., Luo, H., Koren, S., Hallam, S.J., Sullivan, M.B., Woyke, T., Wommack, K.E. & Stepanauskas, R. Single-cell genomics-based analysis of virus-host interactions in marine surface bacterioplankton. *ISME Journal*. 2015; 9 (11): 2386-2399.

Lam, K.K., LaButti, K., Khalak, A. & Tse, D. FinisherSC: A repeat-aware tool for upgrading de-novo assembly using long reads. *Bioinformatics*. 2015; 31 (19): 3207-3209.

Li, Z., Chen, Y., Mu, D., Yuan, J., Shi, Y., Zhang, H., Gan, J., Li, N., Hu, X., Liu, B., Yang, B. & Fan, W. Comparison of the two major classes of assembly algorithms: Overlap–layout–consensus and de-Bruijn-graph. *Briefings in Functional Genomics*. 2012; 11 (1): 25-37.

Lin, Y., Nurk, S., Pevzner, P.A. What is the difference between the breakpoint graph and the de Bruijn graph? *BMC Genomics*. 2014; 15: S6.

Lin, Y. & Pevzner, P.A. Manifold de Bruijn graphs. *Algorithm Bioinformatics*. 2014; 8701: 296-310.

Loman, N.J., Quick, J. & Simpson, J.T. A complete bacterial genome assembled de novo using only nanopore sequencing data. *Nature Methods*. 2015; 12 (8): 733-735.

Lopanik, N.B., Shields, J.A., Buchholz, T.J., Rath, C.M., Hothersall, J., Haygood, M.G., Håkansson, K., Thomas, C.M. & Sherman, D.H. In vivo and in vitro trans-acylation by BryP, the putative bryostatin pathway acyltransferase derived from an uncultured marine symbiont. *Chemistry & Biology*. 2008; 15 (11): 1175-1186.

Medema, M.H., Blin, K., Cimermancic, P., de Jager, V., Zakrzewski, P., Fischbach, M.A., Weber, T., Takano, E. & Breitling, R. antiSMASH: Rapid identification, annotation and analysis of secondary metabolite biosynthesis gene clusters. *Nucleic Acids Research*. 39: w339.

Myers, E.W. The fragment assembly string graph. *Bioinformatics*. 2005; 21: 79-85.

Myers, E.W. Efficient local alignment discovery amongst noisy long reads. *Algorithms in Bioinformatics*. Lecture Notes in Computer Science. Eds Brown D, Morgenstern B. 2014; 8701: 52–67.

Myers, E.W., Sutton, G.G., Delcher, A.L., Dew, I.M., Fasulo, D.P., Flanigan, M.J., Kravitz, S.A., Mobarry, C.M., Reinert, K.H., Remington, K.A., Anson, E.L., Bolanos, R.A., Chou, H.H., Jordan, C.M., Halpern, A.L., Lonardi, S., Beasley, E.M., Brandon, R.C., Chen, L., Dunn, P.J., Lai, Z., Liang, Y., Nusskern, D.R., Zhan, M., Zhang, Q., Zheng, X., Rubin, G.M., Adams, M.D., Venter, J.C. A whole-genome assembly of Drosophila. *Science*. 2000; 287 (5461): 2196-2204.

Nurk, S., Bankevich, A., Antipov, D., Gurevich, A.A., Korobeynikov, A., Lapidus, A., Prjibelski, A.D., Pyshkin, A., Sirotkin, A., Sirotkin, Y., Stepanauskas, R., Clingenpeel, S.R., Woyke, T., McLean, J.S., Lasken, R., Tesler, G., Alekseyev, M.A. & Pevzner, P.A. Assembling single-cell genomes and mini-metagenomes from chimeric MDA products. *Journal of Computational Biology*. 2013; 20 (10): 714-37.

Parra, G., Bradnam, K. & Korf, I. CEGMA: A pipeline to accurately annotate core genes in eukaryotic genomes. *Bioinformatics*. 2007; 23 (9): 1061-1067.

Pevzner, P.A. 1-Tuple DNA sequencing: computer analysis. *Journal of Biomolecular Structure & Dynamics*. 1989; 7 (1): 63-73.

Pevzner, P.A., Tang, H. & Tesler, G. De novo repeat classification and fragment assembly. *Genome Research*. 2004; 14 (9): 1786-1796.

Pevzner, P.A., Tang, H., Waterman, M.S. An Eulerian path approach to DNA fragment assembly. *Proceedings of the National Academy of Sciences of the USA*. 2001; 98 (17): 9748-9753.

Pham, S.K. & Pevzner, P.A. DRIMM-Synteny: Decomposing genomes into evolutionary conserved segments. *Bioinformatics*. 2010; 26 (20): 2509-2516.

Prjibelski, A.D., Vasilinetc, I., Bankevich, A., Gurevich, A., Krivosheeva, T., Nurk, S., Pham, S., Korobeynikov, A., Lapidus, A. & Pevzner, P.A. ExSPAnder: A universal repeat resolver for DNA fragment assembly. *Bioinformatics*. 2014; 30 (12): 293–301.

Risse, J., Thomson, M., Patrick, S., Blakely, G., Koutsovoulos, G., Blaxter, M., & Watson, M. A single chromosome assembly of Bacteroides fragilis strain BE1 from Illumina and MinION nanopore sequencing data. *Gigascience*. 2015; 4: 60.

Ronen, R., Boucher, C., Chitsaz, H., Pevzner, P. SEQuel: Improving the accuracy of genome assemblies. *Bioinformatics*. 2012; 28 (12): 188-196.

Salzberg, S.L., Sommer, D.D., Schatz, M.C., Phillippy, A.M., Rabinowicz, P.D., Tsuge, S., Furutani, A., Ochiai, H., Delcher, A.L., Kelley, D., Madupu, R., Puiu, D., Radune, D., Shumway, M., Trapnell, C., Aparna, G., Jha, G., Pandey, A., Patil, P.B., Ishihara, H., Meyer, D.F., Szurek, B., Verdier, V., Koebnik, R., Dow, J.M., Ryan, R.P., Hirata, H., Tsuyumu, S., Won Lee, S., Seo, Y.S., Sriariyanum, M., Ronald, P.C., Sonti, R.V., Van Sluys, M.A., Leach, J.E., White, F.F. & Bogdanove, A.J. Genome sequence and rapid evolution of the rice pathogen Xanthomonas oryzae PXO99A. *BMC Genomics*. 2008; 9: 204.

Schornack, S., Moscou, M.J., Ward, E.R., Horvath, D.M. Engineering plant disease resistance based on TAL effectors. *Annual Review of Phytopathology*. 2013; 51: 383-406.

Simpson, J.T., Wong, K., Jackman, S.D., Schein, J.E., Jones, S.J. & Birol, I. ABySS: A parallel assembler for short read sequence data. *Genome Research*. 2009; 19 (6): 1117-1123.

Trost, B.M. & Dong G. Total synthesis of bryostatin 16 using atom-economical and chemoselective approaches. *Nature*. 2008; 456 (7221): 485-488.

Ummat, A. & Bashir A. Resolving complex tandem repeats with long reads. *Bioinformatics*. 2014; 30 (24): 3491-3498.

Vasilinetc, I., Prjibelski, A.D., Gurevich, A., Korobeynikov, A. & Pevzner, P.A. Assembling short reads from jumping libraries with large insert sizes. *Bioinformatics*. 2015; 31 (20): 3261-3268.

Williams, M.M., Sen, K., Weigand, M.R., Skoff, T.H., Cunningham, V.A., Halse, T.A. & Tondella, M.L.; CDC Pertussis Working Group. Bordetella pertussis strain lacking pertactin and pertussis toxin. *Emerging Infectious Diseases*. 2016; 22 (2): 319-322.

Zerbino, D.R. & Birney, E. Velvet: Algorithms for de novo short read assembly using de Bruijn graphs. *Genome Research*. 2008; 18 (5): 821-829.

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 from 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 to existing assemblers.

2.2 Introduction

Genome assembly is the problem 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 or 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 Hi-C (Ghurye, et al. 2017) and optical mapping data (Weissensteiner et al. 2017) to improve the contiguity of assemblies.

The *de Bruijn (DB) 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), that capitalize on a similar DB graph-based approach, have also been developed. Most short read assemblers construct the DB graph based on all k-mers in reads and further transform it into a simpler *DB 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* that visits each edge in the assembly graph. However, in the case of SMS reads, the

key assumption of the DB 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 DB graph and how to transform it into an assembly graph, remain largely unaddressed in long read assemblers.

Here, we describe the Flye algorithm for accurately assembling long reads. 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 (that are bridged by some reads in the repeat graph). Afterwards, it uses the repeat graph to resolve unbridged repeats (that 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 Results

Figure 2.1 outlines the various steps of the Flye assembler (see the Methods section for further details).

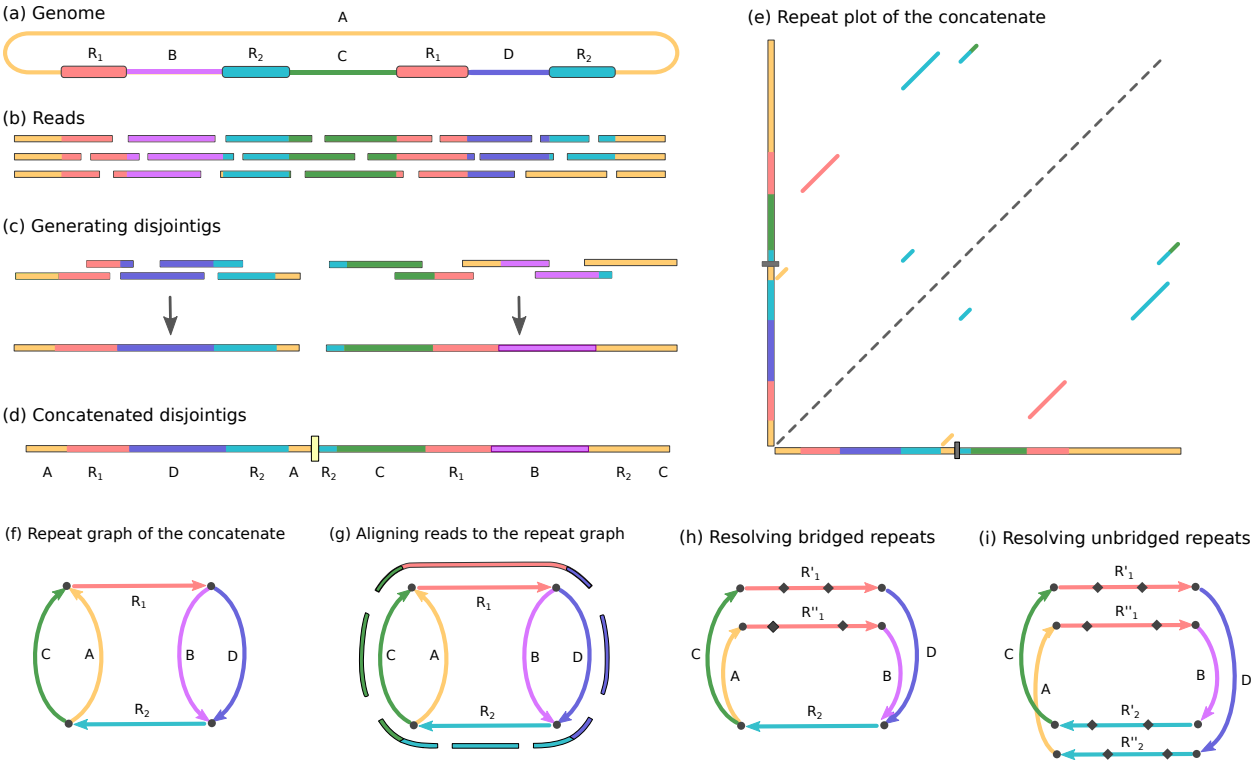

Figure 2.1: An outline of the Flye assembler workflow.
(a) A "genome" with two 99% identical copies of a repeat R_1 and two 99% identical copies of a repeat R_2. Segments A, B, C, and D represent non-repetitive regions. **(b)** A set of reads sampled from the genome. **(c)** Two (misassembled) disjointigs AR_1DR_2A and $R_2CR_1BR_2C$ derived from reads. **(d)** Concatenate of disjointigs. **(e)** Repeat plot of the concatenate. **(f)** Repeat graph constructed by "gluing" vertices in the concatenate according to the repeat-plot. For each 2-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.

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 (Bao et al. 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. The Methods section explains why the resulting graph provides the correct representation of the assembled genome (as if it had been constructed from a complete genome) and describes additional algorithmic details.

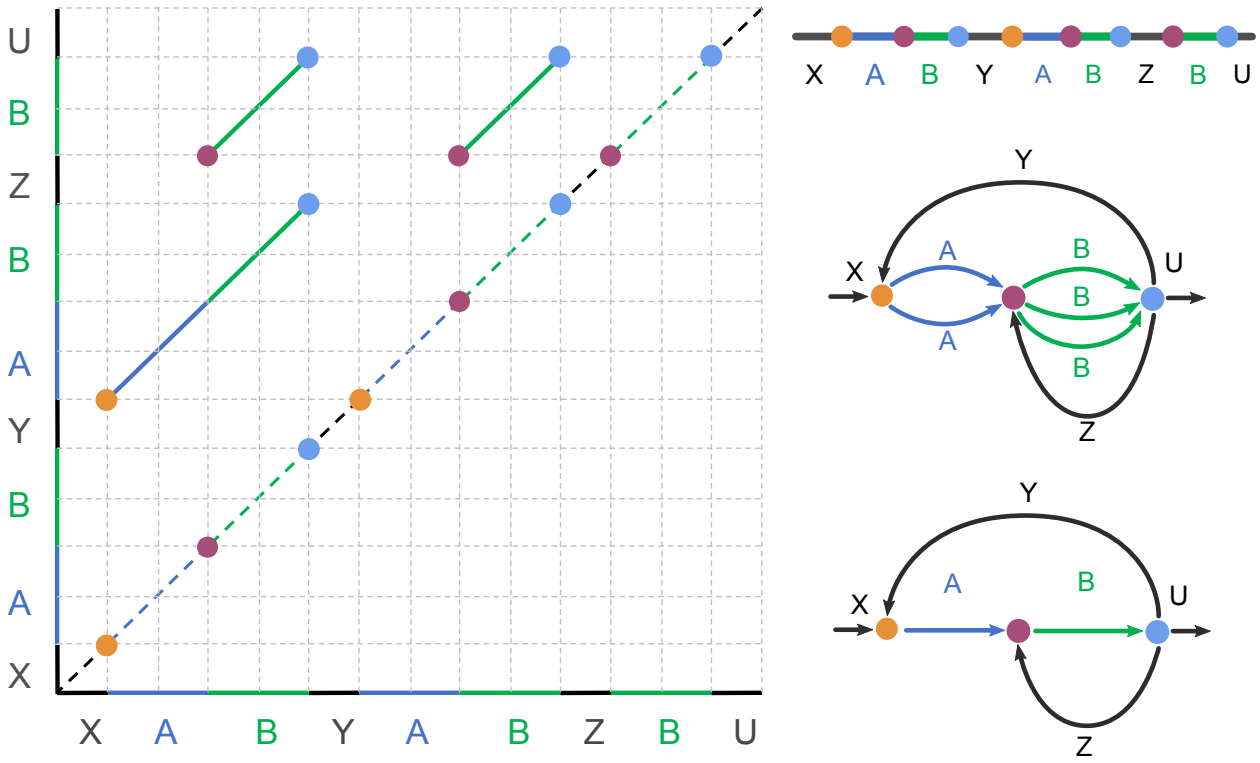

Figure 2.2: Constructing the approximate repeat graph from local self-alignments. **(Left)** Alignment-paths for all local self-alignments within a genome X**AB**Y**AB**Z**B**U formed by segments X, **A**, **B**, Y, Z, and U. Three instances of a mosaic repeat (**AB, AB,** and **B**) are represented 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 rightmost 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. **(Top Right)** Projections of the clustered endpoints on the main diagonal define eight vertices (breakpoints) that will be used for constructing the approximate repeat graph. **(Middle Right)** Breakpoints that belong to the same clusters are glued together. **(Bottom Right)** Gluing parallel edges in the resulting graph produces the approximate repeat graph.

Resolving Unbridged Repeats with Flye.

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; Tischler et al. 2017). The repeat graph constructed by Flye offers a new approach for resolving unbridged repeats based on analyzing the topology of the repeat graph.

Figure 2.3 shows an unbridged repeat with a consensus sequence REP as an edge in the assembly graph. It would be impossible to resolve this repeat (i.e., 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 Figure 2.3. The Methods section describes how Flye resolves unbridged repeats by (i) identifying variations between repeat copies, (ii) matching each read with a specific repeat copy using these variations, and (iii) using these reads to derive a distinct consensus sequence for each repeat copy.

