# Data-efficient De novo Genome Assembly Algorithm : Theory and Practice

*Ka Kit Lam*

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2016-43
http://www.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-43.html

May 9, 2016

**Data-efficient De novo Genome Assembly Algorithm : Theory and Practice**

by

Ka Kit Lam

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

in

Engineering – Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Satish Rao, Chair
Professor Lior Pachter
Professor Allan Sly

Spring 2016

**Data-efficient De novo Genome Assembly Algorithm : Theory and Practice**

## Abstract

Data-efficient De novo Genome Assembly Algorithm : Theory and Practice

by

Ka Kit Lam

Doctor of Philosophy in Engineering – Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Satish Rao, Chair

We study data-efficient and also practical de-novo genome assembly algorithm. Due to the advancement in high-throughput sequencing, the cost of read data has gone down rapidly in recent years. Thus, there is a growing need to do genome assembly in a cost effective manner on a large scale. However, the state-of-the-art assemblers are not designed to be data-efficient. This leads to wastage of read data, which translates to a significant monetary loss for large scale sequencing projects. Moreover, as technology advances, the nature of the read data also evolves. This makes the old assemblers insufficient for discovering all the information behind the data. Therefore, there is a need to re-invent genome assemblers to suit the growing demand.

In this dissertation, we address the data efficiency aspect of genome assembly as follows.

First, we show that even when there is noise in the reads, one can successfully reconstruct the genomes with information requirements close to the noiseless fundamental limit. We develop a new assembly algorithm, X-phased Multibridging, is designed based on a probabilistic model of the genome. We show, through analysis that it performs well on the model, and through simulation that it performs well on real genomes.

Second, we introduce FinisherSC, a repeat-aware and scalable tool for upgrading de-novo haploid genome assembly using long and noisy reads. Experiments with real data suggest that FinisherSC can provide longer and higher quality contigs than existing tools while maintaining high concordance. Thus, FinisherSC achieves higher data efficiency than state-of-the-art haploid genome assembly pipelines.

Third, we study whether post-processing metagenomic assemblies with the original input long reads can result in quality improvement. Previous approaches have focused on pre-processing reads and optimizing assemblers. We introduce BIGMAC, which takes an alternative perspective to focus on the post-processing step. Using both the assembled contigs and original long reads as input, BIGMAC first breaks the contigs at potentially misassembled locations and subsequently scaffolds contigs. Our experiments on metagenomes assembled from long reads show that BIGMAC can improve assembly quality by reducing

the number of mis-assemblies while maintaining/increasing N50 and N75. Thus, BIGMAC achieves higher data efficiency than state-of-the-art metagenomic assembly pipelines.

Moreover, we also discuss some theoretical work regarding genome assembly in terms of data, time and space efficiency; and a promising post-processing tool POSTME in improving metagenomic assembly using both short and long reads.

Blessed to be a blessing

# Contents

# Chapter 1

# Introduction

Determining the ordering of nucleotides in genomes is a fundamental problem in bioinformatics. It is still beyond the reach of current technology to read the whole genome from beginning to end for species of moderate genomic complexity. However, in DNA sequencing, biologists can obtain fragmented snapshots of the genome under investigation, which are called reads. Moreover, due to the advancement in high throughput sequencing, one can obtain many such reads at a reasonable cost. While it is a promising method to determine the genome, it also comes with its unique challenges. Since we do not know the genomic locations from where the reads are extracted, one has to rearrange the reads computationally to determine the genomes. A de novo assembler pieces together reads to form the underlying genome. Since this is an algorithm design problem with a strong connection to practical application, both the design and experimentation are critical components. Let us first review previous works on the design part.

One approach is to find the shortest superstring for the reads [4]. A basic greedy algorithm of successively merging reads of largest overlap length can yield an approximation ratio of 4. Subsequent work has reduced the approximation ratio to 2.5 [22]. Another approach is to use string graph [42, 41]. Under this approach, reads are treated as nodes on a graph and an edge is defined when the corresponding reads have significant overlap. The output genomes (or contigs) thus correspond to tours on the graph subject to certain optimality criterion (e.g. minimum cost network flow [42]). To handle complex repeats using short reads, De Bruijn graph approach is studied [50]. Under this approach, a read is broken into multiple K-mers that are consecutive substrings of length K. The graph considered consists of K-mers as nodes. An edge is defined when two K-mers are consecutive in some reads. The goal is to find an Eulerean trail on the graph. Under some idealized setting, this approach is shown to be optimal[49]. It is noteworthy that de novo assembly problem can also be cast as a maximum likelihood estimation problem[34]. Although this approach theoretically yields an optimal solution, various heuristics are normally employed to handle the intractable computational complexity [34].

Following the algorithm engineering paradigm[55], the next step in the study of de novo assembly algorithm is experimentation. There are various implementations of de novo assem-

blers [42, 63, 47, 50, 1]. However, benchmarking shows that no single assembler dominates because different approaches are competitive in different aspects[6, 54]. This should not be a big surprise because there are multiple dimensions of optimization involved. In particular, one has to simultaneously consider repeats, noise, abundance information, space complexity and time complexity. Frequently, a specific type of data favors assemblers with strength in specific dimensions. Nonetheless, these experiments confirm that the nature of the data is important in the design of de novo assemblers. Therefore, with third generation sequencing platforms (e.g. long read technology) emerging [40], it is natural to anticipate that there are plenty of opportunities to develop optimized de novo assemblers.

In the last few years, advancement in sequencing technology has led to a significant increase in the length of reads [14, 21]. It is promising progress because longer reads help resolve complex repetitive regions in genomes. Some notable applications of these long reads include providing new insights into human genomes by closing interstitial gaps [10] and routinely automating genome finishing for bacterial genomes [12]. Despite having great potential, long read technology poses new computational challenges for de novo assembly. Since single molecule of DNA is sequenced, the reads obtained are mostly corrupted by insertion and deletion noise with a noise level around 15% [14, 21]. Moreover, the throughput of current long read sequencers is generally lower than that of the short read sequencers. Thus, it is common for biologists to generate reads from multiple machines and subsequently seek computational approaches to combine them.

There are three mainstream approaches for de novo assembly involving long reads. In hybrid assembly approach, one uses short reads to error-correct long reads and subsequently assembles the error-corrected long reads[25]. In non-hybrid assembly approach, one error-corrects long reads by long reads themselves and subsequently assembles the error-corrected long reads [12]. In scaffolding approach, one uses long reads to scaffold contigs formed from short reads [16, 5]. These three approaches follow the overlap-layout-consensus paradigm. Due to high noise rate, significant efforts have been made toward improving the computational efficiency for the overlap stage [9, 3, 43]. While this computational bottleneck is gaining better understanding and resolution, a natural next step is to improve data efficiency of de novo assemblers. Improving data efficiency in de novo assembly can be understood as improving assembly quality using the same data available. Data efficiency is especially relevant because tradeoff is made in the overlap computation stage to balance computational speed and accuracy. However, it is not clear how one can achieve quality improvement, nor is it clear how far we are from optimal performance. Thus, addressing the de novo assembly problem in terms of data efficiency does not only shed light on building better software but also save human resources on indefinitely optimizing assemblers.

Shannon studies a mathematical theory of communication, which has a profound impact in the age of digital revolution [57]. It is thus advantageous to mimic the work of Shannon to study other data-related engineering problems, in particular, de novo assembly. Study of data efficiency for de novo assembly problem can be traced back to the time when the notion of Lander-Waterman coverage is introduced [32]. Subsequent work discovers the theoretical significance of Eulerian trail in perfect reconstruction[49] . An information theoretic for-

malization and tight characterization are later studied [37, 7, 39]. For example, it is shown that in i.i.d. genome model, noise is irrelevant for genome assembly[39]. It is also shown that close to optimal assembly is possible for a range of bacterial genomes in a noiseless setting [7]. However, there are gaps in the study of data-efficient algorithms for long read data. For example, it is not clear about the role of noise toward data efficiency beyond the i.i.d. genome model. Moreover, it is also not clear how well the theoretical insights can be translated into quality improvement in de novo assembly for long read data in practice. Thus in the literature, there is a gap between the theory and the practice in the study of data-efficient algorithms for long read data.

In this dissertation, we claim that it is feasible to construct practical and more data-efficient algorithm for de novo assembly for long read data. To establish the thesis, there are two main pillars. On the theoretical part, we use an information theoretic framework to guide the design of data-efficient algorithm to handle noise and to be computationally efficient. On the practical part, we use post-processing as a technique to bring theoretical insights into assembler building process for long read data, which would otherwise involve significant engineering efforts.

In theory, we introduce two algorithms: X-phased Multibridging [29] and OnlineOfflineAlgo [31]. We show that even when there is noise in the reads, one can successfully reconstruct the genomes with information requirements close to the noiseless fundamental limit. A new assembly algorithm, X-phased Multibridging, is designed based on a probabilistic model of the genome. We show, through analysis that it performs well on the model, and through simulations that it performs well on real genomes. We also study computational complexity of a de novo assembly algorithm: OnlineOfflineAlgo. For noiseless reads of length at least twice the minimum length, OnlineOfflineAlgo requires minimum coverage, is time-efficient and is space-optimal under the i.i.d. genome model. The key idea to achieve space and time efficiency is to break the procedure into two phases, an online and an offline phase. The theoretical study has provided several algorithmic insights. X-phased Multibridging demonstrates that repeat-aware overlap rule and phasing of polymorphisms within repeat interiors are effective measures to combat noise in reads. OnlineOfflineAlgo demonstrates that a two-phase process can effectively save computational demand for de novo assembly. To transfer algorithmic insights into practical assemblers, one can build assemblers from scratch. However, this normally requires significant engineering effort. This is not ideal in such a fast moving field. Thus, we seek a more agile approach. In particular, we use a post-processing approach: post-processing assembled contigs with original data.

In practice, we introduce three post-processing tools: FinisherSC [30], BIGMAC [28] and POSTME [27]. We introduce FinisherSC, a repeat-aware and scalable tool for upgrading de-novo haploid genome assembly using long and noisy reads. Experiments with real data suggest that FinisherSC can provide longer and higher quality contigs than existing tools while maintaining high concordance. Thus, FinisherSC achieves higher data efficiency than state-of-the-art haploid genome assembly pipelines. We study whether post-processing metagenomic assemblies with the original input long reads can result in quality improvement. Previous approaches have focused on pre-processing reads and optimizing assemblers. We

introduce BIGMAC, which takes an alternative perspective to focus on the post-processing step. Using both the assembled contigs and original long reads as input, BIGMAC first breaks the contigs at potentially mis-assembled locations and subsequently scaffolds contigs. Our experiments on metagenomes assembled from long reads show that BIGMAC can improve assembly quality by reducing the number of mis-assemblies while maintaining/increasing N50 and N75. Thus, BIGMAC achieves higher data efficiency than state-of-the-art metagenomic assembly pipelines. Finally we study POSTME, a metagenomic assembly post-processor using hybrid data. It shows promising results when compared to Spades-Hybrid, which is a leading software in the field.

As history suggested in the past decade, high throughput sequencing is an actively evolving field. We expect that new sequencing platforms will continue to push the boundary of our understanding on the genomes of living things. One notable application area is metagenomics. However, the problem of metagenomic assembly has long been considered as a difficult computational challenge [11]. Although short read metagenomic assemblers have appeared in the last few years [48, 44], the problem is still wide open [59]. On the other hand, promising development of long reads has raised optimism for the plausibility in cracking the notorious metagenomic assembly problem [24]. With more informative data and more sophisticated assembler design methods, we are better positioned than ever to advance the field. This dissertation can serve as one of the starting points for these exciting research directions.

# Chapter 2

# Near-optimal assembly for shotgun sequencing with noisy reads

Recent work identified the fundamental limits on the information requirements in terms of read length and coverage depth required for successful *de novo* genome reconstruction from shotgun sequencing data, based on the idealistic assumption of no errors in the reads (noiseless reads). In this work, we show that even when there is noise in the reads, one can successfully reconstruct with information requirements close to the noiseless fundamental limit. A new assembly algorithm, X-phased Multibridging, is designed based on a probabilistic model of the genome. It is shown through analysis to perform well on the model, and through simulations to perform well on real genomes.

## 2.1 Background

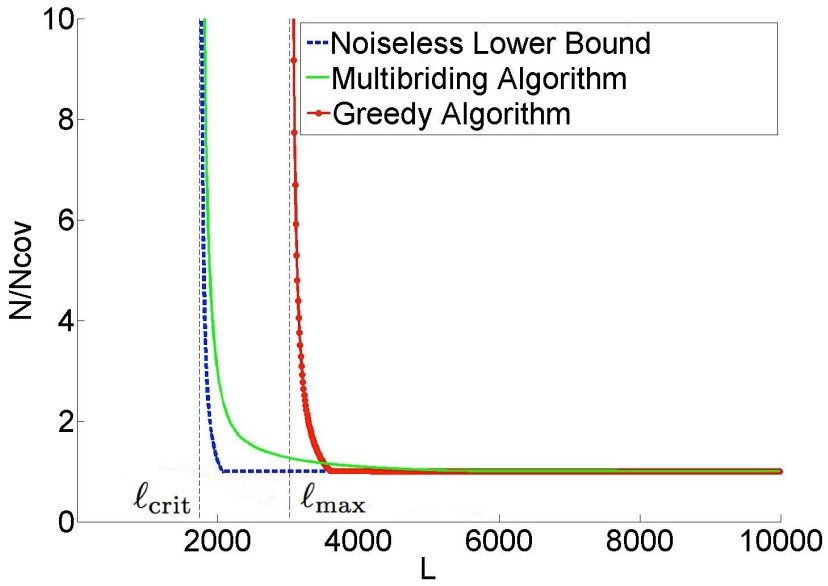
Optimality in the acquisition and processing of DNA sequence data represents a serious technology challenge from various perspectives including sample preparation, instrumentation and algorithm development. Despite scientific achievements such as the sequencing of the human genome and ambitious plans for the future [60, 56], there is no single, overarching framework to identify the fundamental limits in terms of information requirements required for successful output of the genome from the sequence data.

Information theory has been successful in providing the foundation for such a framework in digital communication [57], and we believe that it can also provide insights into understanding the essential aspects of DNA sequencing. A first step in this direction has been taken in the recent work [7], where the fundamental limits on the minimum read length and coverage depth required for successful assembly are identified in terms of the statistics of various repeat patterns in the genome. Successful assembly is defined as the reconstruction of the underlying genome, i.e. genome finishing [51]. The genome finishing problem is particularly attractive for analysis because it is clearly and unambiguously defined and is arguably the ultimate goal in assembly. There is also a scientific need for finished genomes [36][35].

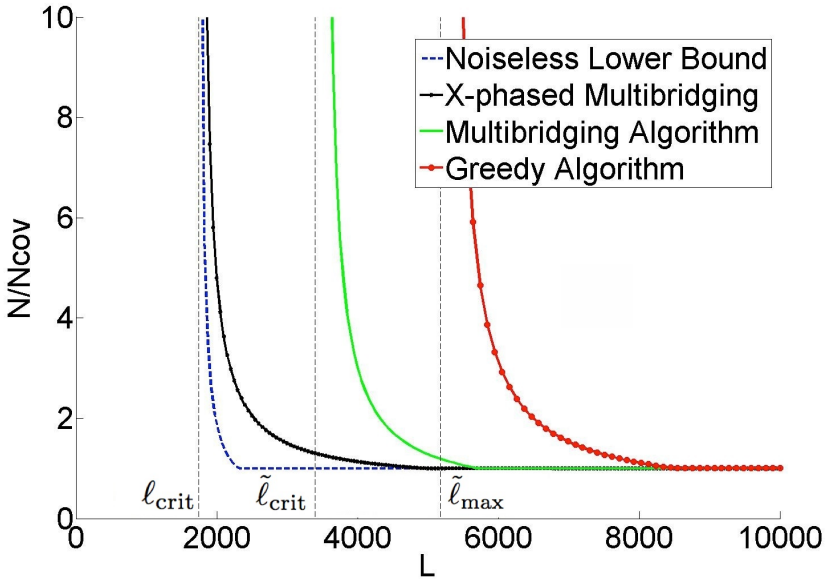
Until recently, automated genome finishing was beyond reach [18] in all but the simplest of genomes. New advances using ultra-long read single-molecule sequencing, however, have reported successful automated finishing [12, 25]. Even in the case where finished assembly is not possible, the results in [7] provide insights on optimal use of read information since the heart of the problem lies in how one can optimally use the read information to resolve repeats.