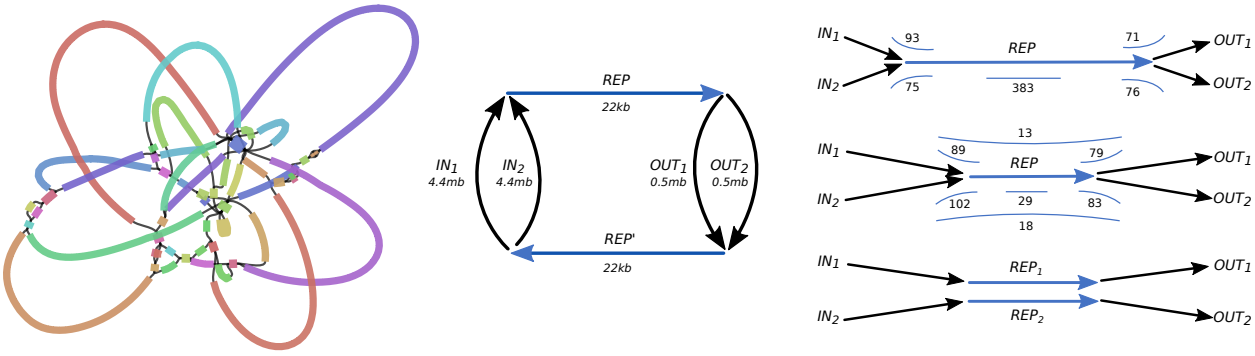

Figure 2.3: Resolving unbridged repeats.
(Left) An assembly graph of SMS reads from the *E. coli* strain EC9964 genome visualized with Bandage (Wick et al. 2015). **(Middle)** The untangled assembly graph (after resolving bridged repeats in the graph on the left) contains a single unbridged repeat REP (and its complement REP') of length 22 kb. The incoming edges into the initial vertex of edge REP are denoted IN_1 and IN_2; the outgoing edges from the terminal vertex are denoted OUT_1 and OUT_2. Two complementary strands are fused together into 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 $IN_1 \rightarrow REP \rightarrow OUT_2 \rightarrow REP'$. **(Top Right)** 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. **(Middle Right)** 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 an 8.6 kb long prefix of REP with divergence 9.8%; we also construct 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). **(Bottom Right)** The resulting resolved instances of the repeat with consensus sequences REP_1 and REP_2 and divergence 6.9%.

Benchmarking Flye.

We benchmarked Flye against various SMS assemblers using six datasets. We used QUAST to evaluate all assemblers (Mikheenko et al. 2018). 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).

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. 2013) 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 (Figure 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 Figure 2.1i. As the result, it assembled the entire genome into a single circular contig.

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 these bacterial genomes (Kamath et al. 2017).

We ignored small connected components in the bacterial assembly graphs (representing plasmids that do not share repeats with chromosomes) and classified an assembly as (i) *complete* if the assembly graph consists of a single loop-edge representing a circular chromosome, (ii) *semi-complete* if the assembly graph contains multiple edges but there exists a single Chinese postman tour in this graph (Edmonds et al. 1973), and (iii) *tangled* if the assembly graph is neither complete nor semi-complete.

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 semi-complete 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 semi-complete 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 semi-complete assembly graph results in a complete genome reconstruction may lead to errors.

Before resolving unbridged repeats, Flye assembled genomes from the BACTERIA dataset into four complete, one semi-complete, and 16 tangled assembly graphs. After resolving unbridged repeats, Flye assemblies resulted in eight complete, five semi-complete, and eight tangled assembly graphs with the number of edges varying from 3 to 25. Figure 2.4 shows

examples of assembly graphs generated by Flye and HINGE, and Table 2.1 illustrates that Flye and HINGE generated very similar assemblies.

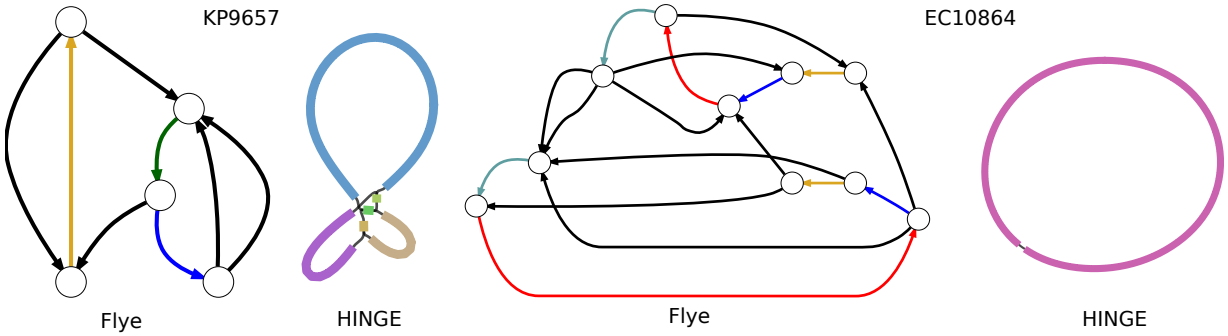

Figure 2.4: A comparison of Flye and HINGE assembly graphs on bacterial genomes from the BACTERIA dataset.
(Left) Flye and Hinge assembly graphs of the KP9657 dataset. There is a single unique edge entering into (and exiting) the unresolved "yellow" repeat and connecting it to the rest of the graph. 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.

Table 2.1: A comparison of the Flye and HINGE assemblies of the bacterial genomes in the BACTERIA dataset.
HINGE results were reproduced from Kamath et al. 2017. "Tangled*" means that the assembly remained tangled and lacked circularization. "n/a" indicates that the assembly graph is not complete and has no unbridged repeats of multiplicity two. "Complete," "Semi-complete" and "Tangled" are defined in the main text.

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

Benchmarking with the METAGENOME Dataset.

The dataset consists of PacBio reads from a synthetic community of 20 bacteria. Since 3 out of 20 bacterial genomes in the metagenomic sample had coverage below 1× (*M. smithii, C. albicans* and *S. 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 the METAGENOME dataset with an NA50 = 1,064 kb (84 misassemblies) and an NA50 = 969 kb (99 misassemblies), respectively. Table 2.2 presents information about the Flye and Canu assemblies of the METAGENOME dataset.

Table 2.2: Information about the Flye and Canu assemblies for 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. "Length" refers to the length of the genome, "Cov" refers to coverage, "#Mis" refers to the number of misassemblies, and "% Assembled" refers to the percent of the genome that was assembled.

			Flye			Canu		
Bacteria	Length (kb)	Cov	% Assembled	NGA50 (kb)	#Mis	% Assembled	NGA50 (kb)	#Mis
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

Flye performed better than Canu for five genomes and Canu performed better than Flye for four genomes. In particular, Flye produced a better assembly of *R. sphaeroides*, which has the lowest coverage (24×) among the 17 analyzed genomes (NGA50 = 2 Mb for Flye as compared to 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 better handling datasets with uneven coverage.

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 assembly on each of the 17 bacteria separately (separate assemblies) by first binning the initial reads using alignments to the references and then running Flye and Canu on the resulting set of reads (see Table 2.3). Six out of the 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 misassembly breakpoints had a matching breakpoint in the Canu contigs, whereas 70% of Canu breakpoints had a matching one in the 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 misassemblies reported by metaQUAST were mainly caused by differences between the genomes in the METAGENOME sample and the reference genomes rather than assembly artifacts.

Table 2.3: 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. "Length" refers to the length of the genome, "Cov" refers to coverage, "#Mis" refers to the number of misassemblies, and "% Assembled" refers to the percent of the genome that was assembled. Bolded numbers indicate significant improvement over the other assembler.

			Flye			Canu		
Bacteria	Length (kb)	Cov	% Assembled	NGA50 (kb)	#Mis	% Assembled	NGA50 (kb)	#Mis
*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

Benchmarking with the YEAST Dataset.

The YEAST dataset contains PacBio and Oxford Nanopore Technology (ONT) reads from the *S. 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 down-sampled the PacBio reads from the original 120× coverage to 30× in the YEAST-PB dataset to have their coverage distribution be similar to the ONT data. Assembling this dataset with the original 120× coverage results in better assemblies; e.g., the NGA50 increased from 560 kb to 732 kb for the Flye assembly (Flye fully assembled 14 out of 16 yeast chromosomes). Table 2.4 illustrates that all of the assemblers tested except HINGE produced YEAST-PB 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 with 5 errors (vs 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%).

Table 2.4: Assembly statistics for the YEAST, WORM, HUMAN and HUMAN+ datasets generated using QUAST 5.0.

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, the corrected assembly is formed after breaking each erroneously assembled contig at its breakpoints resulting in shorter contigs (Mikheenko et al. 2018). 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. "Len" is the total length assembled, "#Mis" is the number of misassemblies, and "Reference coverage" is how much of the reference is found in the assembly.

Dataset	Assembler	Len (Mb)	#Contigs	NG50 (kb)	Reference Coverage	Reference % Identity	#Mis	NGA50 (kb)
YEAST PB	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%	99.81%	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

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 (Canu resulted in 18 errors). Figure 2.5 shows the assembly graph generated by Flye.

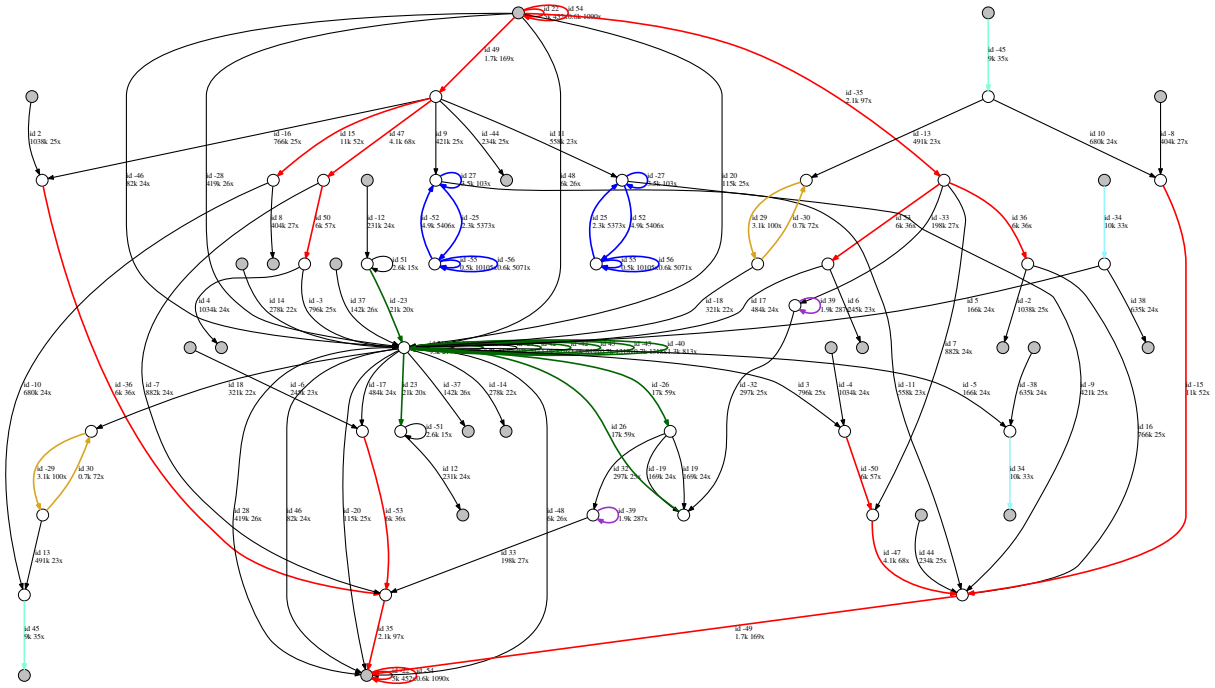

Figure 2.5: The assembly graph of the YEAST-ONT dataset.
Edges that were classified as repetitive by Flye are shown in color, while unique edges are black. Flye assembled the YEAST-ONT dataset into a graph with 21 unique and 34 repeat edges and generated 21 contigs as unambiguous paths in the assembly graph. A path $v_1 \dots v_i, v_{i+1} \dots v_n$ 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 repetitive edges which were not covered by any unique contigs. This visualization was generated using the graphviz tool (Ellson et al. 2003).

Analyzing the WORM Dataset.

The WORM dataset contains PacBio reads from the *C. elegans* genome of length 100 Mb at 40× coverage. Flye and Canu produced the most contiguous assemblies (NGA50 = 1,893 kb and 1,974 kb, respectively). However, Canu showed an increased number of misassemblies (190), compared to 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 the "Information about running time and memory usage" section for more details). With an increase in genome size, Flye achieves close to an order of magnitude speed-up as compared to Canu: e.g., 140 vs. 1100 hours to assemble the *D. 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. Figure 2.6 shows the *C. elegans* assembly graph generated by Flye.

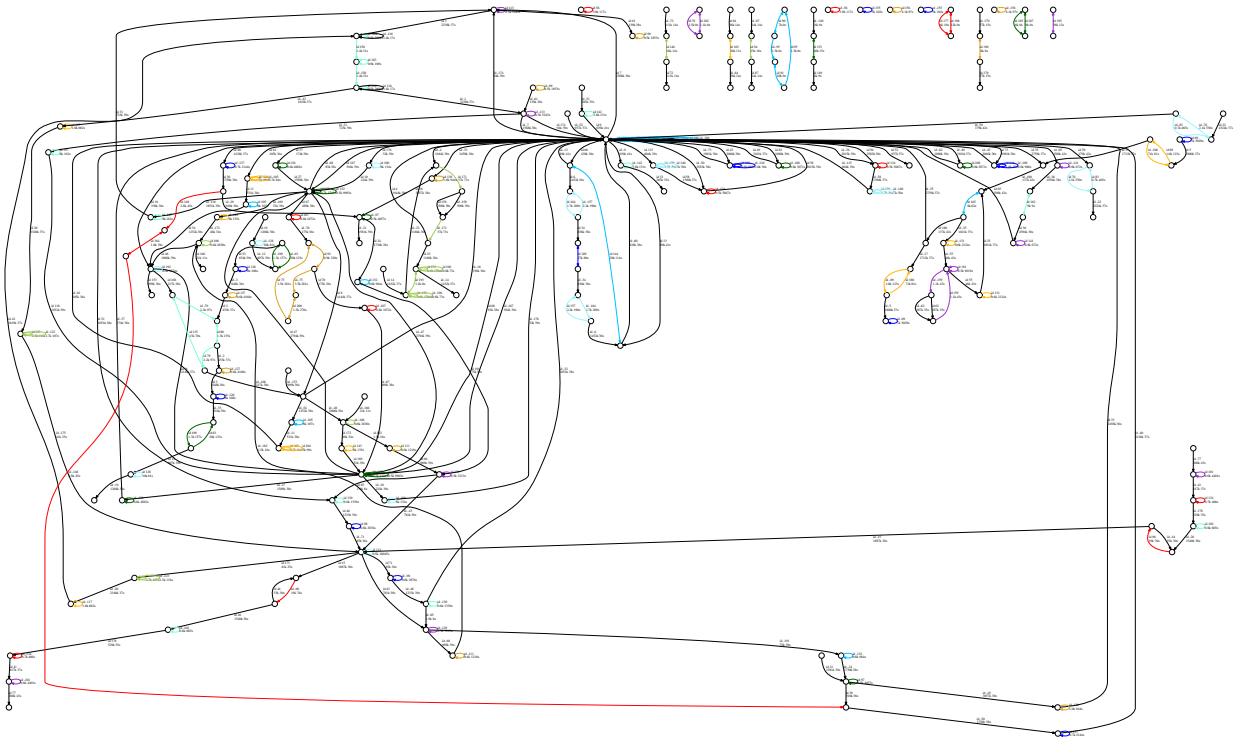

Figure 2.6: The assembly graph of the WORM dataset.
Edges that were classified as repetitive by Flye are shown in color, while unique edges are black. Flye assembled the WORM dataset into a graph with 127 unique and 61 repeat edges and generated 127 contigs as unambiguous paths in the assembly graph. This visualization was generated using the graphviz tool (Ellson et al. 2003).

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 likely represent differences with the reference rather than misassemblies by Flye.

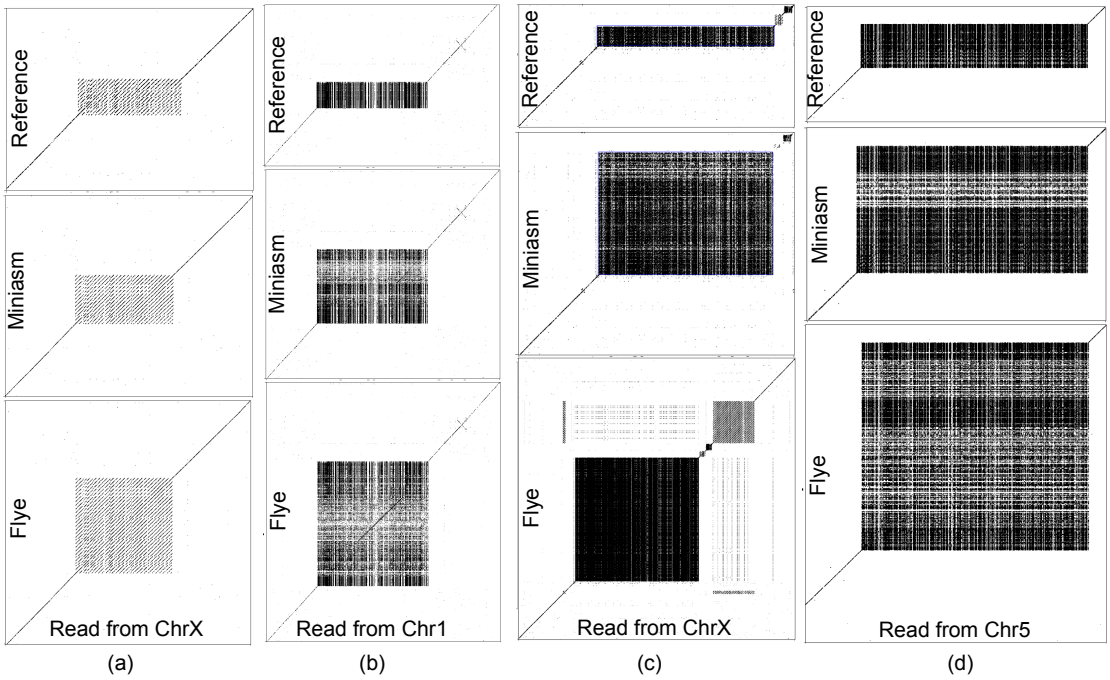

Figure 2.7: Dot-plots showing the alignment of reads against the Flye assembly, the Miniasm assembly and the reference *C. elegans* genome.
(a) The reference genome contains a tandem repeat of length 1.9 kb (10 copies) on chromosome X with the repeated unit having length ≈190 nucleotides. In contrast, 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.

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 (with reads N50 = 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 both the HUMAN and HUMAN+ datasets (Zimin et al. 2017), which are available from the MaSuRCA website.

Currently, the ONT assemblies have rather high base-calling error rates (the Flye and Canu HUMAN assemblies had 1.2% and 2.8% error, respectively) because of the biased error pattern in ONT reads. Although the Nanopolish tool contributed to a reduction in the base-calling error rates of the ONT assemblies (Simpson et al. 2017), the resulting error rate is 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.

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 their 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 (1500 assembly errors) and 2.87 Mb (1200 assembly errors), respectively. The MaSuRCA assembly had slightly higher percent identity with the reference (99.84% as compared to 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.

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. The exact command lines (and the Falcon configuration script) can be found in the supplementary data archive.

- 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 20th, 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 2.5 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 @ 2.00 GHz. RAM usage was 500 GB at peak. The Canu authors

reported ≈30,000 CPU hours of run-time using a cluster with 48-core Intel(R) Xeon(R) CPU @ 2.5 GHz 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 120 GB. The MaSuRCA authors reported needing approximately 50,000 CPU hours.

Table 2.5: 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-PB dataset indicates 9 minutes without polishing and 25 minutes with polishing. "Size" refers to genome size, "Cov" refers to sequencing coverage, "Max threads" is the maximum threads used during assembly, "Mb" indicates megabase-pairs, "G" indicates gigabytes, and "m" indicates minutes.

Dataset	Assembler	Wall clock time	Peak memory usage
YEAST-PB Size: 12 Mb Cov: 31× Max threads: 8	Flye (w/o polishing)	20m (9m)	7G
	Canu	80m	5G
	Falcon	62m	10G
	HINGE	9m	5G
	Miniasm+ABruijn (Miniasm)	16m (1m)	5G
YEAST-ONT Size: 12 Mb Cov: 31× Max threads: 8	Flye (w/o polishing)	19m (12m)	7G
	Canu	184m	6G
	Falcon	103m	11G
	HINGE	11m	8G
	Miniasm+ABruijn (Miniasm)	31m (3m)	5G
WORM Size: 100 Mb Cov: 40× Max threads: 24	Flye (w/o polishing)	128 m (77 m)	30G
	Canu	780 m	41G
	Falcon	945m	18G
	HINGE	803m	52G
	Miniasm+ABruijn (Miniasm)	290m (10m)	23G

Segmental Duplications in the Human Genome.

The repeat graph constructed by Flye reveals the complex mosaic structure of *segmental duplications (SDs)*. 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 (details in the Methods section). After removing all of the unique edges from the assembly graph, only connected components made up of repeat edges remain, each of which encodes an SD. We define the complexity of an SD as the number of edges in its connected component, and the length as the total length of all edges in its connected component. Figure 2.8, Left illustrates a mosaic SD of complexity 7 and length 25.7 kb (the seven colored repeat edges form a connected component in the Flye assembly graph after removing all unique edges). An SD 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 SDs identified by Flye and illustrates the power of the assembly graph for repeat resolution.

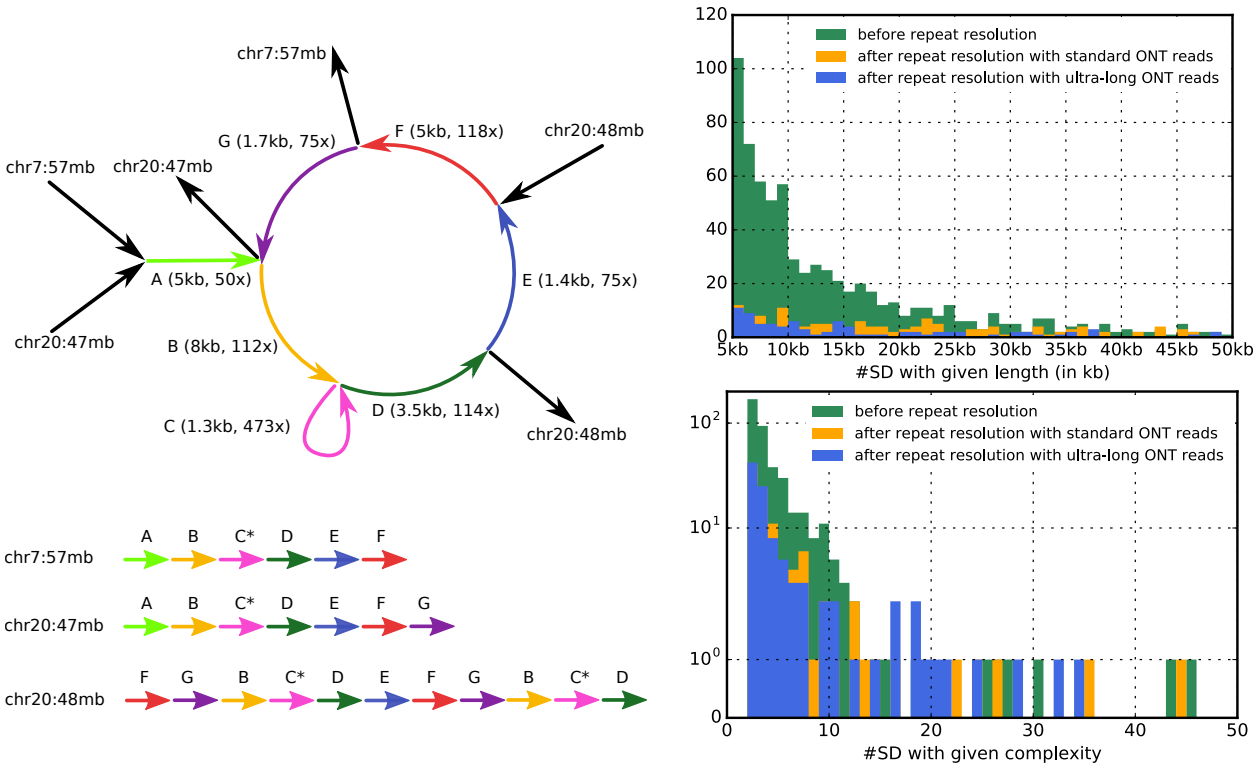

Figure 2.8: The distribution of the lengths and complexities of all SDs from the Flye assembly of the HUMAN dataset (Right) and a detailed example of one such SD (Left). **(Left)** A mosaic SD of complexity 7 is represented as a connected component formed by repeat edges (there are seven colored edges of total length 25.7 kb) in the assembly graph of the HUMAN dataset (flanking unique edges are shown in black). Loop-edge C with coverage 473× represents a tandem repeat C* 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. **(Right)** Statistics are given before resolving bridged repeats (green), after resolving bridged repeats with standard ONT reads (orange), and with standard and 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 repeat resolution. 545 out of 688 of SDs between 5 kb and 50 kb were resolved using the standard ONT reads, and the ultra-long reads resolved an additional 58 SDs. There were 1,256 simple SDs before repeat resolution and 143 after repeat resolution with ultra-long reads. Since Flye already normally resolves SDs shorter than the typical read length, these identified SDs 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 repeat resolution. After repeat

resolution with ultra-long reads, there are only 765 repeat edges, forming 107 connected components in the assembly graph. 73 of them represent mosaic SDs, and 34 of them represent simple SDs (most simple SDs represent isolated edges and loop-edges).

A similar procedure is applied to the HUMAN+ dataset where unique edges are removed and connected components formed by repeat edges remain. These connected components correspond to putative SDs, though they 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 SDs (longer than 50 kb) for the HUMAN+ dataset.

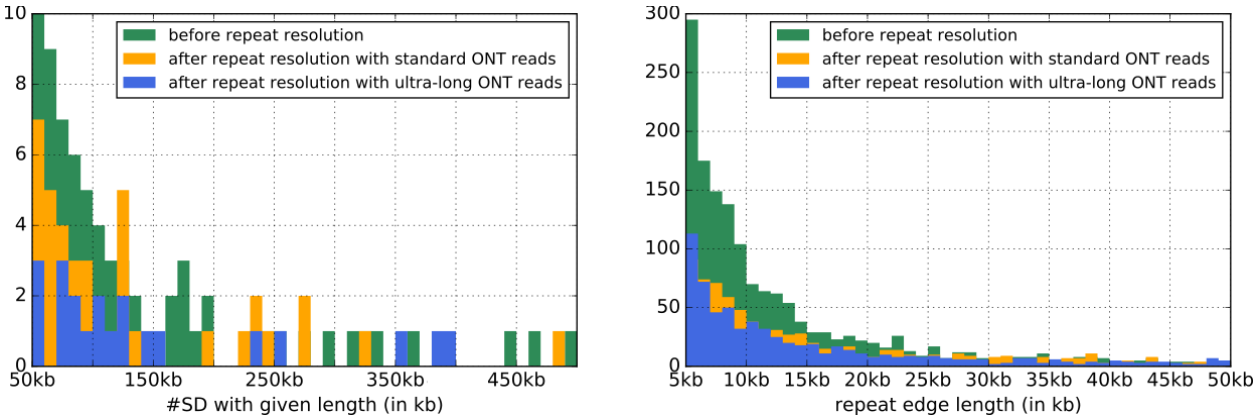

Figure 2.9: The distribution of lengths of ultra-long SDs (longer than 50 kb) for the assembly graph constructed for the HUMAN+ dataset (left) and the lengths of all other repeat edges (right).
(Left) 39 out of 81 SDs (48%) longer than 50 kb were resolved using standard ONT reads to bridge repeats. Ultra-long 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 repeat edges with length exceeding 50 kb before repeat resolution, and 90 repeat edges of this length remaining after repeat resolution with ultra-long reads. Note that while a similar figure in the main text describes the lengths of SDs (the total length of edges in the connected components of the SDs), this figure describes the length of individual repeat edges.

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 26× to 31×, and in divergence from 1.77% to 7.76%.

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 (Rep ID) is given by its edge id in the assembly graph. All repeats have been resolved. "Rep Len" refers to the estimated repeat length of the repeat. The "Cov" or 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 "Div" or 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). "Max Dist btw Pos" 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. See the Methods section for more details.

Rep ID	Rep Len (kb)	Cov	Div	#Tentative Divergent Positions	#Confirmed Divergent Positions	Max Dist btw Pos (kb)	Remaining Gap (kb)	#*cis* Linking Reads	#*trans* Linking Reads
625	152	27×	5.36%	29713	3256	79.2	32.2	2	12
902	51	28×	1.77%	5694	1541	0.7	0	43	13
1018	86	26×	6.77%	17509	11360	0.7	0	17	154
1075	37	28×	3.05%	4379	1406	0.3	0	38	136
1233	49	31×	7.76%	11786	8590	0.3	0	45	2

All resolved repeats correspond to known segmental duplications in the human genome. The sequences of the constructed repeat copies preferentially map to specific copies of segmental duplications, showing that our method is successful even in the presence of Single Nucleotide

Variants (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 2.6 (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.

A Theoretical Framework for Repeat Graph Construction.

In addition to the described Flye algorithm, we provide a mathematical formulation of the repeat characterization problem and describe an alternative algorithm for the repeat graph construction (Figure 2.10). The Methods section provides additional details and explains the relation between the theoretical framework and the implementation in Flye.

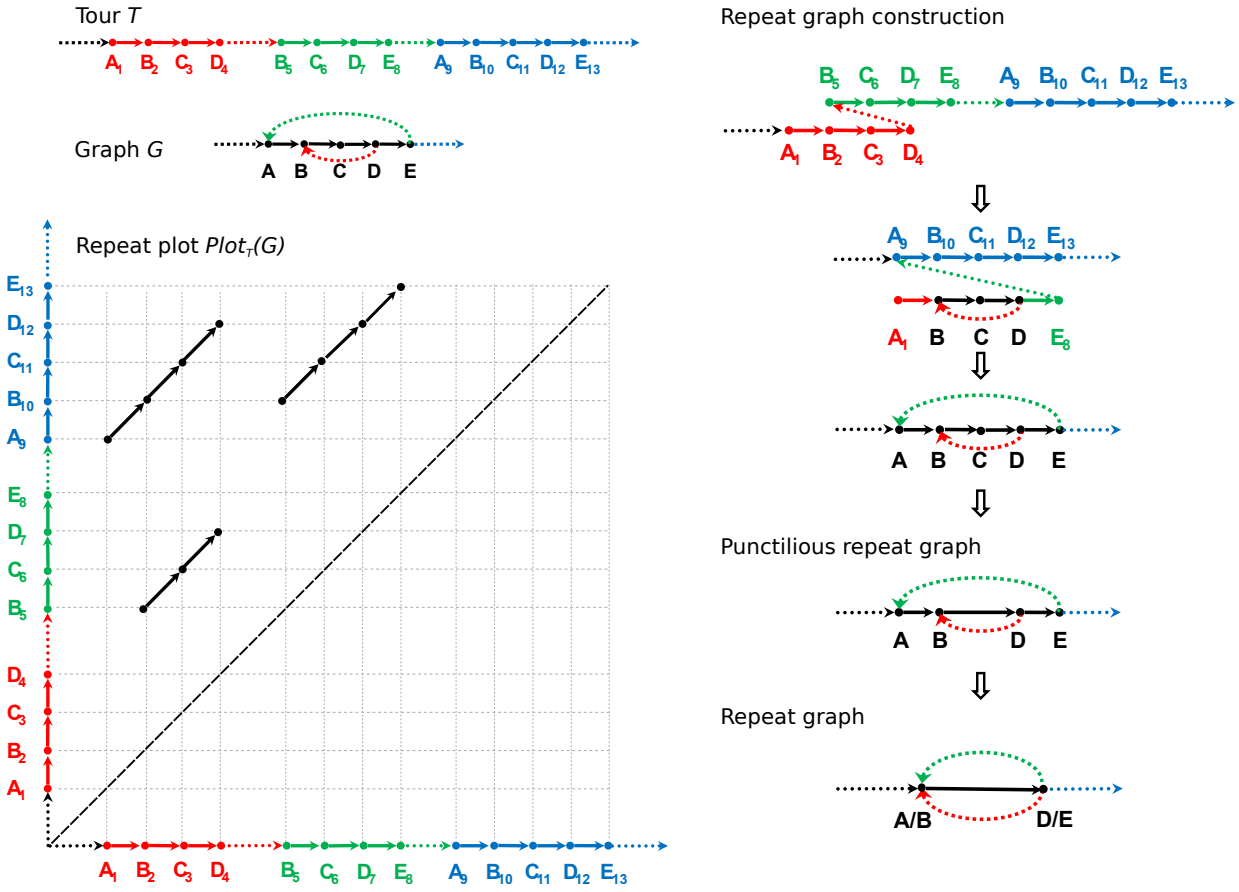

Figure 2.10: Constructing the repeat plot of a tour in the graph (Left) and constructing the repeat graph from a repeat plot (Right).
(Left) A tour $T = \ldots A_1B_2C_3D_4 \ldots B_5C_6D_7E_8 \ldots A_9B_{10}C_{11}D_{12}E_{13} \ldots$ in a graph G with red, green, and blue instances of a repeat that 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). **(Right)** 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 **(Upper Bottom Right)** is transformed into the repeat graph **(Lower Bottom Right)** by contracting short edges (A, B) and (D, E).

2.4 Discussion

We describe the Flye algorithm for constructing the assembly graph of 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.