Figure 2.1a gives an example result for the repeat statistics of *E. coli* K12. The x-axis of the plot is the read length and the y-axis is the coverage depth normalized by the Lander-Waterman depth (number of reads needed to cover the genome [32]). The lower bound identifies the necessary read length and coverage depth required for *a*ny assembly algorithm to be successful with these repeat statistics. An assembly algorithm called Multibridging Algorithm was presented, whose read length and coverage depth requirements are very close to the lower bound, thus tightly characterizing the fundamental information requirements. The result shows a critical phenomenon at a certain read length $L = \ell_{crit}$: below this critical read length, reconstruction is impossible no matter how high the coverage depth; slightly above this read length, reconstruction is possible with Lander-Waterman coverage depth. This critical read length is given by $\ell_{crit} = \max\{\ell_{int}, \ell_{tri}\}$, where $\ell_{int}$ is the length of the longest pair of exact interleaved repeats and $\ell_{tri}$ is the length of the longest exact triple repeat in the genome, and has its roots in earlier work by Ukkonen on Sequencing-by-Hybridization [61]. The framework also allows the analysis of specific algorithms and the comparison with the fundamental limit; the plot shows for example the performance of the Greedy Algorithm and we see that its information requirement is far from the fundamental limit.

A key simplifying assumption in [7] is that there are no errors in the reads (noiseless reads). However reads are noisy in all present-day sequencing technologies, ranging from primarily substitution errors in Illumina® platforms, to primarily insertion-deletion errors in Ion Torrent® and PacBio® platforms. The following question is the focus of the current paper: in the presence of read noise, can we still successfully assemble with a read length and coverage depth close to the minimum in the noiseless case? A recent work [23] with an existing assembler suggests that the information requirement for genome finishing substantially exceeds the noiseless limit. However, it is not obvious whether the limitations lie in the fundamental effect of read noise or in the sub-optimality of the algorithms in the assembly pipeline.




(a) Information requirement for noiseless reads




(b) Information requirement for noisy reads

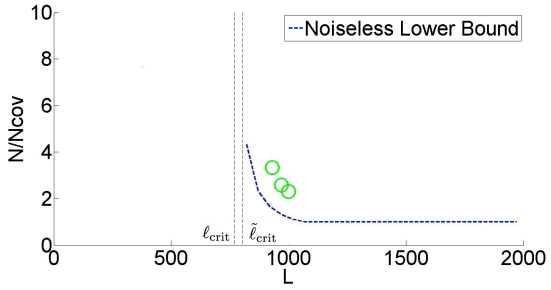
Figure 2.1: Information requirement to reconstruct *E. coli* K12. $\ell_{crit} = 1744, \tilde{\ell}_{crit} = 3393$

## 2.2 Results

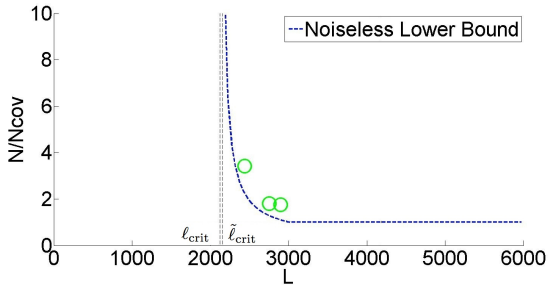
The difficulty of the assembly problem depends crucially on the genome repeat statistics. Our approach to answering the question of the fundamental effect of read noise is based on design and analysis using a parametric probabilistic model of the genome that matches the key features of the repeat statistics we observe in genomes. In particular, it models the presence of long flanked repeats which are repeats flanked by statistically uncorrelated region. Figure 2.1b shows a plot of the predicted information requirement for reliable reconstruction by various algorithms under a substitution error rate of 1%. The plot is based on analytical formulas derived under our genome model with parameters set to match the statistics of *E. coli* K12. We show that it is possible in many cases to develop algorithms that approach the noiseless lower bound even when the reads are noisy. Specifically, the X-phased Multibridging Algorithm has close to the same critical read length $L = \ell_{crit}$ as in the noiseless case and only slightly greater coverage depth requirement for read lengths greater than the critical read length.

We then proceed to build a prototype assembler based on the analytical insights and we perform experiments on real genomes. As shown in Figure 2.2, we test the prototype assembler by using it to assemble noisy reads sampled from 4 different genomes. At coverage and read length indicated by a green circle, we successfully assemble noisy reads into one contig (in most cases with more than 99% of the content matched when compared with the ground truth). Note that the information requirement is close to the noiseless lower bound. Moreover, the algorithm (X-phased Multibridging) is computationally efficient with the most computational expensive step being the computation of overlap of reads/K-mers, which is an unavoidable procedure in most assembly algorithms.

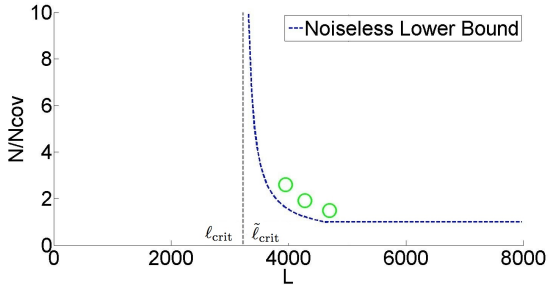
The main conclusion of this work is that, with an appropriately designed assembly algorithm, the information requirement for genome assembly is surprisingly insensitive to read noise. The basic reason is that the redundancy required by the Lander-Waterman coverage constraint can be used to denoise the data. This is consistent with the asymptotic result obtained in [39] and the practical approach taken in [12]. However, the result in [39] is based on a very simplistic i.i.d. random genome model, while the model and genomes considered in the present paper both have long repeats. A natural extension of the Multibridging Algorithm in [7] to handle noisy reads allows the resolution of these long flanked repeats if the reads are long enough to span them, thus allowing reconstruction provided that the read length is greater than $L = \tilde{\ell}_{crit} = \max\{\tilde{\ell}_{int}, \tilde{\ell}_{tri}\}$, where $\tilde{\ell}_{int}$ is the length of the longest pair of flanked interleaved repeats and $\tilde{\ell}_{tri}$ is the length of the longest flanked triple repeat in the genome. This condition is shown as a vertical asymptote of the "Multibridging Algorithm" curve in Figure 2.1b. By exploiting the redundancy in the read coverage to resolve read errors, the X-phased Multibridging can phase the polymorphism across the flanked repeat copies using only reads that span the exact repeats. Hence, reconstruction is achievable with a read length close to $L = \ell_{crit}$, which is the noiseless limit.



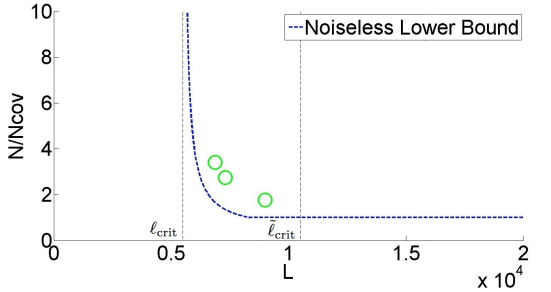

(a) *Prochlorococcus marinus*




(b) *Helicobacter pylori*




(c) *Methanococcus maripaludis*




(d) *Mycoplasma agalactiae*

Figure 2.2: Simulation results on a prototype assembler (substitution noise of rate 1.5 %)

### Related work

All assemblers must somehow address the problem of resolving noise in the reads during genome reconstruction. However, the traditional approaches to measuring assembly performance makes quantitative comparisons challenging for unfinished genomes [45]. In most cases, the heart of the assembly problem lies in processing of the assembly graph, as in [63, 17, 58]. A common strategy for dealing with ambiguity from the reads lies in filtering the massively parallel sequencing data using the graph structure prior to traversing possible assembly solutions. In the present work, however, we are focused on the often-overlooked goal of optimal data efficiency. Thus, to the extent possible we distinguish between the read error and the mapping ambiguity associated with the shotgun sampling process. The proposed assembler, X-phased Multibridging, adds information to the assembly graph based on a novel analysis of the underlying reads.

## 2.3 Methods

The path towards developing X-phased Multibridging is outlined as follows.

1. Setting up the shotgun sequencing model and problem formulation.
2. Analyzing repeats structure of genome and their relationship to the information requirement for genome finishing.
3. Developing a parametric probabilistic model that captures the long tail of the repeat statistics.
4. Deriving and analyzing an algorithm that require minimal information requirements for assembly – close to the noiseless lower bound.
5. Performing simulation-based experiments on real and synthetic genomes to characterize the performance of a prototype assembler for genome finishing.
6. Extending the algorithm to address the problem of indel noise.

## 2.4 Shotgun sequencing model and problem formulation

### Sequencing model

Let $\mathbf{s}$ be a length $G$ target genome being sequenced with each base in the alphabet set $\Sigma = \{A, C, G, T\}$. In the shotgun sequencing process, the sequencing instrument samples $N$ reads, $\vec{r}_1, \ldots, \vec{r}_N$ of length $L$ and sampled uniformly and independently from $\mathbf{s}$. This unbiased sampling assumption is made for simplicity and is also supported by the characteristics of single-molecule (e.g. PacBio®) data. Each read is a noisy version of the corresponding

length $L$ substring on the genome. The noise may consist of base insertions, substitutions or deletions. Our analysis focus on substitution noise first. In a later section, indel noise is addressed. In the substitution noise model, let $p$ be the probability that a base is substituted by another base, with probability $p/3$ to be any other base. The errors are assumed to be independent across bases and across reads.

### Formulation

Successful reconstruction by an algorithm is defined by the requirement that, with probability at least $1-\epsilon$, the reconstruction $\hat{\mathbf{s}}$ is a single contig which is within edit distance $\delta$ from the target genome $\mathbf{s}$. If an algorithm can achieve that guarantee at some $(N, L)$, it is called $\epsilon$-feasible at $(N, L)$. This formulation implies automated genome finishing, because the output of the algorithm is one single contig. The fundamental limit for the assembly problem is the set of $(N, L)$ for which successful reconstruction is possible by some algorithms. If $\hat{\mathbf{s}}$ is directly spelled out from a correct placement of the reads, the edit distance between $\hat{\mathbf{s}}$ and $\mathbf{s}$ is of the order of $pG$, where the error rate is $p$. This motivates fixing $\delta = 2pG$ for concreteness. The quality of the assembly can be further improved if we follow the assembly algorithm with a consensus stage in which we correct each base, e.g. with majority voting. But the consensus stage is not the focus in this paper.

## 2.5 Repeats structure and their relationship to the information requirement for successful reconstruction

### Long exact repeats and their relationship to assembly with noiseless reads

We take a moment to carefully define the various types of exact repeats. Let $\mathbf{s}_t^\ell$ denote the length-$\ell$ substring of the DNA sequence $\mathbf{s}$ starting at position $t$. An exact repeat of length $\ell$ is a substring appearing twice, at some positions $t_1, t_2$ (so $\mathbf{s}_{t_1}^\ell = \mathbf{s}_{t_2}^\ell$) that is maximal (i.e. $s(t_1-1) \neq s(t_2-1)$ and $s(t_1+\ell) \neq s(t_2+\ell)$).

Similarly, an exact triple repeat of length-$\ell$ is a substring appearing three times, at positions $t_1, t_2, t_3$, such that $\mathbf{s}_{t_1}^\ell = \mathbf{s}_{t_2}^\ell = \mathbf{s}_{t_3}^\ell$, and such that neither of $s(t_1-1) = s(t_2-1) = s(t_3-1)$ nor $s(t_1+\ell) = s(t_2+\ell) = s(t_3+\ell)$ holds.

A copy of a repeat is a single one of the instances of the substring appearances. A pair of exact repeats refers to two exact repeats, each having two copies. A pair of exact repeats, one at positions $t_1, t_3$ with $t_1 < t_3$ and the second at positions $t_2, t_4$ with $t_2 < t_4$, is interleaved if $t_1 < t_2 < t_3 < t_4$ or $t_2 < t_1 < t_4 < t_3$. The length of a pair of exact interleaved repeats is defined to be the length of the shorter of the two exact repeats. A typical appearance of a

pair of exact interleaved repeat is –X–Y–X–Y– where X and Y represent two different exact repeat copies and the dashes represent non-identical sequence content.

We let $\ell_{max}$ be the length of the longest exact repeat, $\ell_{int}$ be the length of the longest pair of exact interleaved repeats and $\ell_{tri}$ be the length of the longest exact triple repeat.

As mentioned in the introduction, it was observed that the read length and coverage depth required for successful reconstruction using noiseless reads for many genomes is governed by long exact repeats. For some algorithms (e.g. Greedy Algorithm), the read length requirement is bottlenecked by $\ell_{max}$. The Multibridging Algorithm in [7] can successfully reconstruct the genome with a minimum amount of information. The corresponding minimum read length requirement is the critical exact repeat length $\ell_{crit} = \max(\ell_{int}, \ell_{tri})$.

## Flanked repeats

While exact repeats are defined as the segments terminated on each end by a single differing base (Fig 2.3a), flanked repeats are defined by the segments terminated on each end by a statistically uncorrelated region. We call that ending region to be the *random flanking region*. A distinguishing characteristic of the random flanking region is a high Hamming distance to segment length ratio between the ends of two repeat copies. The ratio in the random flanking region is around 0.75, which matches with that when the genomic content is independently and uniformly randomly generated. We observe that long repeats of many genomes terminate with random flanking region. Additional statistical analysis is detailed in the Appendix.

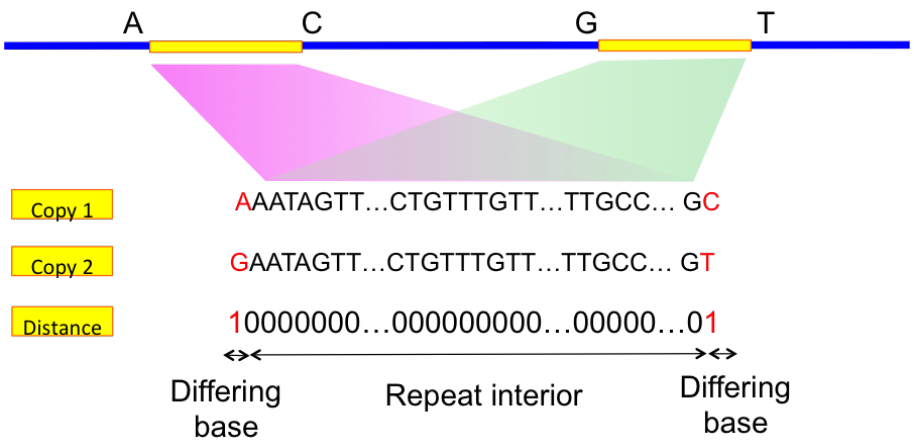
If the repeat interior is exactly the same between two copies of the flanked repeat (Fig 2.3b), the corresponding flanked repeat is called a flanked exact repeat. If there are a few edits (called polymorphism) within the repeat interior (Fig 2.3c), the corresponding flanked repeat is called a flanked approximate repeat.

The length of the repeat interior bounded by the random flanking region is then the flanked repeat length. We let $\tilde{\ell}_{max}$ be the length of the longest flanked repeat, $\tilde{\ell}_{int}$ be the length of the longest pair of flanked interleaved repeats and $\tilde{\ell}_{tri}$ be the length of the longest flanked triple repeat. The critical flanked repeat length is then $\tilde{\ell}_{crit} = \max(\tilde{\ell}_{int}, \tilde{\ell}_{tri})$.

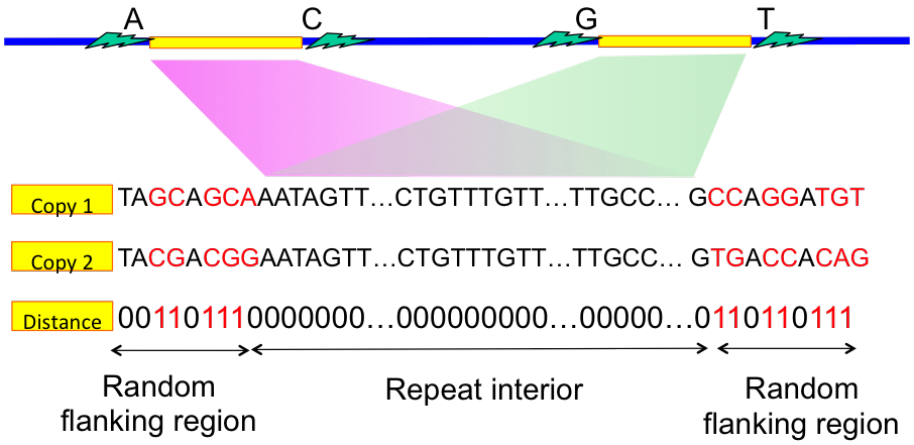
## Long flanked exact repeats and their relationship to assembly with noisy reads

If all long flanked repeats are flanked exact repeats, we can utilize the information in the random flanking region to generalize Greedy Algorithm and Multibridging Algorithm to handle noisy reads. The corresponding information requirement is very similar to that when we are dealing with noiseless reads.