For the BACTERIA datasets, Flye and HINGE showed good agreement in the structure of constructed assembly graphs. Flye showed substantial improvement over 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 for 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 SDs. Our algorithm for resolving unbridged repeats resolved only a small fraction of these SDs since it is currently limited to simple SDs (the vast majority of human SDs are mosaic and complex). Moreover, it currently has difficulties resolving *highly similar SDs*, e.g. SDs with ~1% divergence. Although we reported the

resolution of highly similar SDs on simulated datasets (as did a previous study; Tischler et al. 2017), most unbridged repeats resolved by Flye and Canu are simple repeats with divergence exceeding 3%. Extending Flye to mosaic SDs and highly similar SDs has the potential to resolve most of the remaining unbridged repeats, since the vast majority of SDs in the human genome diverged by more than 1% (Pu et al. 2018). Since there are only 53 long SDs (with length exceeding 15 kb) in the human genome that diverged 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 SDs (Jiang et al. 2007; Pu et al. 2018). Flye assembly graphs provide a useful framework for reconstructing SDs and planning additional genome finishing experiments.

2.5 Methods

The Repeat Characterization Problem.

Below we describe the abstract repeat characterization problem and explain how it relates to genome assembly. Consider a tour $T = v_1, v_2, \ldots v_n$ of 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, i.e., $v_i = v_j$. 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 (Figure 2.10). 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, which 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 et al. 1970). In the case of repeat characterization, we are interested in the dot-plot $Plot$ formed by the 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 (where 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, e.g., 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 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; i.e., reads longer than N90 account for ≈90%

of the total read length ($minOverlap$ varies from 3000 to 5000 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 (Figure 2.10) 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 (i.e., vertices with either in-degree or out-degree not equal to one) 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 (Figure 2.10).

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 m copies in the genome results in $\binom{m}{2}$ pairwise alignments and each of the corresponding $\binom{m}{2}$ alignment-paths may have unique starting and/or ending vertices that differ from all other starting/ending positions, there will be many gluing operations for the starting and/or 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 Figure 2.10 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 pairwise 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 (Figure 2.10). 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 one are called unique edges and all other edges are called repeats.

Approximate Repeat Graphs.

The described approach, although simple in theory, results in various complications in the case of real genomes, particularly in the case of inconsistent pairwise alignments. 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 (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 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; e.g., vertices A and B were contracted into a single vertex A/B in the repeat graph in Figure 2.10. 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 Figure 2.10) 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 Figure 2.10). This set of vertices forms a set of short, contiguous genomic segments (segments [1, 2], [4, 5], [7, 8, 9, 10], and [12, 13] in Figure 2.10) 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 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 Figure 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 Figure 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. Below, we describe how pairwise alignments may be inconsistent and explain how mosaic repeats and inconsistencies of local alignments may result in an "incorrect" clustering-based repeat graph.

Inconsistent Pairwise Alignments.

Pevzner et al. 2004 introduced the concept of alignment-based de Bruijn graphs known as *A-Bruijn graphs* and applied them to the problems of 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 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: $\mathbf{A_0C_1T_2G_3G_4C_5T_6G_7A_8C_9T_{10}}$ (A_0 and A_8 share the same color; C_1, C_5, and C_9 share the same color, and T_3, T_6, and T_{10} share the same color). Figure 2.11 visualizes such a partitioning as a multiple self-alignment where each column represents positions from the same subset.

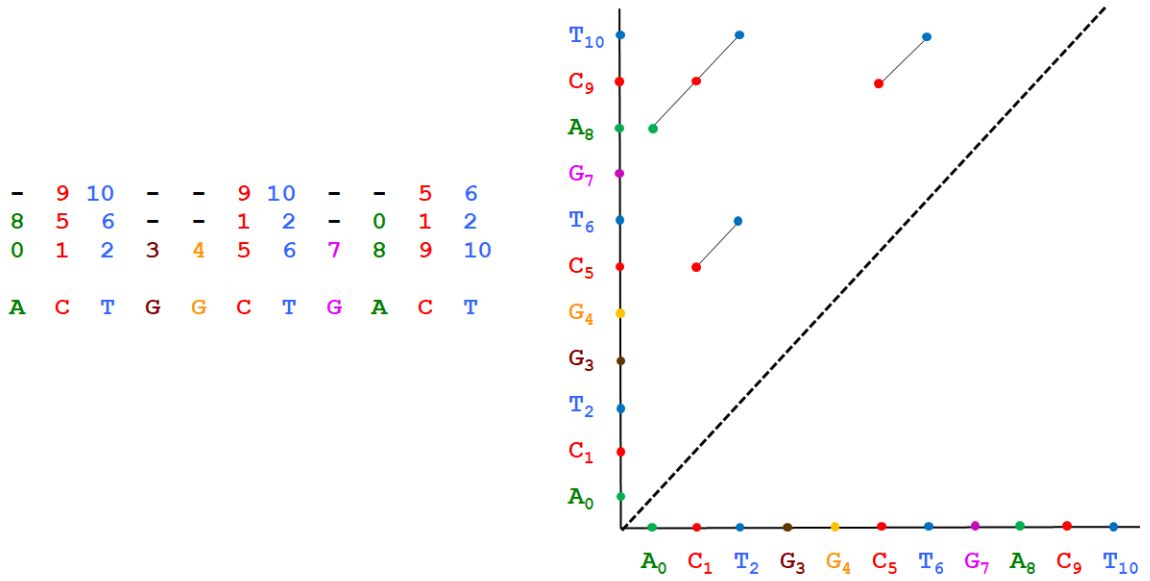

Figure 2.11: Multiple self-alignment defined by the partitioning of $A_0C_1T_2G_3G_4C_5T_6G_7A_8C_9T_{10}$ into six subsets (left) and the corresponding dot-plot (right). In contrast to the traditional representation of a multiple alignment (where each entry represents a nucleotide or a dash in the multiple alignment matrix), each entry in the multiple self-alignment matrix represents a position in the sequence or a dash.

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 2.11 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 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.

Inconsistent Alignments Result in Excessively Complex Repeat Graphs.

Figure 2.12 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. In contrast to the graph in Figure 2.2, the graph in Figure 2.12 is not alignment-compatible; e.g., the repeat **A**+**B** corresponds to a single path in Figure 2.2 but two paths in Figure 2.12. Although it may appear to be a minor annoyance in the case of the toy example in Figure 2.12, inconsistent alignments may result in excessively complex repeat graphs for 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 2.12 (by adding the 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 2.12 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 2.12 motivates us to develop a new algorithm for extending the set $Breakpoints$ described below. Note that the middle point of the long diagonal in Figure 2.12 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 below adds the missing projection to the set $Breakpoints$ and results in the same approximate repeat graph as shown in Figure 2.2 (Figure 2.12, bottom panel).

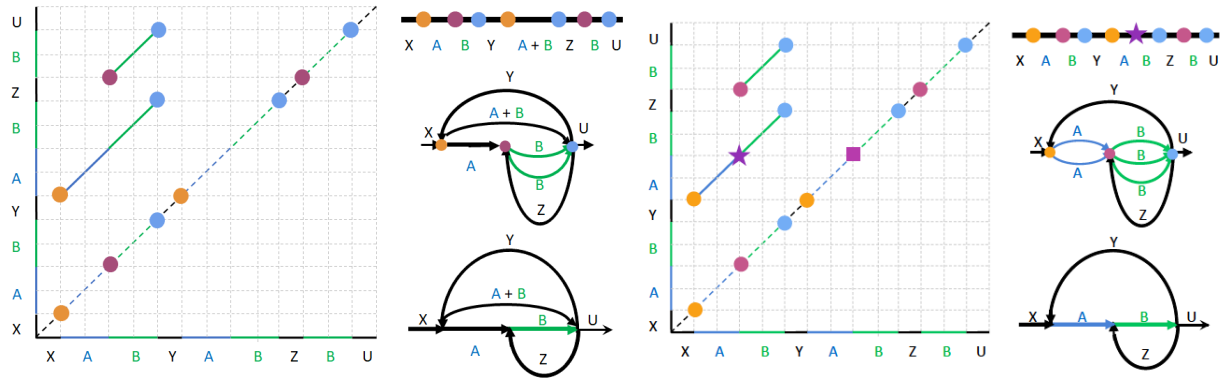

Figure 2.12: Inconsistent pairwise alignments result in an "incorrect" repeat graph (as compared to the graph shown in Figure 2.2), thus necessitating an extension of the set of alignment endpoints.
(Left) Alignment-paths for two pairwise self-alignments within a genome X**AB**Y**AB**Z**B**U. Only two 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 of Left Half)** Projections of the clustered endpoints on the main diagonal define seven vertices of the approximate repeat graph. **(Middle Right of Left Half)** Gluing breakpoints that belong to the same clusters. **(Bottom Right of Left Half)** 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.

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 Figure 2.2 (shown in purple) belong to $Breakpoints$. Such internal points should be re-classified 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 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. Figure 2.12 provided an example of an invalid point and how it affects the resulting repeat graph.

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 $BreakpointSegments$. Two segments from $BreakpointSegments$ are *equivalent* if there exists a point in one of the alignment-paths such that one of its projections to 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 $BreakpointSegments$ 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 to 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 $BreakpointSegments$, selecting a middle point from each segment in $BreakpointSegments$, 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 $Alignments$, i.e., 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.

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 et al. 2015 reported that the Celera (Myers et al., 2000), Minimus (Treangen et al., 2013), and Lola (Sharon et al., 2015) assemblers produced suboptimal assemblies of contigs

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

Similar challenges remain unresolved for short reads as well. 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 (Prjibelsky 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 to this graph again. Another advantage of this (less tangled) contig-based assembly graph lies in applications relating to hybrid assembly, e.g., the 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 thus far.

From the Repeat Graph of a Genome to the Assembly Graph of Contigs.

Due to the challenges described above, the ABruijn assembler (Lin et al. 2016) opted to output a set of contigs rather than constructing the repeat graph of a genome based on these contigs. 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, i.e., when it reaches a branching vertex in the (unknown) repeat graph. Afterwards, ABruijn decides whether to continue or to stop the path extension in order 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 at least $minOverlap$ nucleotides in front of the first branching vertex and behind 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 significant effort into making sure that the generated contigs are correctly assembled (that they 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 from error-prone disjointigs (also known as an assembly graph).

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 (which has a 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 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).

The FlyeWalk Algorithm.

The **FlyeWalk** algorithm (Figure 2.13) computes alignments (within the **Overlap**, **MapReads**, and **ExtendRead** procedures) using the longest $jump$-subpath approach described in Lin et al. 2016. In contrast to 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*.

```
FlyeWalk(AllReads, MinOverlap)
    Disjointigs ← empty set of contigs
    UnprocessedReads ← AllReads
    for each Read in UnprocessedReads
        ChainOfReads ← ExtendRead(UnprocessedReads, Read, MinOverlap)
        DisjointigReads ← MapReads(AllReads, ChainOfReads, MinOverlap)
        DisjointigSequence ← Consensus(AllReads, DisjointigReads, MinOverlap)
        add DisjointigSequence to Disjointigs
        remove DisjointigReads from UnprocessedReads
    return Disjointigs

ExtendRead(UnprocessedReads, Read, MinOverlap)
    ChainOfReads ← sequence of reads consisting of a single read Read
    while forever
        NextRead ← FindNextRead(UnprocessedReads, Read, MinOverlap)
        if NextRead = empty string
          return ChainOfReads
        else
          add NextRead to ChainOfReads
          Read ← NextRead
          remove Readfrom UnprocessedReads
```

Figure 2.13: The pseudocode for 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.

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 the chain

$ChainOfReads$ by constructing $concatenate(ChainOfReads)$. Afterwards, all reads from the dataset are aligned to the draft disjointig sequence using minimap2 (Li 2017) 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 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.

Constructing 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 (where 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). Thus, the assembly graph constructed from accurate contigs (which can be viewed as virtual reads) also approximates the repeat graph of the genome. This is explained in further detail below.

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 2.3, Middle: 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 $IN_1 \rightarrow REP \rightarrow OUT_2 \rightarrow REP'$ and $IN_2 \rightarrow REP \rightarrow OUT_1 \rightarrow REP'$ 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 $Strings = \{s(1), \dots, s(t)\}$ 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 $Strings^* = s(1) * s(2) * \dots * s(t)$ 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 $s(1), \ldots, s(t)$ 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 $\{disjointig_1, disjointig_2, \ldots, disjointig_t\}$ 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:

$$\{s(1,1), s(1,2), \ldots, s(1,n_1)\}, \{s(1,1), s(2,1), \ldots, s(2,n_2)\}, \ldots, \{s(t,1), s(t,1), \ldots, s(t,n_t)\}$$

where $s(i,j)$ stands for the j-th read in the i-th disjointig (reads are ordered in increasing order based on 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; i.e., non-overlapping reads within this disjointig may share sufficiently long alignments that however do not contribute to first-stage gluing.

The first-stage gluing of reads that were sampled from a single disjointig results in the same consensus of this disjointig that is constructed by **FlyeWalk**. Thus, the application of such "intra-disjointig" gluing operations to all disjointigs results in the set of disjointigs $\{disjointig_1, disjointig_2, \ldots, disjointig_t\}$. 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 of the disjointigs constructed by **FlyeWalk** (some of them may be misassembled) results in the same assembly graph as the gluing of all of the reads, and thus results in the repeat graph of the genome.

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 $Genome$ and $Alignments$ are unknown but the alignments between substrings of the genome (i.e. the 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 (i.e. the 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 all of $Genome$.

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 (i.e. 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 the 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 (i.e. a read) spanning all these positions. This condition explains why it is important that Flye generates disjointigs satisfying the overhang property.

After constructing the repeat graph, Flye proceeds to simplify this graph to improve the assembly. Figure 2.3, Left presents the assembly graph of the SMS reads from an *E. coli* genome. Flye further untangles this graph into a graph with just six edges (Figure 2.3, Middle) following the procedure described below.

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 these edges using dynamic programming.

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 candidate successor edges: 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 for unique edges or multiple alignments for repetitive edges. This analysis revealed that the *C. elegans* assembly graph has 339 unique and 219 repetitive edges. Flye 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 (misclassifying 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.

Resolving Bridged Repeats in the Assembly Graph.

As described above, Flye aligns all reads to the constructed assembly graph and uses them to identify the repeat edges in this graph. It further transforms the assembly graph into the condensed assembly graph by contracting all 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 from 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 $transition(e, f)$ as the number of (e, f)-reads plus the number of (f', e')-reads, where e' and f' are the complementary edges of e and f, i.e., 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 $transition(e, f)$.

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 1965) 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 (explained further, below) 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.

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 two 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 had been a part of a bigger mosaic repeat that was partially resolved).

Resolving Unbridged Repeats in the Assembly Graph.

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 upon 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 (described in further detail below). The current version of Flye is limited to resolving unbridged repeats of multiplicity two in both haploid (e.g., bacterial) and diploid (e.g., 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. For example, the 93 reads that traverse edges IN_1 and REP (Figure 2.3) 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 from the incoming edges (93 and 75 reads in Figure 2.3) 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 Figure 2.3) 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 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.

Revealing Variable Positions Within Repeats.

To reveal the variable positions within a repeat (which corresponds to 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. (The earlier section titled "Human

Segmental Duplications Identified by Flye" discussed 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 2.3) in the assembly graph of the EC9964 dataset (other repeats result in very similar distributions). Since this repeat has many variations between its 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 its substitution, deletion, and insertion rates. Figure 2.14 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%.

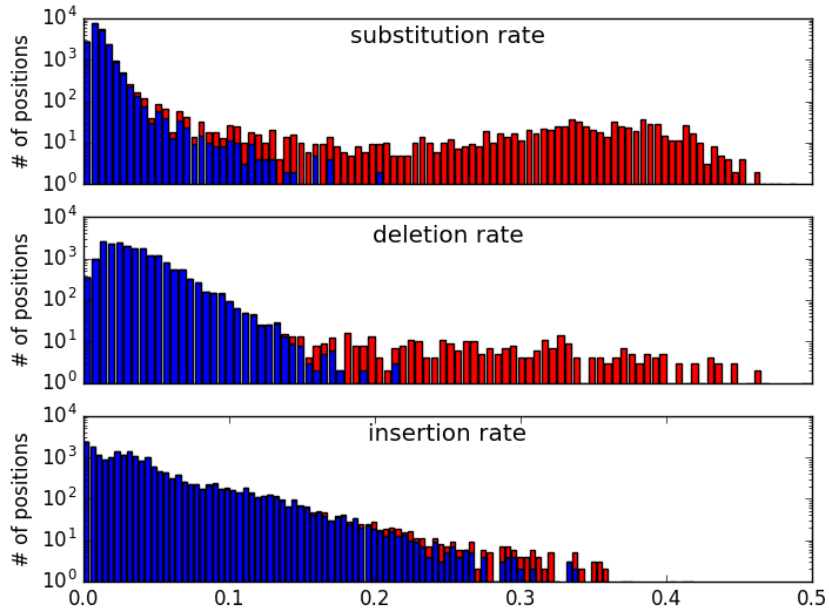

Figure 2.14: Separating variable and non-variable positions within repeats using substitution, deletion, and insertion rates computed for the *REP* repeat in the EC9964 dataset.
Substitution **(Top)**, deletion **(Middle)**, and insertion **(Bottom)** rates at each position in the multiple alignment of reads. Blue bars represent mutation rates for non-variable positions, and red bars represent rates for variable positions. The number of positions with a given mutation rate (shown on the 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.