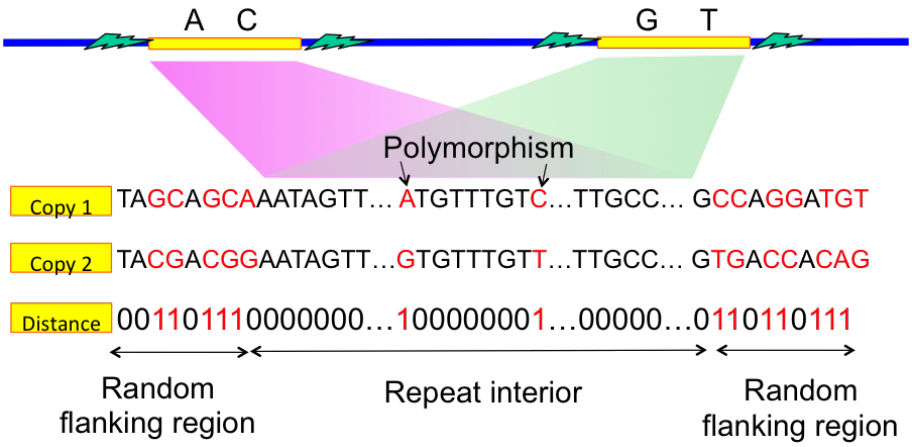
The key intuition is as follows. A criterion for successful reconstruction is the existence of reads to span the repeats to their neighborhood. When a read is noiseless, it only need to be long enough to span the repeat interior to its neighborhood by one base (Fig 2.4a) so as to




(a) Exact repeat




(b) Flanked exact repeat

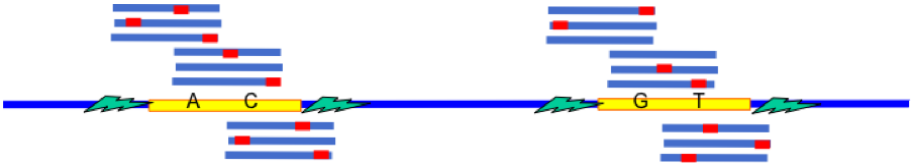



(c) Flanked approximate repeat

Figure 2.3: Repeat pattern


(a) Noiseless reads spanning an exact repeat and its terminating bases


(b) Noisy reads spanning a flanked exact repeat and its terminating random flanking region




(c) Noisy reads extending to span a flanked approximate repeat and its terminating random flanking region

Figure 2.4: Intuition behind the information requirement

differentiate between two exact repeat copies. When a read is noisy, it then need to be long enough to span the repeat interior plus a short extension into the random flanking region (Fig 2.4b) so as to confidently differentiate between two flanked repeat copies. However, the Hamming distance between two flanked repeat copies' neighborhood in the random flanking region is very high even within a short length. This can be used to differentiate between two flanked repeat copies confidently even when the reads are noisy. The short extension into the random flanking region has a length which is typically of order of tens whereas the long repeat length is of order of thousands. Therefore, relative to the repeat length, the change of the critical read length requirement from handling noiseless reads to noisy reads is only marginal when all long repeats are flanked exact repeats.

## Long flanked approximate repeats and their relationship to assembly with noisy reads

If a long flanked repeat is a flanked approximate repeat, the flanked repeat length may be significantly longer than the length of its longest enclosed exact repeat. Merely relying on the information provided by the random flanking region requires the reads to be of length longer than the flanked repeat length for successful reconstruction. This explains why the information requirement for Greedy Algorithm and Multibridging Algorithm has a significant increase when we use noisy reads instead of noiseless reads (as shown in Fig 2.1b). However, if we utilize the information provided by the coverage, we can still confidently differentiate different repeat copies by phasing the small edits within the repeat interior (Fig

2.4c). Specifically, we design X-phased Multibridging whose information requirement is close to the noiseless lower bound even when some long repeats are flanked approximate repeats, as shown in Fig 2.1b.

### From information theoretic insight to algorithm design

Because of the structure of long flanked repeats, there are two important sources of information that we specifically want to utilize when designing data-efficient algorithms to assemble noisy reads. They are

- The random flanking region beyond the repeat interior
- The coverage given by multiple reads overlapping at the same site

Greedy Algorithm(Alg 1) utilizes the random flanking region when considering overlap. The minimum read length needed for successful reconstruction is close to $\tilde{\ell}_{max}$.

Multibridging Algorithm(Alg 2) also utilizes the random flanking region but it improves upon Greedy Algorithm by using a De Bruijn graph to aid the resolution of flanked repeats. The minimum read length needed for successful reconstruction is close to $\tilde{\ell}_{crit}$.

X-phased Multibridging(Alg 3) further utilizes the coverage given by multiple reads to phase the polymorphism within the repeat interior of flanked approximate repeats. The minimum read length needed for successful reconstruction is close to $\ell_{crit}$, which is the noiseless lower bound even when some long repeats are flanked approximate repeats.

## 2.6 Model for genome

To capture the key characteristics of repeats and to guide the design of assembly algorithms, we use the following parametric probabilistic model for genome. A target genome is modeled as a random vector $\mathbf{s}$ of length $G$ that has the following three key components (a pictorial representation is depicted in Figure 2.5).

**Random background:** The background of the genome is a random vector, composed of uniformly and independently picked bases from the alphabet set $\Sigma = \{A, C, G, T\}$.

**Long flanked repeats:** On top of the random background, we randomly position the longest flanked repeat and the longest flanked triple repeat. Moreover, we randomly position a flanked repeat interleaving the longest flanked repeat, forming the longest pair of flanked interleaved repeat. The corresponding length of the flanked repeats are $\tilde{\ell}_{\mathrm{max}}$, $\tilde{\ell}_{tri}$ and $\tilde{\ell}_{int}$ respectively. It is noted that $\tilde{\ell}_{\mathrm{max}} > \max(\tilde{\ell}_{\mathrm{int}}, \tilde{\ell}_{\mathrm{tri}})$.

**Polymorphism and long exact repeats:** Within the repeat interior of the flanked repeats, we randomly position $n_{\mathrm{max}}$, $n_{int}$ and $n_{tri}$ edits (polymorphism) respectively. The sites of polymorphism are chosen such that the longest exact repeat, the longest pair of exact interleaved repeats and the longest exact triple repeat are of length $\ell_{\mathrm{max}}$, $\ell_{\mathrm{int}}$ and $\ell_{tri}$ respectively.

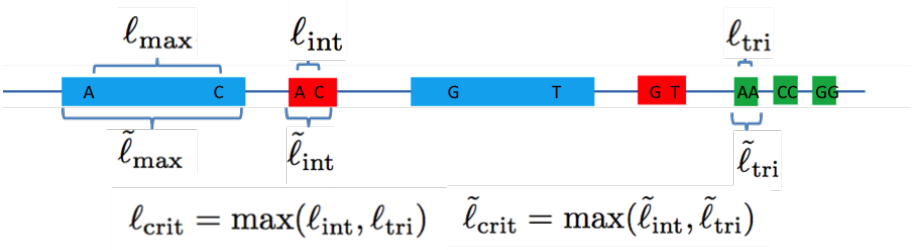



Figure 2.5: Model for genome

## 2.7 Algorithm design and analysis

### Greedy Algorithm

Read $R_2$ is a successor of read $R_1$ if there exists length-$W$ suffix of $R_1$ and length-$W$ prefix of $R_2$ such that they are extracted from the same locus on the genome. Furthermore, there is no other reads that can satisfy the same condition with a larger $W$. To properly determine successors of reads in the presence of long repeats and noise, we need to define an appropriate overlap rule for reads. In this section, we show the conceptual development towards defining such a rule, which is called RA-rule.