Additional Details for the Unbridged Repeat Resolution Approach.

Initially, the unbridged repeat resolution algorithm recruits all reads traversing edges IN_1 and REP to the first repeat copy and all reads traversing IN_2 and REP to the second repeat copy, and then it computes the consensus of each repeat copy using these 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 each repeat copy to ensure that consensus sequences are

sufficiently accurate (the default value of $minCoverage = 10\times$). 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 2.3). 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 the prefixes of the 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 error rate when the coverage is as low as 5-10× (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 (using full coverage) rather than only reads contributing to a given template (which have coverage as low as $minCoverage$), they provide a more reliable standard for read recruitment.

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

- *Evaluating the tentative divergent positions.* We map all classified reads again, this time to two consensus copies of the repeat (rather than to 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 the 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 the 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 its vote for another copy by at least 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 the generated 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 2.3), the same procedure is performed while "moving backward" in the opposite direction (e.g., moving backwards from edges OUT_1 and OUT_2 in Figure 2.3), or equivalently, moving forward along the reverse complement of the repeat. There are two stopping rules for the described algorithm: (i) when the constructed prefix of the repeat resulting from "moving forward" overlaps with the constructed 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 2.3), this procedure may also 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: pairing IN_1/OUT_1 with IN_2/OUT_2, or pairing IN_1/OUT_2 with IN_2/OUT_1.

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 (though some resolvable repeats may still be classified as unresolved). Note that even in the case of unresolved repeats, this algorithm still finds more accurate consensus sequences for the prefixes and suffixes of the repeat. Table 2.7 presents the results of running the unbridged repeat resolution algorithm on 22 unbridged repeats of multiplicity two from 11 genomes in the BACTERIA dataset.

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. * indicates a repeat of multiplicity 3. "Cov" or 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). "Div" or 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). "Max Dist btw Pos" refers to the maximal distance between adjacent confirmed divergent positions. "Rem 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. In the case that the forward and reverse consensus sequences overlap, the remaining gap is set to 0. "#Tent Div Pos" is the number of tentative divergent positions, and "#Conf Div Pos" is the number of confirmed divergent positions.

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

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 can 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 PacBio reads from these repeats with coverage 100×, 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 to use as the maximum distance 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.

Evaluating the Accuracy of the Unbridged Repeat Resolution Approach.

To evaluate the accuracy of the unbridged repeat resolution approach, we simulated a 1 Mb genome which contains two copies of a single repeat of length L (for $L = 10$ kb, 20 kb, and 40 kb) with divergence x% (for $x = 0.05\%$, 0.15%, and 0.45%). We further used the PBSim tool (Ono et al. 2013) to simulate PacBio reads from this genome with coverage 100× and length varying from 5 kb to 15 kb. The PBSim tool generated reads with an insertion rate of 9%, a deletion rate of 4.5%, and a substitution rate of 1.5%. We also generated two replicates for each simulation, performing $3 * 3 * 2 = 18$ simulations in total.

All repeats of length 10 kb were resolved by Flye prior to unbridged repeat resolution, so they were not included in this analysis. Of the remaining repeats, all repeats with divergence

0.15% and 0.45% were resolved correctly with overwhelming support from linking reads, and the repeat copy sequence reconstructions had accuracy >99.95%. All repeats with divergence 0.05% were not resolved due to the scarcity of divergent positions, but over half of the sequences of each repeat copy were reconstructed with accuracy exceeding 99.8%.

We further lowered the divergence rate to 0.01%, 0.02%, 0.03%, and 0.04% and repeated the experiment. With such low divergence rates, the repeat sequences could not be fully reconstructed, but all reconstructed sequences still had ≥99.9% accuracy. We also lowered the coverage to 30× and 50× and repeated the experiment with similar results (Tables 2.8 – 2.10).

Based on these simulations, we conclude that our unbridged repeat resolution approach correctly links incoming and outgoing edges using the evidence from linking reads, and it can successfully reconstruct the sequences of distinct repeat copies. Even at low divergence rates and low coverage, our reconstructions are very accurate though we may only reconstruct partial sequences due to regions with no divergent positions or low coverage.

Table 2.8: Unbridged repeat resolution simulation results.
Genomes containing repeats of multiplicity two were simulated for repeat lengths 20 kb and 40 kb, and divergence rates 0.05%, 0.15%, and 0.45%. PacBio reads were simulated for these genomes with a mean error rate of 15% and lengths ranging from 5 kb to 15 kb. Our unbridged repeat resolution approach was applied to these reads to determine how these repeats should be resolved (by pairing incoming with outgoing edges) and to reconstruct the distinct repeat copy sequences for each of these simulations. "Len," "Cov" and "Div" are the length, coverage, and divergence rates of the simulations, respectively. "Rep" is the replicate. "#Corr Div Pos" and "#Conf Div Pos" are the number of correct confirmed divergent positions and the total number of confirmed divergent positions found by Flye. "Max Dist btw Pos" refers to the maximal distance between adjacent confirmed divergent positions. "Mean Rec Seq Len" and "Mean Rec Seq Acc" are the mean reconstructed sequence length and accuracy, respectively.

Len (kb)	Cov (×)	Div (%)	Rep	#Corr Div Pos	#Conf Div Pos	Max Dist btw Pos (kb)	#Linking Reads	Resolved	Mean Rec Seq Len (kb)	Mean Rec Seq Acc
20	100	0.05	A	7	7	5.0	0	NO	14	99.93%
			B	9	10	10.0	0	NO	20	99.79%
		0.15	A	32	40	2.8	439	YES	20	99.98%
			B	35	41	2.3	462	YES	20	99.97%
		0.45	A	103	109	0.87	471	YES	20	99.96%
			B	81	88	1.5	552	YES	20	99.98%
40	100	0.05	A	19	8	28.0	0	NO	40	99.88%
			B	18	6	32.9	0	NO	15	99.94%
		0.15	A	68	82	2.8	840	YES	40	99.95%
			B	64	73	2.9	788	YES	40	99.97%
		0.45	A	194	215	1.0	842	YES	40	99.97%
			B	169	188	1.5	861	YES	40	99.97%

Table 2.9: Unbridged repeat resolution low divergence simulation results.
Genomes containing repeats of multiplicity two were simulated for very low divergence rates from 0.01% to 0.04%. Although our unbridged repeat resolution approach was not able to resolve these repeats, we still partially reconstructed the repeat copy sequence with high accuracy. *One simulation with repeat length 40 kb and divergence 0.04% did not generate repeat sequences for the unbridged repeat resolution tool to be applied so no results are shown. "Len," "Cov" and "Div" are the length, coverage, and divergence rates of the simulations, respectively. "Rep" is the replicate. "#Corr Div Pos" and "#Conf Div Pos" are the number of correct confirmed divergent positions and the total number of confirmed divergent positions found by Flye. "Max Dist btw Pos" refers to the maximal distance between adjacent confirmed divergent positions. "Mean Rec Seq Len" and "Mean Rec Seq Acc" are the mean reconstructed sequence length and accuracy, respectively.

Len (kb)	Cov (×)	Div (%)	Rep	#Corr Div Pos	#Conf Div Pos	Max Dist btw Pos (kb)	#Linking Reads	Resolved	Mean Rec Seq Len (kb)	Mean Rec Seq Acc
20	100	0.01	A	1	1	19.5	0	NO	11	99.96%
			B	2	2	17.7	0	NO	11	99.95%
		0.02	A	2	2	13.0	0	NO	12	99.93%
			B	3	5	10.2	0	NO	16	99.96%
		0.03	A	6	8	7.8	0	NO	13	99.92%
			B	4	5	13.5	0	NO	12	99.94%
		0.04	A	7	7	7.8	0	NO	14	99.96%
			B	7	6	13.4	0	NO	11	99.92%
40	100	0.01	A	5	4	34.2	0	NO	13	99.94%
			B	3	4	33.3	0	NO	12	99.94%
		0.02	A	7	8	26.3	0	NO	15	99.96%
			B	6	1	38.0	0	NO	15	99.91%
		0.03	A	14	5	28.0	0	NO	16	99.93%
			B	8	2	31.3	0	NO	14	99.94%
		0.04	A*	17	4	29.7	0	NO	13	99.89%

Table 2.10: Unbridged repeat resolution low coverage simulation results.
Genomes containing repeats of multiplicity two were simulated for low coverage rates of 50× and 30×. The other conditions were kept the same as Table 2.8. Even at lower coverage, most repeats above 0.1% divergence were able to be resolved. "Len," "Cov" and "Div" are the length, coverage, and divergence rates of the simulations, respectively. "Rep" is the replicate. "#Corr Div Pos" and "#Conf Div Pos" are the number of correct confirmed divergent positions and the total number of confirmed divergent positions found by Flye. "Max Dist btw Pos" refers to the maximal distance between adjacent confirmed divergent positions. "Mean Rec Seq Len" and "Mean Rec Seq Acc" are the mean reconstructed sequence length and accuracy, respectively.

Len (kb)	Cov (×)	Div (%)	Rep	#Corr Div Pos	#Conf Div Pos	Max Dist btw Pos (kb)	#Linking Reads	Resolved	Mean Rec Seq Len (kb)	Mean Rec Seq Acc
20	50	0.05	A	7	2	19.4	0	NO	13	99.92%
			B	9	11	11.9	0	NO	10	99.87%
		0.15	A	32	34	3.8	20	YES	20	99.97%
			B	35	42	2.3	21	YES	20	99.94%
		0.45	A	103	113	0.9	252	YES	20	99.95%
			B	81	90	1.5	289	YES	20	99.97%
40	50	0.05	A	19	6	30.6	0	NO	8	99.87%
			B	18	4	34.8	0	NO	12	99.88%
		0.15	A	68	81	3.8	13	YES	40	99.95%
			B	64	75	2.9	35	YES	40	99.95%
		0.45	A	194	213	0.9	425	YES	40	99.96%
			B	169	189	1.5	411	YES	40	99.97%
20	30	0.05	A	7	4	17.4	0	NO	10	99.87%
			B	9	6	16.9	0	NO	6	99.82%
		0.15	A	32	36	7.6	2	NO	19	99.89%
			B	35	39	2.9	7	YES	18	99.91%
		0.45	A	103	117	0.9	30	YES	20	99.93%
			B	81	94	1.5	23	YES	20	99.89%
40	30	0.05	A	19	2	37.2	0	NO	5	99.85%
			B	18	4	36.7	0	NO	9	99.80%
		0.15	A	68	46	19.3	0	NO	23	99.90%
			B	64	9	36.9	0	NO	12	99.68%
		0.45	A	194	220	0.9	175	YES	40	99.91%
			B	169	192	1.0	208	YES	40	99.92%

2.6 Additional Information

Author Contributions.

All authors contributed to developing the Flye algorithms and writing the paper. Mikhail Kolmogorov, Yu Lin, and Jeffrey Yuan implemented the Flye algorithm. Mikhail Kolmogorov benchmarked Flye and other assembly tools. Pavel A. Pevzner directed the work.

Competing Financial Interests.

The authors declare no competing financial interests

Code Availability.

The Flye code used in this study is available in the online version of the paper. The most recent Flye version is freely available at *http://github.com/fenderglass/Flye*.

Data Availability.

All described datasets are publicly available through the corresponding repositories:

- The supplementary files, including the assemblies generated by Flye, are available at *https://doi.org/10.5281/zenodo.1143753/*.
- NCTC PacBio reads: *http://www.sanger.ac.uk/resources/downloads/bacteria/nctc/*.
- PacBio metagenome dataset: *https://github.com/PacificBiosciences/DevNet/wiki/Human_Microbiome_Project_MockB_Shotgun/*.

- PacBio *C. elegans* dataset: *https://github.com/PacificBiosciences/DevNet/wiki/C.-elegans-data-set/*.
- PacBio / ONT *S. cerevisiae* dataset: *https://github.com/fg6/YeastStrainsStudy/*.
- The ONT reads from the HUMAN/HUMAN+ datasets are available at: *https://github.com/nanopore-wgs-consortium/NA12878/*. The matching Illumina reads are available as SRA project ERP00122.
- The Canu HUMAN+ assembly was downloaded from: *https://genomeinformatics.github.io/na12878update/*.
- MaSuRCA assemblies for HUMAN and HUMAN+ are available from: *http://masurca.blogspot.com/*.

2.7 Acknowledgements

We are indebted to Sergey Nurk for his multiple rounds of critique and suggestions that have greatly improved the paper. We are also grateful to Alla Mikheenko, Bahar Behsaz, Lianrong Pu, and Glenn Tesler for their insightful comments. This work is supported by NSF/MCB-BSF grant 1715911.

Chapter 2, in full, has been accepted for publication as "Assembly of long error-prone reads using repeat graphs" as it will appear in *Nature Biotechnology* by Mikhail Kolmogorov, Jeffrey Yuan, Yu Lin, and Pavel A. Pevzner. The material has been reformatted with some minor revisions and edits for improved readability. The dissertation author was a primary author of this material.

2.8 References

Antipov, D., Korobeynikov, A., McLean, J. S., & Pevzner, P. A. hybridSPAdes: an algorithm for hybrid assembly of short and long reads. *Bioinformatics*. 2015; 32 (7): 1009-1015.

Bankevich. A., Nurk, S., Antipov, D., Gurevich, A.A., Dvorkin, M., Kulikov, A.S., Lesin, V.M., Nikolenko, S.I., Pham, S., Prjibelski, A.D., Pyshkin, A.V., Sirotkin, A.V., Vyahhi, N., Tesler, G., Alekseyev, M.A. & Pevzner, P.A. SPAdes: A new genome assembly algorithm and its applications to single-cell sequencing. *Journal of Computational Biology*. 2012; 19 (5): 455-477.

Bankevich A. & Pevzner, P.A. TruSPAdes: barcode assembly of TruSeq synthetic long reads. *Nature Methods*. 2015; 13 (3): 248-250.

Bao, Z. & Eddy, S.R. Automated de novo identification of repeat sequence families in sequenced genomes. *Genome Research*. 2002; 12 (8): 1269-1276.

Berlin, K., Koren, S., Chin, C.S., Drake, J.P., Landolin, J.M. & Phillippy, A.M. Assembling large genomes with single-molecule sequencing and locality-sensitive hashing. *Nature Biotechnology*. 2015; 33 (6): 623-630.

Chaisson, M.J., Huddleston, J., Dennis, M.Y., Sudmant, P.H., Malig, M., Hormozdiari, F., Antonacci, F., Surti, U., Sandstrom, R., Boitano, M., Landolin, J.M., Stamatoyannopoulos, J.A., Hunkapiller, M.W., Korlach, J. & Eichler, E.E. Resolving the complexity of the human genome using single-molecule sequencing. *Nature*. 2015; 517 (7536): 608-611.

Chin, C.S., Alexander, D.H., Marks, P., Klammer, A.A., Drake, J., Heiner, C., Clum, A., Copeland, A., Huddleston, J., Eichler, E.E., Turner, S.W. & Korlach, J. Nonhybrid, finished microbial genome assemblies from long-read SMRT sequencing data. *Nature Methods*. 2013; 10 (6): 563-569.

Chin, C.S., Peluso, P., Sedlazeck, F.J., Nattestad, M., Concepcion, G.T., Clum, A., Dunn, C., O'Malley, R., Figueroa-Balderas, R, Morales-Cruz, A., Cramer, G.R., Delledonne, M., Luo, C., Ecker, J.R., Cantu, D., Rank, D.R., & Schatz, M.C. Phased diploid genome assembly with single-molecule real-time sequencing. *Nature Methods*. 2016; 13 (12): 1050-1054.

Edmonds, J. Paths, trees, and flowers. *Canadian Journal of Mathematics*. 1965; 17: 449-467.

Edmonds, J. & Johnson, E.L. Matching, Euler tours and the Chinese postman. *Mathematical Programming.* 1973; 5 (1): 88-124.

Ellson, J., Gansner, E.R., Koutsofios, L., North, S.C. & Woodhull, G. Graphviz and Dynagraph – static and dynamic graph drawing tools. Florham Park, NJ: AT&T Labs – Research; 2003: 127-148. http://graphviz.org.

Ghurye, J., Pop, M., Koren, S., Bickhart, D. & Chin, C.S. Scaffolding of long read assemblies using long range contact information. *BMC Genomics*. 2017; 18 (1): 527.

Gibbs, A.J. & McIntyre, G.A. The Diagram, a Method for Comparing Sequences. Its Use with Amino Acid and Nucleotide Sequences. *European Journal of Biochemistry*. 1970; 16 (1): 1-11.

Giordano, F., Aigrain, L., Quail, M.A., Coupland, P., Bonfield, J.K., Davies, R.M., Tischler, G., Jackson, D.K., Keane, T.M., Li, J., Yue, J.X., Liti, G., Durbin, R. & Ning, Z. De novo yeast genome assemblies from MinION, PacBio and MiSeq platforms. *Scientific Reports*. 2017; 7 (1): 3935.

Jain, M., Koren, S., Miga, K.H., Quick, J., Rand, A.C., Sasani, T.A., Tyson, J.R., Beggs, A.D., Dilthey, A.T., Fiddes, I.T., Malla, S., Marriott, H., Nieto, T., O'Grady, J., Olsen, H.E., Pedersen, B.S., Rhie, A., Richardson, H., Quinlan, A.R., Snutch, T.P., Tee, L., Paten, B., Phillippy, A.M., Simpson, J.T., Loman, N.J. & Loose, M. Nanopore sequencing and assembly of a human genome with ultra-long reads. *Nature Biotechnology*. 2018; 36 (4): 338-345.

Jiang, Z., Tang, H., Ventura, M., Cardone, M.F., Marques-Bonet, T., She, X., Pevzner, P.A. & Eichler, E.E. Ancestral reconstruction of segmental duplications reveals punctuated cores of human genome evolution. *Nature Genetics.* 2007; 39 (11): 1361-1368.

Kamath, G.M., Shomorony, I., Xia, F., Courtade, T.A. & David, N.T. HINGE: long-read assembly achieves optimal repeat resolution. *Genome Research.* 2017; 27 (5): 747-756.

Koren, S., Walenz, B.P., Berlin, K., Miller, J.R., Bergman, N.H. & Phillippy, A.M. Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation. *Genome Research.* 2017; 27 (5): 722-736.

Koren, S., Schatz, M.C., Walenz, B.P., Martin, J., Howard, J.T., Ganapathy, G., Wang, Z., Rasko, D.A., McCombie, W.R., Jarvis, E.D. & Phillippy, A.M. Hybrid error correction and de novo assembly of single-molecule sequencing reads. *Nature Biotechnology*. 2012; 30 (7): 693–700.

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

Li, H. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*. 2018; 34 (18): 3094-3100.

Lin, Y., Yuan, J., Kolmogorov, M., Shen, M.W., Chaisson, M. & Pevzner, P.A. Assembly of long error-prone reads using de Bruijn graphs. *Proceedings of the National Academy of Sciences USA*. 2016; 113 (52): E8396-E8405.

Lin, Y., Nurk, S., Pevzner, P.A. What is the difference between the breakpoint graph and the de Bruijn graph? *BMC Genomics*. 2014; 15: S6.

Mikheenko A., Prjibelski A., Saveliev V., Antipov D. & Gurevich A. Versatile genome assembly evaluation with QUAST-LG. *Bioinformatics*. 2018; 34 (13): i142-i150.

Myers, E.W., Sutton, G.G., Delcher, A.L., Dew, I.M., Fasulo, D.P., Flanigan, M.J., Kravitz, S.A., Mobarry, C.M., Reinert, K.H., Remington, K.A., Anson, E.L., Bolanos, R.A., Chou, H.H., Jordan, C.M., Halpern, A.L., Lonardi, S., Beasley, E.M., Brandon, R.C., Chen, L., Dunn, P.J., Lai, Z., Liang, Y., Nusskern, D.R., Zhan, M., Zhang, Q., Zheng, X., Rubin, G.M., Adams, M.D., Venter, J.C. A whole-genome assembly of Drosophila. *Science*. 2000; 287 (5461): 2196-2204.

Nattestad, M., Goodwin, S., Ng, K., Baslan, T., Sedlazeck, F.J., Rescheneder, P., Garvin, T., Fang, H., Gurtowski, J., Hutton, E., Tseng, E., Chin, C.S., Beck, T., Sundaravadanam, Y., Kramer, M., Antoniou, E., McPherson, J.D., Hicks, J., McCombie, W.R. & Schatz, M.C. Complex rearrangements and oncogene amplifications revealed by long-read DNA and RNA sequencing of a breast cancer cell line. *Genome Research.* 2018; 28 (8): 1126-1135.

Nowoshilow, S., Schloissnig, S., Fei, J.F., Dahl, A., Pang, A.W.C., Pippel, M., Winkler, S., Hastie, A.R., Young, G., Roscito, J.G., Falcon, F., Knapp, D., Powell, S., Cruz, A., Cao, H., Habermann, B., Hiller, M., Tanaka, E.M. & Myers, E.W. The axolotl genome and the evolution of key tissue formation regulators. *Nature*. 2018; 554 (7690): 50-55.

Nurk, S., Meleshko, D., Korobeynikov, A. & Pevzner, P.A. metaSPAdes: a new versatile metagenomic assembler. *Genome Research*. 2017; 27 (5): 824-834.

Ono, Y., Asai, K. & Hamada, M. PBSIM: PacBio reads simulator—toward accurate genome assembly. *Bioinformatics.* 2013; 29 (1): 119-121.

Pevzner, P.A., Tang, H. & Tesler, G. De novo repeat classification and fragment assembly. *Genome Research*. 2004; 14 (9): 1786-1796.

Prjibelski, A.D., Vasilinetc, I., Bankevich, A., Gurevich, A., Krivosheeva, T., Nurk, S., Pham, S., Korobeynikov, A., Lapidus, A. & Pevzner, P.A. ExSPAnder: A universal repeat resolver for DNA fragment assembly. *Bioinformatics*. 2014; 30 (12): 293–301.

Pu. L., Lin, Y. & Pevzner P.A. Detection and analysis of ancient segmental duplications in mammalian genomes. *Genome Research.* 2018; 28 (6): 901-909.

Schmid, M., Frei, D., Patrignani, A., Schlapbach, R., Frey, J.E., Remus-Emsermann, M.N.P. & Ahrens, C.H. Pushing the limits of de novo genome assembly for complex prokaryotic genomes harboring very long, near identical repeats. *Nucleic Acids Research.* 2018; 46 (17): 8953-8965.

Sharon, I., Kertesz, M., Hug, L.A., Pushkarev, D., Blauwkamp, T.A., Castelle, C.J.,

Amirebrahimi, M., Thomas, B.C., Burstein, D., Tringe, S.G., Williams, K.H. & Banfield, J.F. Accurate, multi-kb reads resolve complex populations and detect rare microorganisms. *Genome Research*. 2015; 25 (4): 534-543.

Simpson, J.T., Workman, R.E., Zuzarte, P.C., David, M., Dursi, L.J. & Timp, W. Detecting DNA cytosine methylation using nanopore sequencing. *Nature Methods*. 2017; 14 (4): 407-410.

Tischler, G. Haplotype and Repeat Separation in Long Reads. Preprint at *bioRxiv*. 2017; doi: https://doi.org/10.1101/145474.

Treangen, T.J., Koren, S., Sommer, D.D., Liu, B., Astrovskaya, I., Ondov, B., Darling, A.E., Phillippy, A.M. & Pop, M. MetAMOS: a modular and open source metagenomic assembly and analysis pipeline. *Genome Biology*. 2013; 14 (1): R2.

Vyahhi, N., Pyshkin, A., Pham, S. & Pevzner, P.A. From de Bruijn graphs to rectangle graphs for genome assembly. *Lecture Notes in Computer Science*. 2012; 7534: 249-261.

Walker, B.J., Abeel, T., Shea, T., Priest, M., Abouelliel, A., Sakthikumar, S., Cuomo, C.A., Zeng, Q., Wortman, J., Young, S.K. & Earl, A.M. Pilon: an integrated tool for comprehensive microbial variant detection and genome assembly improvement. *PloS One*. 2014; 9 (11): e112963.

Weissensteiner, M.H., Pang, A.W.C., Bunikis, I., Höijer, I., Vinnere-Petterson, O., Suh, A. & Wolf, J.B.W. Combination of short-read, long-read, and optical mapping assemblies reveals large-scale tandem repeat arrays with population genetic implications. *Genome Research.* 2017; 27 (5): 697-708.

Wick R. R., Judd L. M., Gorrie C. L. & Holt K.E. Unicycler: Resolving bacterial genome assemblies from short and long sequencing reads. *PLoS Computational Biology*. 2017; 13 (6): e1005595.

Wick, R.R., Schultz, M.B., Zobel, J. & Holt, K.E. Bandage: interactive visualization of de novo genome assemblies. *Bioinformatics*. 2015; 31 (20): 3350-3352.

Zimin, A.V., Puiu, D., Luo, M.C., Zhu, T., Koren, S., Marçais, G., Yorke, J.A., Dvořák, J. & Salzberg, S.L. 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.* 2017; 27 (5): 787-792.

CHAPTER 3:

DiploidFlye: Haplotype Phasing of Long Read Assemblies Using Repeat Graphs

3.1 Abstract

With the recent advancements in long single molecule sequencing reads, genome assemblers have been able to produce high-quality assemblies for large, complex organisms such as humans. However, these assemblers often fail to account for the increased complexity of these diploid genomes, either ignoring or collapsing the differences between heterozygous sequences. To address this problem, we developed diploidFlye, an extension of the Flye assembler that detects heterozygous variations and generates haplotype-aware contigs called haplocontigs. DiploidFlye utilizes the Flye repeat graph to simplify and accelerate this process, phasing each unique edge of the graph into two haplocontigs. We show that diploidFlye can accurately phase a large fraction of the *Arabidopsis thaliana* genome, producing correct haplocontigs for these regions.

3.2 Introduction

Recent experimental and computational advances in generating and analyzing single molecule sequencing (SMS) reads have improved the contiguity of *de novo* genome assemblies (Mostovoy et al. 2016). Long read SMS technologies (like Pacific Biosciences and Oxford Nanopore) have enabled the assembly of large, complex genomes, primarily because the long reads can span difficult repeat regions in these genomes (Chin et al. 2013; Koren et al. 2017). Although SMS assemblers have a better ability to resolve the difficult repeats found in complex genomes, they often neglect another feature of these larger genomes, namely, their diploidy.

Most SMS assemblers fail to distinguish variations between the haplotypes of diploid genomes, usually generating mosaic contigs representing a mixture of haplotype alleles (Chaisson et al. 2015; Chin et al. 2016). Thus, the heterozygosity of these assemblies is unknown and variations in sequence, structure, and gene presence between homologous chromosomes are lost. This is a significant problem because haplotype information plays a crucial role in various areas such as linkage analysis, association studies, population genetics, and clinical genetics (Snyder et al. 2015; Brown et al. 2017). For example, if the haplotypes of a transplant donor's human leukocyte antigen (HLA) region closely matches those of the recipient, then the transplant usually has improved outcomes (Crawford et al. 2005).

We present diploidFlye, a haplotype-aware extension of the Flye assembler (Kolmogorov et al. in press) that generates haplocontigs (contigs whose sequence is derived from individual haplotypes) from the Flye assembly graph. DiploidFlye identifies variations between homologous chromosome sequences, separates reads into haplotypes based on these variations, and then constructs the sequence of the haplocontigs from these reads, repeating these steps in an iterative fashion. This procedure follows a similar approach to the Flye algorithm for resolving

unbridged repeats in the assembly graph laid out in Kolmogorov et al. (in press). DiploidFlye takes advantage of the repeat graph constructed by Flye to only focus on the unique (i.e. non-repetitive) edges in the graph, which greatly simplifies the problem. These edges form the lion's share of the assembly for most genomes and are much easier to phase than the difficult repetitive edges. Furthermore, diploidFlye is able to phase all of the unique edges of the assembly graph in parallel, greatly speeding up this procedure.

Currently, the FALCON assembler along with FALCON-Unzip, the associated haplotype-resolving tool, is the most widely used SMS assembler for generating haplotype-aware contigs (Chin et al. 2016). FALCON follows the string graph approach, first building a string graph from reads, separating reads into haplotypes based on bubbles in the string graph caused by variations between homologous chromosomes, and finally assembling primary contigs and alternative haplotigs (contigs containing alternative haplotype alleles). However, it is difficult to distinguish repetitive regions from homologous chromosomes in the string graph approach. The variations that FALCON uses to construct alternative haplotigs could in reality represent differences between long repeats in the genome that are from the same haplotype. In this case, FALCON may be confounding variations in haplotypes with variations in repeats and outputting alternative repeat sequences from the same haplotype rather than alternative haplotigs. In contrast, Flye follows the de Bruijn graph (DBG) approach, so it constructs the repeat graph (also known as the assembly graph) that distinguishes between repetitive and non-repetitive edges. Thus, diploidFlye is able to use the assembly graph to avoid confounding variations in haplotypes with variations in repeats. Furthermore, by using the assembly graph, diploidFlye is able to separate the phasing procedure into hundreds of smaller, independent runs, greatly simplifying and accelerating the process in comparison to FALCON.

3.3 Methods

Overview.

DiploidFlye considers all of the unique edges of the Flye assembly graph and attempts to phase each edge into two haplotypes in parallel. The process of phasing each unique edge occurs through several steps. First, tentative variant positions are identified based on the multiple alignment of all reads against the edge. Based on these tentative positions, regions with the highest divergence are found that act as anchor regions for constructing the two distinct haplocontigs for this edge. All reads that map to these anchor regions are assigned to one of the two haplotypes based on the variant positions, and then the prefix of each haplocontig sequence is constructed from the consensus of these reads.

Now begins an iterative procedure to extend the haplocontig prefixes to the right, where new reads are recruited to each haplotype based on the variant positions in the prefix, the haplocontig prefixes are extended based on the consensus of these new reads, and then more reads are recruited to the haplotypes based on the extended prefix sequences, etc. This iterative procedure terminates when the rest of the edge has been phased or when we fail to continue extending the haplocontig sequences to the right. The same procedure also extends the haplocontig prefixes to the left in parallel. After processing all unique edges in the assembly graph, diploidFlye outputs a pair of haplocontigs for each edge that was successfully phased. Additionally, diploidFlye also outputs the sequences of all haploid edges present in the repeat graph after Flye's repeat resolution procedures.

Identifying Heterozygous Positions.

Similar to the unbridged repeat resolution approach from the Flye assembler (Kolmogorov et al. in press), we identify heterozygous positions for each unique edge by first mapping reads to the edge and generating a multiple alignment of all reads that overlap with the unique edge. We proceed to calculate the substitution, deletion, and insertion rates of each column using the second most frequent nucleotide in the same way as Flye as well. If the mutation rate of a specific position exceeds certain thresholds (0.1, 0.2, and 0.3 for substitutions, deletions, and insertions, respectively), then that position is identified as a heterozygous position. If the rate of heterozygosity is too low (below 0.01%), then the edge is considered haploid and the algorithm halts for this edge.

Anchoring the Haplocontig Sequences.

Before diploidFlye can begin to construct the haplocontig sequences, it must find a high confidence anchor region to start from. Regions that have higher heterozygosity usually indicate a larger structural variation such as a long indel, from which reads can be recruited to different haplotypes with higher confidence. To locate these regions, we split the edge into 1 kb non-overlapping windows and simply find a window with the highest heterozygosity rate, which we call the distinguishing window. If the heterozygosity rate of the distinguishing window is below a certain threshold (1% by default), we have low confidence that we will be able to distinguish between the haplotype sequences using this window, so we simply stop and label the edge as unable to be phased.

Figure 3.1 illustrates how we recruit reads to different haplotypes for the distinguishing window. We take the segments of all reads that map to and span the entire window, and for each

read segment we calculate its distance from the consensus sequence: The match rate of a read segment aligned to the consensus is the number of matches in the alignment divided by the total length of the alignment for this window, and the distance is the complement of the match rate (one minus the match rate). Then we cluster the reads into two groups (one for each haplotype), using agglomerative clustering on the distances of each read. Finally, we construct an initial haplocontig prefix for each group using the consensus of all of the reads in each group.

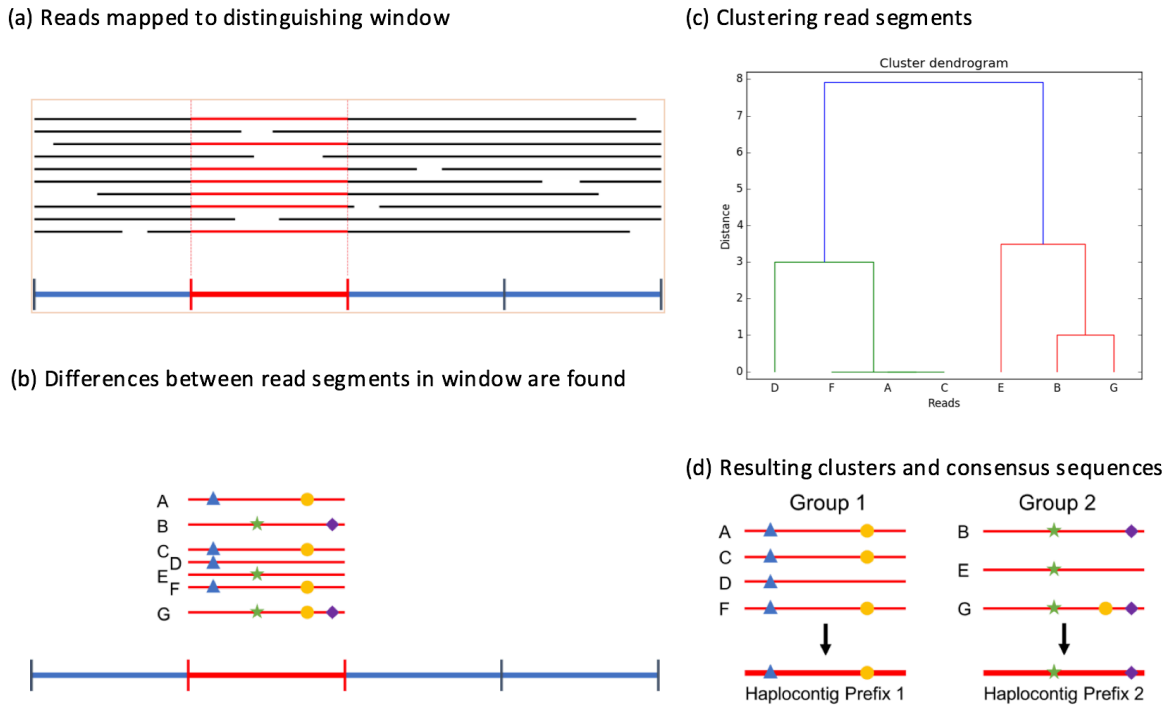

Figure 3.1: An overview of how diploidFlye anchors the haplocontig sequences. **(a)** The edge is split into 1 kb windows and the distinguishing window with highest divergence is identified (shown in red). Then segments of reads that map to and span the distinguishing window are extracted. These read segments will be used as anchors for the initial haplocontig sequences. **(b)** The distance between each read segment and the edge consensus sequence is calculated, here represented by different symbols on the read segments. **(c)** A sample dendrogram produced by running agglomerative clustering using the distances of the read segments. Two distinct clusters can be seen in the dendrogram (green and red). **(d)** The two resulting groups of reads represent different haplotype sequences. The consensus of the reads in each group is used as the initial haplocontig prefix for each haplotype. (Note: the entire length of the reads in each group is used to generate the haplocontig prefixes rather than only the read segments as shown here).