**Noiseless reads and long exact repeats:** If the reads are noiseless, all reads can be paired up with their successors correctly with high probability when the read length exceeds $\ell_{max}$. It was done [7] by greedily pairing reads and their candidate successors based on their overlap score in descending order. When a read and a candidate successor are paired, they will be removed from the pool for pairing. Here the overlap score between a read and a candidate successor is the maximum length such that the suffix of the read and prefix of the candidate successor match *exactly*.

**Noisy reads and random background:** Since we cannot expect exact match for noisy reads, we need a different way to define the overlap score. Let us consider the following toy situation. Assume that we have exactly one length-$(\ell + 1)$ noisy read starting at each locus of a length $G$ random genome(i.e. only consists of the random background). Each read then overlaps with its successor precisely by $\ell$ bases. Analogous to the noiseless case, one would expect to pair reads greedily based on overlap score. Here the overlap score between a read and a candidate successor is the maximum length such that the suffix ($x$) of the read and prefix ($y$) of the candidate successor match *approximately*. To determine whether they match *approximately*, one can use a predefined a threshold factor $\alpha$ and compute the Hamming distance $d(x, y)$. If $d(x, y) \leq \alpha \cdot \ell$, then they match *approximately*, otherwise not. Given this decision rule, we can have false positive (i.e. having any pairs of reads mistakenly paired up) and false negative (i.e. having any reads not paired up with the true successors). If false positive and false negative probability are small, this naive method is a reliable enough metric. This can be achieved by using a long enough length $\ell > \ell_{iid}$ and an appropriately chosen threshold $\alpha$.

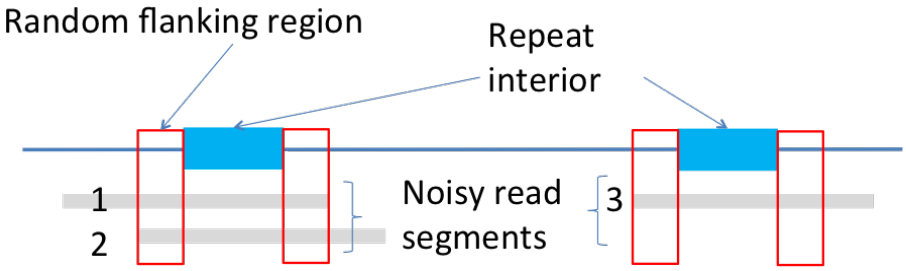



Figure 2.6: Intuition about why we define the overlap rule to be RA-overlap rule

Recall that $\epsilon$ is the overall failure probability. By bounding the sum of false positive and false negative probability by $\epsilon/3$, one can find $\ell_{iid}(p, \epsilon/3, G)$ and $\alpha(p, \epsilon/3, G)$ to be the $(\ell_{iid}, \alpha)$ solution to the following pair of equations:

$$G^2 \cdot \exp(-\ell_{iid} \cdot D(\alpha||\frac{3}{4})) = \tfrac{\epsilon}{6} \quad (2.1)$$

$$G \cdot \exp(-\ell_{iid} \cdot D(\alpha||2p - \frac{4}{3}p^2)) = \tfrac{\epsilon}{6} \quad (2.2)$$

where $D(a||b) = a \log \frac{a}{b} + (1-a) \log \frac{1-a}{1-b}$ is the Kullback-Leibler divergence.

**Noisy reads and long flanked repeats:** However, when the genome contains long flanked repeats on top of the random background, this naive rule of determining overlap is not enough. Let us look at the example in Fig. 2.6. As shown in Fig. 2.6, because of long flanked repeats, we have a small ratio of overall distance against the overlap length for the segments that are extracted from different copies of the repeat (e.g Segment 1 and Segment 3 in Fig. 2.6). Therefore, the overall Hamming distance between two segments is not a good enough metric for defining overlap. If we abide by the naive rule, we need to increase the read length significantly longer than the flanked repeat length so as to guarantee confidence in deciding approximate match. Otherwise, it will either result in a high false positive rate (if we set a large $\alpha$) or a high false negative rate (if we set a small $\alpha$). To properly handle such scenario, we define a repeat-aware rule(or RA-rule).

- RA-matching: Two segments $(x, y)$ of length $W$ match under the RA-rule if and only if the distance between whole segments is $< \alpha \cdot W$ and **both** of its ending segments(of length $\ell_{iid}$) also have distance $< \alpha \cdot \ell_{iid}$.
- RA-overlap: The overlap score between a read and a candidate successor under the RA-rule is the maximum length such that the suffix of the read and prefix of the candidate successor match under the RA-matching.

The RA-rule is particularly useful because it puts an emphasis on both ends of the overlap region. Since the ends are separated by a long range, one end will hopefully originate from the

random flanking region of the flanked repeat. If we focus on the segments originating from the random flanking region, the distance per segment length ratio will be very high when the segments originate from different copies of the repeat but very low when they originate from the same copy of the repeat. This is how we utilize the random flanking region to differentiate between repeat copies and determine correct successors in the presence of long flanked repeats and noise.

If we use Greedy Algorithm (Alg 1) to merge reads greedily with this overlap rule (RA-rule), Prop 2.7.1 shows the information requirement under the previously described sequencing model and genome model. A plot is shown in Fig 2.1b. Since $\ell_{\imath\imath d}$ is of order of tens whereas $\tilde{\ell}_{max}$ is of order of thousands, the read length requirement for Greedy Algorithm to succeed is dominated by $\tilde{\ell}_{max}$. The detailed proof of Prop 1 is given in Appendix.

**Algorithm 1** Greedy Algorithm

Initialize contigs to be reads
**for** $W = L$ *to* $\ell_{\imath\imath d}$ **do**
  **if** *any two contigs x,y are of overlap* $W$ *under RA-rule* **then**
    merge $x, y$ into one contig.
  **end**
**end**

**Proposition 2.7.1.** *With* $\ell_{\imath\imath d} = \ell_{\imath\imath d}(p, \frac{\epsilon}{3}, G)$, *if*

$$L > \tilde{\ell}_{max} + 2\ell_{\imath\imath d},$$

$$N > \max\left(\frac{G\ln(3/\epsilon)}{L - \tilde{\ell}_{max} - 2\ell_{\imath\imath d}}, \frac{G\ln(3N/\epsilon)}{L - 2\ell_{\imath\imath d}}\right)$$

*then, Greedy Algorithm(Alg 1) is* $\epsilon-$*feasible at* $(N, L)$.

## Multibridging Algorithm

The read length requirement of Greedy Algorithm has a bottleneck around $\tilde{\ell}_{max}$ because it requires at least one copy of each flanked repeat to be spanned by at least one read for successful reconstruction. Spanning a repeat by a single read is called bridging in [7]. A natural question is whether we need to have all repeats bridged for successful reconstruction.

In the noiseless setting, [7] shows that this condition can be relaxed. Using noiseless reads, one can have successful reconstruction given all copies of each exact triple repeat being bridged, and at least one copy of one of the repeats in each pair of exact interleaved repeats being bridged.

A key idea to allow such a relaxation in [7] is to use a De Bruijn graph to capture the structure of the genome.

When the reads are noisy, we can utilize the random flanking region to specify a De Bruijn graph with high confidence by RA-rule and arrive at a similar relaxation. By some

graph operations to handle the residual errors, we can have successful reconstruction with read length $\tilde{\ell}_{crit}+2\cdot\ell_{iid} < L < \tilde{\ell}_{max}$. The algorithm is summarized in Alg 2. Prop 2.7.2 shows its information requirement under the previously described sequencing model and genome model. A plot is shown in Fig 2.1b. We note that Alg 2 can be seen as a noisy reads generalization of Multibridging Algorithm for noiseless reads in [7].

**Description and its performance**

**Proposition 2.7.2.** *With* $\ell_{iid} = \ell_{iid}(p, \frac{\epsilon}{3}, G)$, if

$$L > \tilde{\ell}_{crit} + 2\ell_{iid},$$
$$N > \max\left(\frac{G\ln(3/\epsilon)}{L - \tilde{\ell}_{crit} - 2\ell_{iid}}, \frac{G\ln(3N/\epsilon)}{L - 2\ell_{iid}}\right)$$

*then, Multibridging Algorithm(Alg 2) is* $\epsilon-$*feasible at* $(N, L)$.