Iteratively Extending Haplocontigs.

Now that we have haplocontig prefix sequences, we can attempt to extend them to the right (extension to the left occurs analogously, in parallel). This process is similar to the Flye algorithm for "moving forward" into the repeat (Kolmogorov et al. in press). Since we will utilize the heterozygous positions identified earlier, we want to evaluate their reliability first. At each position, we simply consider the symbol (A, C, G, T, -) occurring at that position for each haplocontig prefix. If the two symbols match, then that position is rejected as homozygous, but if they disagree, then that position becomes a confirmed heterozygous position. Positions that are not yet covered by the haplocontig prefixes remain tentative heterozygous positions; we only use confirmed heterozygous positions for read assignment.

Next, we map all reads to the two haplocontig prefixes to construct a more accurate alignment. For each read, we compute its vote for each haplotype based on the number of confirmed heterozygous positions for which the symbol of the read agrees with the symbol of the haplocontig prefix (ignoring all other positions). The read is assigned to a haplotype if its vote for this haplotype is larger than its vote for the other haplotype by at least a minimum threshold (the default value is three), and it remains unassigned otherwise.

Afterwards, we use all reads that have been assigned to a haplotype to construct a new consensus haplocontig sequence. Since the newly assigned reads come from the right end of the haplocontig, constructing a new haplocontig sequence with these reads corresponds to extending the haplocontig prefix to the right. Note that we only construct haplocontigs up to where the coverage of reads is at least $minCoverage$ in both haplocontigs to ensure that the generated consensus sequences are accurate. Furthermore, both haplocontigs are then truncated to the length of the shortest haplocontig to ensure consistency in comparison during read assignment.

The process of confirming heterozygous positions, assigning reads, and then constructing haplocontig consensus sequences is then repeated in an iterative fashion to extend the haplocontig further and further to the right. The iteration ends when the haplocontig sequences cannot extend any further, either when the end of the edge has been reached or when there is a large gap before the next confirmed heterozygous position. If the ends of the edge are reached after extending both to the right and to the left, then we have successfully phased the edge. As long as the haplocontig is above a minimum length (the default value is 5 kb), then diploidFlye will output whatever haplocontigs have been constructed for the edge regardless of if it has been entirely phased.

Heterozygous Bulges Resolved by Flye.

When the heterozygosity rate is sufficiently high, sometimes heterozygous regions will be detected by the repeat graph and appear as bulges. If this happens, then diploid edges between bulges will appear to be repeats of multiplicity two (which we call simple repeats). In this case, Flye will perform its algorithms for resolving bridged repeats and simple unbridged repeats. If either of these two algorithms are successful, then the diploid edges will become haplotype-separated unique edges, which is exactly what diploidFlye is trying to produce. Thus, diploidFlye does not attempt to phase these unique edges and simply includes them in the set of haplocontigs that it outputs. There may also be unique edges present in the repeat graph that do not appear as simple repeats. Since these unique edges also correspond to haplotype-specific sequences, diploidFlye will detect and output them as haplocontigs as well.

3.4 Results

To ensure that we have a gold standard for validation, we use a set of *Arabidopsis thaliana* datasets presented in the FALCON paper (Chin et al. 2016). These datasets are sequenced from a plant trio: an inbred Col-0 parent (called the COL0 dataset), an inbred Cvi-0 parent (called the CVI0 dataset), and the F1 progeny resulting from a cross of these parents (called the F1 dataset). The total genome size is 135 Mb for each of these datasets, though the so-called "golden path" reference length is only 120 Mb (Arabidopsis Genome Initiative 2000). Pacific Biosciences reads with P4-C2 sequencing chemistry were generated from these genomes at coverages of 115×, 110× and 140× for COL0, CVI0, and F1, respectively. The reads from both the COL0 and CVI0 datasets had an average length of ~6 kb and an N50 of ~9 kb. The reads from the F1 dataset had an average length of ~11.5 kb and an N50 of ~17.5 kb (see Chin et al. 2016 for further details). Table 3.1 presents the results of running Flye on each of these datasets.

Table 3.1: Assembly statistics for Flye assemblies of the COL0, CVI0, and F1 datasets. The assembly quality was evaluated using the QUAST 5.0 tool (Mikheenko et al. 2018) with the TAIR10 genome (Lamesch et al. 2012) as a reference. 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, the corrected assembly is formed after breaking each erroneously assembled contig at its breakpoints resulting in shorter contigs. The NGA50 of an assembly is defined as the NG50 of its corrected assembly (Mikheenko et al. 2018). "Len" is the total length assembled, "#Mis" is the number of misassemblies, and "Reference coverage" is the percentage of the total reference length found in the assembly.

Dataset	Len (Mb)	#Contigs	NG50 (kb)	Reference Coverage	Reference % Identity	#Mis	NGA50 (kb)
COL0	117	472	6,051	97.0%	99.91%	251	1,818
CVI0	118	420	6,290	89.0%	98.81%	4977	67
F1	137	1246	437	96.8%	99.17%	3664	137

These datasets were compared to the TAIR10 Arabidopsis reference genome (Lamesch et al. 2012), which was assembled from the Col-0 strain. Thus, the differences in NGA50 values and the number of misassemblies illustrate that the COL0 and CVI0 datasets are highly divergent, rather than indicating problems with the assembly. We calculated the divergence rate by aligning the COL0 and CVI0 assemblies to each other and dividing the total error count (number of substitutions, insertions and deletions) by the sum of the match count and error count (note that a long insertion or deletion counts as a single error). Using this method, we found that there is a high 1.7% divergence rate between the genomes.

As expected, the quality of the F1 dataset exhibits intermediate values between the COL0 and CVI0 datasets since it contains sequences from both. However, the presence of two highly divergent strains in a single assembly also caused difficulties for the assembler, which is shown by the greater number of contigs and the lower NG50 value for the F1 assembly. This effect can be seen more clearly when comparing the repeat graphs constructed during the Flye assembly for these three datasets. Table 3.2 presents some graph statistics for these repeat graphs (the graphs are too large and tangled to display here), illustrating that the F1 repeat graph is significantly larger and more disconnected than the COL0 and CVI0 graphs, likely due to its high heterozygosity.

Table 3.2: Graph statistics for the COL0, CVI0, and F1 datasets.
Basic graph statistics are presented for the repeat graphs constructed by Flye when assembling the COL0, CVI0, and F1 datasets. The total number of nodes, unique edges, repetitive edges and connected components are shown for each dataset.

Dataset	# Nodes	# Unique Edges	# Repetitive Edges	# Connected Components
COL0	371	624	945	32
CVI0	321	476	939	26
F1	2003	2262	1409	343

The high heterozygosity of the F1 dataset causes complications for the application of diploidFlye. Highly heterozygous regions may appear as "bulges" in the repeat graph, made up of edges that have roughly half the expected coverage of unique edges (see Figure 3.2 for an example).

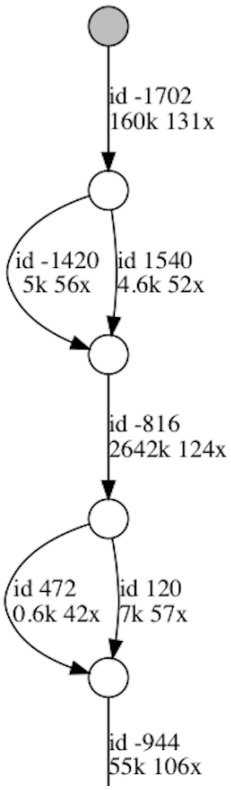

Figure 3.2: An example of two bulges forming a simple repeat in the F1 repeat graph. A simple repeat is formed in the graph (with ID -816) when two bulges appear in the graph. The average coverage of the dataset is around 120× so the bulges represent haploid edges formed by heterozygous variants. This simple repeat will be processed by Flye's algorithm for resolving simple unbridged repeats.

These edges correspond to haplotype-resolved sequences and should simply be reported by diploidFlye as haplocontigs. Furthermore, these bulges often produce simple unbridged repeats of multiplicity two and so are resolved by the unbridged repeat resolution algorithm in Flye. Flye reports 55 simple unbridged repeats for the F1 repeat graph, 47 of which were resolved successfully (as opposed to only 4 simple unbridged repeats appearing in the COL0 and CVI0 repeat graphs).

After both repeat resolution algorithms have finished, there exist 1104 unique edges in the F1 repeat graph (forward and reverse strands count as a single edge, and self-complementary

edges were omitted for simplicity). These edges sum to a total length of 122 Mb, 90% of the expected total length of the *Arabidopsis thaliana* genome (135 Mb). For the purposes of this dissertation, we will confine our attention to only 541 edges (with total length 66 Mb) which will be sufficient to demonstrate the efficacy of diploidFlye. Of these edges, 238 have low coverage corresponding to haploid edges, which accounts for 2.7 Mb of sequence (a simple threshold of 80× was used, equal to two-thirds of the average aligned coverage of 120×, based on an apparent separation in the distribution of edge coverages). DiploidFlye thus will not attempt to phase these edges, simply outputting them as haplocontigs.

DiploidFlye was run on the remaining 303 edges. For 89 of these edges, diploidFlye was either unable to find a distinguishing window with sufficient heterozygosity to cluster reads or the clustered reads had insufficient coverage to generate reliable haplocontig prefixes. Initial reads were clustered and haplocontig prefix sequences were generated for the remaining 214 edges. Figure 3.3 presents the cluster dendrogram of an edge where diploidFlye successfully clustered the 101 initial read segments from the distinguishing window into a group with 49 reads (green) and a group with 52 reads (red).

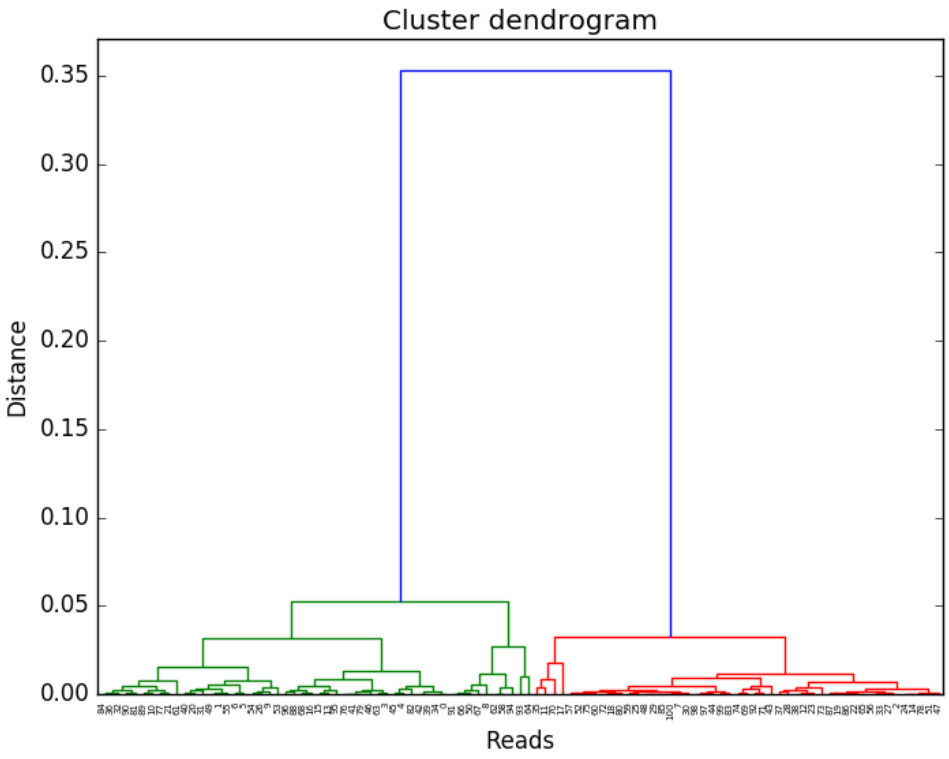

Figure 3.3: An example of a cluster dendrogram generated by performing agglomerative clustering on an edge in the F1 repeat graph.
This cluster dendrogram was generated while running agglomerative clustering on edge 1130 of the repeat graph from the F1 assembly. The 101 initial read segments are clustered into a group with 49 reads (green) and a group with 52 reads (red). The long length of the blue edges indicates that the two clusters (green and red) are well separated.

For these remaining 214 edges, diploidFlye attempted to iteratively extend the haplocontigs to the left and to the right by confirming heterozygous positions, assigning reads to haplotypes, and constructing consensus sequences as described in the Methods section. Figure 3.4 shows the lengths of all haplocontigs that diploidFlye generated relative to the total lengths of each edge (as reported by the Flye repeat graph). DiploidFlye produced haplocontigs of at least half the edge length for 186 edges (87% of the remaining edges), haplocontigs of at least 90% of the edge length for 151 edges (71%), and haplocontigs for the entire edge (≥99%) for

106 edges (50%). However, the haplocontigs of the longest edges failed to extend to the end of the edge (as shown on the right end of Figure 3.4), so the total haplocontig sequence length only constitutes 59% of the total edge sequence length.

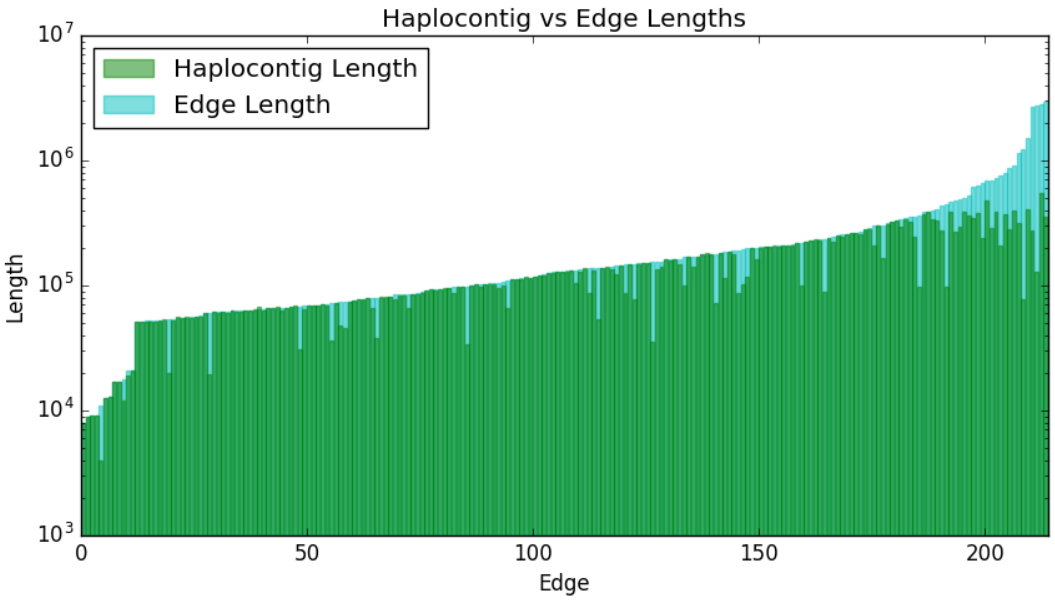

Figure 3.4: Haplocontig length versus total edge length.
A plot of the mean haplocontig lengths generated by diploidFlye and the total edge length for each of the 214 edges for which diploidFlye was able to produce haplocontig sequences. Note that the y-axis is on a log scale, indicating that the haplocontigs generated for the longest edges were significantly shorter than the edge length.