Detailed proof is given in the Appendix. The following sketch highlights the motivation behind the key steps of Multibridging Algorithm.

[Step1] We set a large K value to make sure the K-mers overlapping the shorter repeat of the longest pair of flanked interleaved repeats and the longest flanked triple repeat can be separated as distinct clusters.

[Step2] Clustering is done using the RA-rule because of the existence of long flanked repeats and noise.

[Step3] A K-mer cluster corresponds to an equivalence class for K-mers matched under the RA-rule. This step forms a De Bruijn graph with K-mer clusters as nodes.

[Step4] Because of large K, the graph can be disconnected due to insufficient coverage. In order to reduce the coverage constraint, we connect the clusters greedily.

[Step5, 7] These two steps simplify the graph.

[Step6] Branch clearing repairs any incorrect merges near the boundary of long flanked repeat.

[Step8] Since an Euler path in the condensed graph corresponds to the correct genome sequence, it is traversed to form the reconstructed genome.

**Algorithm 2** Multibridging Algorithm

1. Choose K to be $\tilde{\ell}_{crit} + 2\ell_{iid}$ and extract K-mers from reads.
2. Cluster K-mers based on the RA-rule.
3. Form uncondensed De Bruijn graph $G_{De-Bruijn} = (V, E)$ with the following rule:
   - a) K-mers clusters as node set $V$.
   - b) $(u, v) = e \in E$ if and only if there exists K-mers $u_1 \in u$ and $v_1 \in v$ such that $u_1$,$v_1$are consecutive K-mers in some reads.
4. Join the disconnected components of $G_{De-Bruijn}$ together by the following rule:

**for** $W = K - 1$ *to* $\ell_{iid}$ **do**
  **for** *each node u which has either no predecessors / successors in* $G_{De-Bruijn}$ **do**
    a) Find the predecessor/successor $v$ for $u$ from all possible K-mers clusters such that overlap length(using any representative K-mers in that cluster) between $u$ and $v$ is $W$ under RA-rule.
    b) Add dummy nodes in the De Bruijn graph to link $u$ with $v$ and update the graph to $G_{De-Bruijn}$
  **end**
**end**

5. Condense the graph $G_{De-Bruijn}$ to form $G_{string}$ with the following rule:
   - a) Initialize $G_{string}$ to be $G_{De-Bruijn}$ with node labels of each node being its cluster group index.
   - b) **while** $\exists$*successive nodes* $u \to v$ *such that* $out - degree(u) = 1$*and* $in - degree(v) = 1$ **do**
     bi) Merge $u$ and $v$ to form a new node $w$
     bii) Update the node label of $w$ to be the concatenation of node labels of $u$ and $v$
     **end**
6. Clear Branches of $G_{string}$:

**for** *each node u in the condensed graph* $G_{string}$ **do**
  **if** $out - degree(u) > 1$ *and that all the successive paths are of the same length(measured by the number of node labels) and then joining back to node v and the path length* $< \ell_{iid}$ **then**
    we merge the paths into a single path from $u$ to $v$.
  **end**
**end**

7. Condense graph $G_{string}$
8. Find the genome :
   - a) Find an Euler Cycle/Path in $G_{string}$ and output the concatenation of the node labels to form a string $\vec{\mathbf{s}}_{labels}$.
   - b) Using $\vec{\mathbf{s}}_{labels}$ and look up the associated K-mers to form the final recovered genome $\hat{\mathbf{s}}$.

**Some implementation details: improvement on time and space efficiency**

For Multibridging Algorithm, the most computational expensive step is the clustering of K-mers. To improve the time and space efficiency, this clustering step can be approximated by performing pairwise comparison of reads.

Based on the alignment of the reads, we can cluster K-mers from different reads together using a disjoint set data structure that supports union and find operations. Since only reads are used in the alignment, only the K-mer indices along with their associated read indices and offsets need to be stored in memory— not all the K-mers.

Pairwise comparison of reads roughly runs in $\tilde{\Theta}(N^2L^2)$ if done in the naive way. To speed up the pairwise comparison of noisy reads, one can utilize the fact that the read length is long. We can extract all consecutive $f$-mers (which act as fingerprints) of the reads and do a lexicographical sort to find candidate neighboring reads and associated offsets for comparison. Since the reads are long, if two reads overlap, there should exist some perfectly matched $f$-mers which can be identified after the lexicographical sort. This allows an optimized version of Multibridging Algorithm to run in $\tilde{\Theta}(NL \cdot \frac{NL}{G})$ time and $\tilde{\Theta}(NLf)$ space.

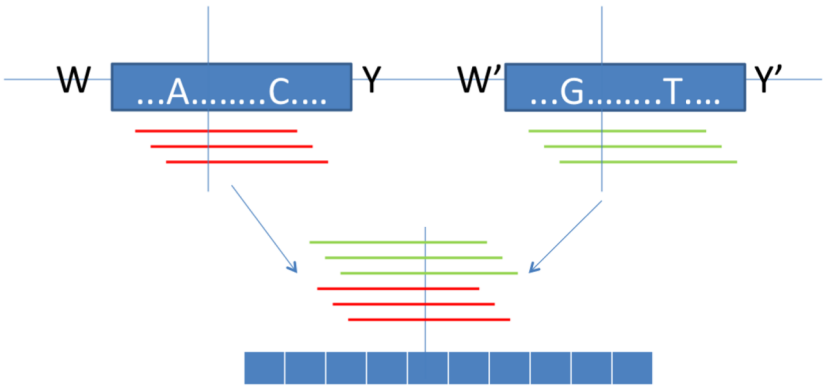
## X-phased Multibridging

As shown in Fig 2.1b, when long repeats are flanked approximate repeats, there can be a big gap between the noiseless lower bound and the information requirement for Multibridging Algorithm. A natural question is whether this is due to a fundamental lack of information from the reads or whether Multibridging Algorithm does not utilize all the available information. In this section, we demonstrate that there is an important source of information provided by coverage which is not utilized by Multibridging Algorithm. In particular, we introduce X-phased Multibridging, an assembly algorithm that utilizes the information provided by coverage to phase the polymorphism in long flanked repeat interior. The information requirement of X-phased Multibridging is close to the noiseless lower bound (as shown in Fig 2.1b) even when some long repeats are flanked approximate repeats.

### Description of X-phased Multibridging

Multibridging Algorithm utilizes the random flanking region to differentiate between repeat copies. However, for a flanked approximate repeat, its enclosed exact repeat does not terminate with the random flanking region but only terminates with sparse polymorphism. When we consider the overlap of *two* reads originating from different copies of a flanked approximate repeat, the distinguishing polymorphism is so sparse that it cannot be used to confidently differentiate between repeat copies. Therefore, there is a need to use the extra redundancy introduced by the coverage from *multiple* reads to confidently differentiate between repeat copies and that is what X-phased Multibridging utilizes.

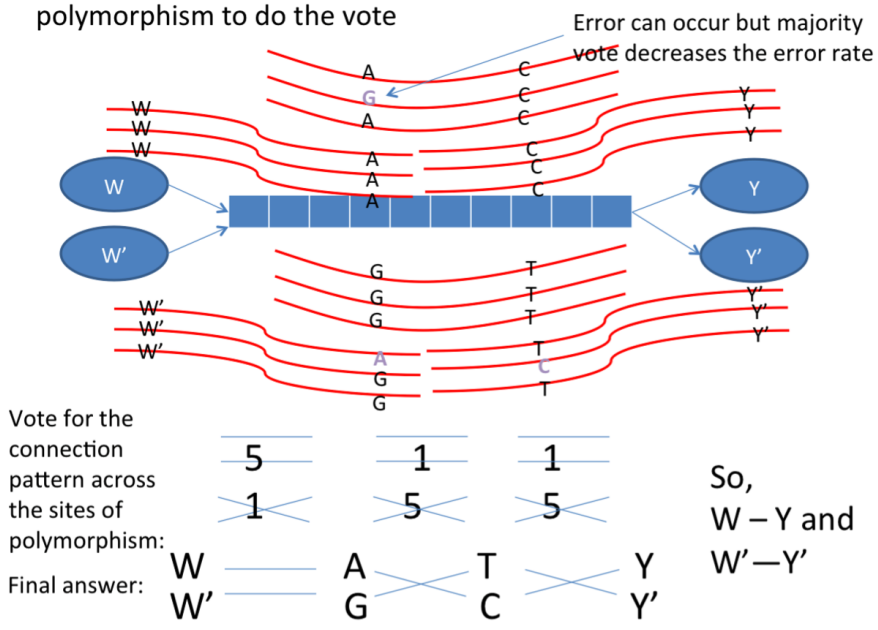
X-phased Multibridging (Alg 3) follows the algorithmic design of Multibridging Algorithm. However, it adds an extra phasing procedure to differentiate between repeat copies