In order to evaluate the performance of diploidFlye, we utilized the Flye assemblies of the parental COL0 and CVI0 datasets. We refer to k-mers that are present in the COL0 assembly but absent in the CVI0 assembly as COL0 k-mers, and similarly, k-mers that occur in the CVI0 assembly but not in the COL0 one are CVI0 k-mers (we arbitrarily select $k = 25$ for this test). For each phased haplocontig generated by diploidFlye, we count the number of COL0 k-mers and the number of CVI0 k-mers.

Figure 3.5 presents a plot of the proportion of COL0 k-mers vs the proportion of CVI0 k-mers for every phased haplocontig. If the haplocontigs correctly correspond to specific haplotypes, then we expect the points to occur close to the x-axis or close to the y-axis. For the purposes of evaluation, a threshold of two-thirds was chosen to distinguish between haplocontigs that were correctly phased and those that were not (as shown by the dotted blue lines). Two haplocontigs are generated for each edge: If either of its haplocontigs falls below this threshold (lies between the dotted blue lines), then that edge is considered incorrectly phased and both of its edges are colored green in Figure 3.5. Based on these criteria, 193 edges (90%) were correctly phased by diploidFlye.

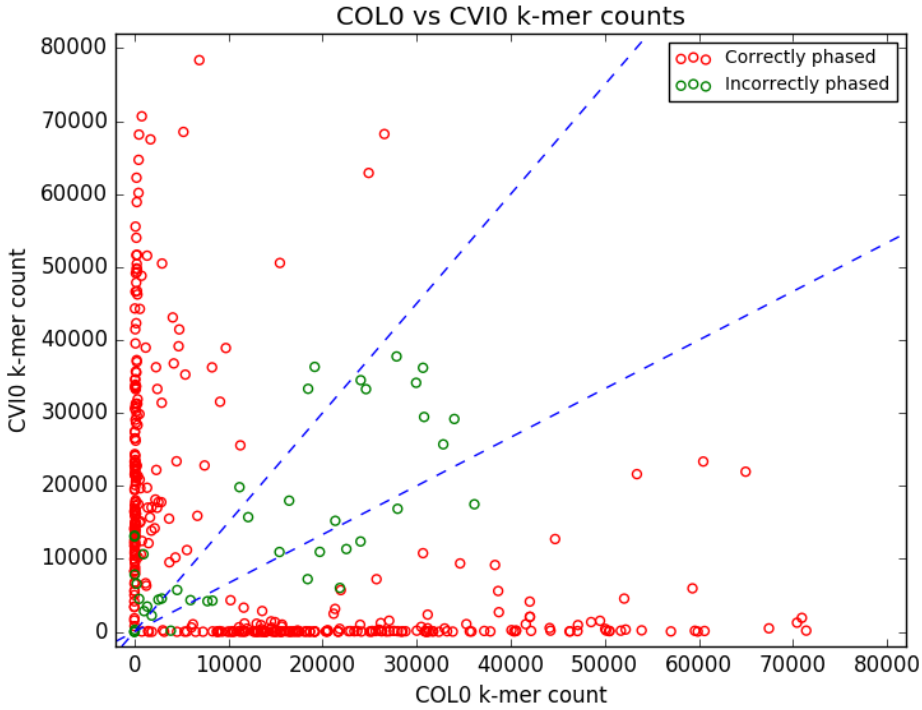

Figure 3.5: The counts of COL0 k-mers vs CVI0 k-mers for all haplocontigs with $k = 25$. A plot of the COL0 k-mers and the CVI0 k-mers for each haplocontig produced by diploidFlye with $k = 25$. Each point represents a single haplocontig, so each edge corresponds to two points, one for each of its haplocontigs. The dotted blue lines indicate the threshold (two-thirds) used to determine whether or not a haplocontig was correctly phased. Incorrectly phased haplocontigs that lie between the blue lines are colored green. Haplocontigs generated from the same edge as incorrectly phased haplocontigs are also colored green.

The correctly phased haplocontigs sum to a total length of 27.7 Mb, 59% of the total length of these 193 edges. The incorrectly phased haplocontigs sum to a total length of 3.0 Mb, 69% of the total length of the 21 edges they were generated from. Overall, combining the correctly phased haplocontigs with the haploid edges detected by diploidFlye, we obtained correct haplocontig sequences for 431 out of 541 edges considered (80%) with a combined length of 30.4 Mb out of 66 Mb (46%).

3.5 Discussion

This chapter presents a novel method for producing haplocontigs (i.e. haplotype-specific contigs) from the assembly of a genome. DiploidFlye relies on the Flye repeat graph to distinguish between unique and repetitive edges of the assembly to simplify the problem, and then it attempts to phase each unique edge by identifying heterozygous positions, assigning reads to haplotypes, and constructing haplocontig sequences in an iterative fashion.

It is important for diploidFlye to only consider unique edges in the Flye repeat graph so that it does not confound variations between repetitive regions of the genome with heterozygous positions. However, the classification of unique and repetitive edges in the repeat graph is a difficult problem due to high variance in coverage and imprecision in detecting overlaps between long reads. Furthermore, highly heterozygous regions may also split unique edges into pairs of separate haploid unique edges, causing some truly unique edges to be classified as repetitive. Flye will resolve many of these misclassified edges using its bridged and unbridged repeat resolution procedures (as seen in the 47 simple unbridged repeats in the F1 repeat graph resolved by Flye), but some may still be erroneously labeled repetitive. Currently, diploidFlye utilizes a makeshift strategy to distinguish between haploid and diploid unique edges based on coverage and heterozygosity rate. However, a comprehensive approach to standardize the classification of unique edges and distinguish between haploid and diploid edges would improve diploidFlye.

Currently, this chapter simply establishes that diploidFlye's novel approach was able to successfully phase 431 out of 541 edges, 80% of the edges that were considered, making up 30.4 Mb of the 66 Mb considered. If this proportion holds true for the rest of the F1 dataset, then we can expect 883 edges and 63 Mb of the genome to be phased by diploidFlye. In comparison,

FALCON is able to produce primary contigs of total length 140 Mb and alternative haplotigs with total length 105 Mb for the F1 dataset.

Thus far, diploidFlye is not yet able to produce haplocontigs at the same scale as FALCON. Although most of the edges considered were correctly phased, we still failed to produce haplocontigs for the majority of the sequence length considered because haploid edges tend to be short and the generated haplocontigs failed to span entire edges. The results could thus be improved in two ways: by generating correct haplocontigs for more edges and by extending the length of haplocontigs for correctly phased edges. The former can be addressed by modifying the criteria used to find distinguishing windows, cluster initial reads, and utilize heterozygous positions. The latter can be addressed by confirming more heterozygous positions, allowing a greater number of iterations, or modifying the stopping criteria when extending haplocontigs.

Additionally, diploidFlye currently takes a longer time to run than expected, mostly due to the bottleneck of iteratively polishing very long, unique edges (polishing is often the bottleneck of Flye assemblies). This can be addressed by further optimizing the speed of generating polished consensus sequences.

Furthermore, the range of heterozygosity rates at which diploidFlye can be applied must be further explored, especially its applicability to haplotyping the human genome. Nevertheless, diploidFlye is already able to phase 80% of the edges considered, producing haplocontigs for a significant fraction of the assembly. Therefore in its current state, diploidFlye already serves an important role as an extension to the Flye assembler by accounting for the heterozygosity of diploid genomes and producing suitable haplocontigs to improve their assemblies.

3.6 Acknowledgements

We would like to thank Misha Kolmogorov, Anton Bankevich, and Andrey Bzikadze for the many fruitful discussions we had related to this material.

Chapter 3, in full, is currently being prepared for submission for publication as "DiploidFlye: haplotype phasing of long read assemblies using repeat graphs" by Jeffrey Yuan and Pavel A. Pevzner. The dissertation author is a primary author of this material.

3.7 References

Arabidopsis Genome Initiative. Analysis of the genome sequence of the flowering plant Arabidopsis thaliana. *Nature*. 2000; 408 (6814): 796-815.

Brown, R., Kichaev, G., Mancuso, N., Boocock, J. & Pasaniuc, B. Enhanced methods to detect haplotypic effects on gene expression. *Bioinformatics*. 2017; 33 (15): 2307-2313.

Chaisson, M.J., Wilson, R.K. & Eichler, E.E. Genetic variation and the de novo assembly of human genomes. *Nature Reviews Genetics*. 2015; 16 (11): 627-40.

Chin, C.S., Alexander, D.H., Marks, P., Klammer, A.A., Drake, J., Heiner, C., Clum, A., Copeland, A., Huddleston, J., Eichler, E.E., Turner, S.W. & Korlach, J. Nonhybrid, finished microbial genome assemblies from long-read SMRT sequencing data. *Nature Methods*. 2013; 10 (6): 563-569.

Chin, C.S., Peluso, P., Sedlazeck, F.J., Nattestad, M., Concepcion, G.T., Clum, A., Dunn, C., O'Malley, R., Figueroa-Balderas, R, Morales-Cruz, A., Cramer, G.R., Delledonne, M., Luo, C., Ecker, J.R., Cantu, D., Rank, D.R., & Schatz, M.C. Phased diploid genome assembly with single molecule real-time sequencing. *Nature Methods*. 2016; 13 (12): 1050-1054.

Crawford, D.C. & Nickerson, D.A. Definition and clinical importance of haplotypes. *Annual Review of Medicine*. 2005; 56: 303-20.

Kolmogorov, M., Yuan, J., Lin, Y. & Pevzner, P.A. Assembly of long error-prone reads using repeat graphs. *Nature Biotechnology*. In press.

Koren, S., Walenz, B.P., Berlin, K., Miller, J.R., Bergman, N.H. & Phillippy, A.M. Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation. *Genome Research.* 2017; 27 (5): 722-736.

Lamesch, P., Berardini, T.Z., Li, D., Swarbreck, D., Wilks, C., Sasidharan, R., Muller, R., Dreher, K., Alexander, D.L., Garcia-Hernandez, M., Karthikeyan, A.S., Lee, C.H., Nelson, W.D., Ploetz, L., Singh, S., Wensel, A. & Huala, E. The Arabidopsis Information Resource (TAIR): improved gene annotation and new tools. *Nucleic Acids Research*. 2012; 40 (Database issue): D1202-10.

Mikheenko A., Prjibelski A., Saveliev V., Antipov D. & Gurevich A. Versatile genome assembly evaluation with QUAST-LG. *Bioinformatics*. 2018; 34 (13): i142-i150.

Mostovoy, Y., Levy-Sakin, M., Lam, J., Lam, E.T., Hastie, A.R., Marks, P., Lee, J., Chu, C., Lin, C., Džakula, Ž., Cao, H., Schlebusch, S.A., Giorda, K., Schnall-Levin, M., Wall, J.D. & Kwok, P.Y. A hybrid approach for de novo human genome sequence assembly and phasing. *Nature Methods*. 2016; 13 (7): 587-90.

Snyder, M.W., Adey, A., Kitzman, J.O. & Shendure, J. Haplotype-resolved genome sequencing: experimental methods and applications. *Nature Reviews Genetics*. 2015; 16 (6): 344-58.

CONCLUSION

De novo genome assemblies have greatly improved with the development of long single molecule sequencing (SMS) reads and other technologies such as 10X genomics, BioNano, and Hi-C sequencing techniques (Li et al. 2017; Mostovoy et al. 2016). Larger and more complex genomes can now be assembled almost at the resolution of entire chromosomes (Pendleton et al. 2015). However, there are still two main challenges hindering the complete assembly of complex genomes: repetitive regions and variations between similar sequences, such as variations between different instances of a repeat or between parental haplotypes (Chaisson et al. 2015). This dissertation has focused on developing new methods to address both of these challenges.

First of all, we limited our attention to only long SMS reads because they greatly improve the scale of the repeats that are resolved in the course of assembly. Although repeats that are longer than the average length of a read still remain unresolved, a large proportion of them that are shorter than 1-2 kb are easily resolved by simply using SMS reads. Next, we developed a de Bruijn graph (DBG) assembler for SMS reads because the DBG approach naturally reveals the repetitive regions in the course of assembly, whereas the overlap-layout-consensus approach does not. Thus, we developed ABruijn, which modifies the DBG approach by building an A-Bruijn graph from only frequent *k*-mers in the reads rather than all *k*-mers as in the DBG. We showed that ABruijn performs especially well on difficult repetitive genomes such as *Xanthomonas oryzae* compared to other OLC assemblers. However, complex repetitive regions especially in large genomes also turned out to be a major computational bottleneck for ABruijn, limiting its scalability, leading us to develop Flye.

As described in Chapter 2, Flye avoids the difficulty of considering all possible paths through repetitive regions by greedily choosing a path and generating disjointigs. Fortunately,

these disjointigs can be used to construct an accurate repeat graph, which can then be used to finish the assembly process and produce contigs. The remarkable thing about the repeat graph is that it provides a visualization of all remaining unresolved repeats in the genome, laying out everything that needs to be done to solve the first challenge of resolving long repetitive regions. This visualization is incredibly useful not only for understanding the current state of the assembly but also for determining the best method for proceeding to "finish" the assembly. Flye then proceeds to exploit the repeat graph to improve the assembly as much as possible, first resolving any bridged repeats using bridging reads, and then resolving unbridged simple repeats using variations between repeats. Using these methods, Flye is able to produce more contiguous assemblies of the human genome than other state-of-the-art assemblers such as Canu and MaSuRCA. To further improve the assembly, the repeat graph generated by Flye can be used to target the remaining unresolved repeats using other technologies such as 10X genomics, BioNano and Hi-C (Li et al. 2017).

The second challenge hindering genome assembly is variations between similar sequences. These variations include differences between instances of the same repeat (e.g. polymorphisms in segmental duplications), differences in how many times a sequence is repeated (the copy number), and differences between haplotypes of diploid or polyploid organisms (heterozygosity) (Chaisson et al. 2015). Due to the similarity of the sequences, these variations are often undetected or collapsed into consensus sequences that do not represent any single genomic sequence, and thus pose a major source of (often undetected) error in genome assembly (Chaisson et al. 2015). Flye attempts to address this issue in the case of simple unbridged repeats of multiplicity two. The Flye approach for resolving these unbridged repeats represents the first method for using the variation between repeats to resolve long repeats, and it

also generates a distinct sequence for each instance of the repeat. Thus, if Flye is able to resolve unbridged repeats, it produces output sequences that are sensitive to the small variations between repeat copies, addressing this challenge, albeit only for this limited case.

In Chapter 3, we discussed how diploidFlye also attempts to address the challenge of variations between similar sequences by producing haplocontigs that are sensitive to the differences between the parental haplotypes. To this end, diploidFlye utilizes the Flye repeat graph to simplify and parallelize the problem of phasing haplotypes. The unique edges in the graph represent regions where variation in the sequence must be due to heterozygosity and thus can easily be phased, so diploidFlye focuses on these regions. Of course, diploidFlye only addresses the problem for these relatively simple regions. Further work is required to deal with the more difficult case of phasing repetitive regions, in which case the variations due to heterozygosity must be distinguished from the variations between repeat copies.

Despite the advances made by ABruijn, Flye, and diploidFlye, there are still many obstacles to overcome regarding repetitive regions and variations between similar sequences. Very long, identical repeats still cannot be resolved by Flye and must be spanned by technologies that produce longer reads. More sophisticated algorithms must be developed to resolve and reconstruct more complex repetitive regions such as long tandem repeats and mosaic repeats of higher multiplicity; these reconstructions must be sensitive to variations between repeat copies and copy number. New algorithms are also needed to address these difficulties for diploid or other polyploid organisms such as plants. Thus, despite the progress made by recent developments in sequencing technologies and assembly algorithms, there is still a long way to go before entire finished genomes can be assembled *de novo*, but each technological advance and every new algorithm brings us a little closer to that goal.

References

Chaisson, M.J., Wilson, R.K. & Eichler, E.E. Genetic variation and the de novo assembly of human genomes. *Nature Reviews Genetics*. 2015; 16 (11): 627-40.

Li, C., Lin, F., An, D., Wang, W. & Huang, R. Genome sequencing and assembly by long reads in plants. *Genes*. 2018; 9 (1): 6.

Mostovoy, Y., Levy-Sakin, M., Lam, J., Lam, E.T., Hastie, A.R., Marks, P., Lee, J., Chu, C., Lin, C., Džakula, Ž., Cao, H., Schlebusch, S.A., Giorda, K., Schnall-Levin, M., Wall, J.D. & Kwok, P.Y. A hybrid approach for de novo human genome sequence assembly and phasing. *Nature Methods*. 2016; 13 (7): 587-90.

Pendleton, M., Sebra, R., Pang, A.W., Ummat, A., Franzen, O., Rausch, T., Stütz, A.M., Stedman, W., Anantharaman, T., Hastie, A., Dai, H., Fritz, M.H., Cao, H., Cohain, A., Deikus, G., Durrett, R.E., Blanchard, S.C., Altman, R., Chin, C.S., Guo, Y., Paxinos, E.E., Korbel, J.O., Darnell, R.B., McCombie, W.R., Kwok, P.Y., Mason, C.E., Schadt, E.E. & Bashir, A. Assembly and diploid architecture of an individual human genome via single-molecule technologies. *Nature Methods*. 2015; 12 (8): 780-6.