1. Consensus step:
Use all reads to decide where the sites of polymorphism are within the repeat ( e.g. Given 60 reads overlap at a base and its counter part in the other copy; If 30 A, 30 G => it is a site of polymorphism; 54 A, 6G => it is not a site of polymorphism)

(a) Consensus Step

2. Read extension step : Use the reads that span the sites of polymorphism to do the vote




(b) Read Extension Step

Figure 2.7: Illustration of how to phase polymorphism to extend reads across repeats

of long flanked repeats that Multibridging Algorithm cannot confidently differentiate. We recall that after running step 7 of Multibridging Algorithm, a node in the graph $G_{string}$ corresponds to a substring of the genome and has node label consisting of consecutive K-mer cluster indices. An X-node of $G_{string}$ is a node that has in-degree and out-degree $\geq 2$. X-node indeed corresponds to a flanked repeat. The incoming/outgoing nodes of the X-node correspond to the incoming/outgoing random flanking region of the flanked repeat.

To be concrete, we focus the discussion on a pair of flanked interleaved repeats, assuming triple repeats are not the bottleneck. However, the ideas presented can be generalized to repeats of more copies.

For the flanked approximate repeat with length $\ell_{int} < L$ and $\tilde{\ell}_{int} > L$ (as shown in Fig 2.7), there is no node-disjoint paths joining incoming/outgoing random flanking region with the distinct repeat copies in $G_{string}$. It is because the reads are corrupted by noise and the polymorphism is too sparse to differentiate between the repeat copies. Executing Multibridging Algorithm directly will result in the formation of an X-node, which is an artifact due to K-mers from different copies of the flanked approximate repeat erroneously clustered together.

Successful reconstruction requires an algorithm to pair up the correct incoming/outgoing nodes of the X-node(i.e. decide how $W, W'$ and $Y, Y'$ are linked in Fig 2.7). This is handled by the phasing procedure in X-phased Multibridging, which uses all the reads information. The phasing procedure is composed of two main steps:

- Consensus step: Confidently find out where the sites of polymorphism are located within the flanked repeat interior.
- Read extension step: Confidently determine how to extend reads using the random flanking region and sites of polymorphism as anchors.

**Consensus step** For the X-node of interest, let $D$ be the set of reads originating from any sites of the associated flanked repeat region and let $x_1$ and $x_2$ denote the associated repeat copies. Since the random flanking region is used as anchor, it is treated as the starting base (i.e. $x_1(0) = W$ and $x_2(0) = W'$). For the $i^{th}$ subsequent site of the flanked repeat (where $1 \leq i \leq \tilde{\ell}_{int}$), we determine the consensus according to Eq (2.3). This can be implemented by counting the frequency of occurrence of each alphabet overlapping at each site of the repeat. The consensus result determines the sites of polymorphism and the most likely pairs of bases at the sites of polymorphism.

$$\max_{F \subset \{A,C,G,T\}^2} \mathcal{P}(\{x_1(i), x_2(i)\} = F \mid D) \tag{2.3}$$

**Read extension step** After knowing the sites of polymorphism, we use those reads that span the sites of polymorphism or random flanking region to help decide how to extend reads across the flanked repeat. Let $\sigma$ be the possible configuration of alphabets at the sites of polymorphism and random flanking region (e.g. $\sigma = (ACY, GTY')$ means that the two

copies of the flanked repeat with the corresponding random flanking region respectively are W–A–C–Y, W'–G–T–Y' where the common bases are omitted).

The following maximum a posteriori estimation is used to decide the correct configuration.

$$\max_{\sigma} \mathcal{P}(\hat{\sigma} = \sigma \mid D, \{x_1(i), x_2(i)\}_{i=1}^{\tilde{\ell}_{int}}) \tag{2.4}$$

where $\hat{\sigma}$ is the estimator, $D$ is the raw read set, and $x_1, x_2$ are the estimates from the consensus step. It is noted that the size of the feasible set for $\sigma$ is $2^{n_{int}+1}$.

In practice, for computational efficiency, the maximization in Eq (2.4) can be approximated accurately even if it is replaced by the simple counting illustrated in Fig 2.7, which we call count-to-extend algorithm(countAlg). CountAlg uses the raw reads to establish majority vote on how one should extend to the next sites of polymorphism using only the reads that span the sites of polymorphism.

### Performance

After introducing the phasing procedure in X-phased Multibridging, we proceed to find its information requirement for successful reconstruction.

The information requirement for X-phased Multibridging is the amount of information required to reduce the error of the phasing procedure to a negligible level. The phasing procedure – step 2 in Alg. 3 – is a combination of consensus and read extension steps, which contribute to the error as follows.

Let $\mathcal{E}$ be the error event of the repeat phasing procedure for a repeat, $\epsilon_1$ be the error probability for the consensus step, $\epsilon_2$ be the error probability for the read extension step given $k$ reads spanning each consecutive site of polymorphism within the flanked repeat, $\delta_{cov}$ be the probability for having $k$ reads spanning each consecutive sites of polymorphism(i.e. $k$ bridging reads) within the flanked repeat. We have,

$$\mathcal{P}(\mathcal{E}) \leq \epsilon_1 + \epsilon_2 + \delta_{cov} \tag{2.5}$$

Therefore, to guarantee confidence in the phasing procedure, it suffices to upper bound $\epsilon_1$, $\epsilon_2$ and $\delta_{cov}$. We tabulate the error probabilities of $\epsilon_1$, $\epsilon_2$ in Table 2.1 for phasing a flanked repeat (whose length is 5000 whereas the genome length is 5M). The flanked repeat has two sites of polymorphism which partition it into three equally spaced segments.

From Table 2.1, when $p = 0.01$, the information requirement translates to the condition of having three bridging reads spanning the shorter exact repeat of the longest pair of exact interleaved repeats. Therefore, the information requirement for X-phased Multibridging shown in Fig 2.1b also corresponds to this condition. It is noted that X-phased Multibridging has the same vertical asymptote as the noiseless lower bound. The vertical shift is due to the increase of requirement on the number of bridging reads from $k = 1$ (noiseless case) to $k = 3$ (noisy case).

**Algorithm 3** X-phased Multibridging

1. Perform Step 1 to Step 7 of MultiBridging Algorithm
2. For every X-node $x \in G_{string}$
   - a) Align all the relevant reads to the flanked repeat $x$
   - b) Consensus step: Consensus to find location of polymorphism by solving Eq (2.3)
   - c) Read extension step: If possible, resolve flanked repeat(i.e. pair up the incoming/outgoing nodes of $x$) by either countAlg or by solving Eq (2.4)
3. Perform Step 8 of MultiBridging Algorithm as in Alg 2

| $p$ | Coverage (NL/G) | $\epsilon_1$ |
|---|---|---|
| 0.01 | 20 | 0.00 |
| 0.01 | 40 | 0.00 |
| 0.01 | 60 | 0.00 |
| 0.1 | 20 | 0.16 |
| 0.1 | 40 | 0.00 |
| 0.1 | 60 | 0.00 |

(a) Calibration for $\epsilon_1$

| $p$ | Number of bridging reads $k$ | Upper bound for $\epsilon_2$ |
|---|---|---|
| 0.01 | 1 | 0.060 |
| 0.01 | 3 | 0.0036 |
| 0.01 | 5 | 0.00024 |
| 0.1 | 11 | 0.089 |
| 0.1 | 21 | 0.022 |
| 0.1 | 31 | 0.0059 |

(b) Calibration for $\epsilon_2$

Table 2.1: Calibration of error probability made by the phasing procedure of X-phased Multibridging

## 2.8 Simulation of the prototype assembler

Based on the algorithmic design presented, we implement a prototype assembler for automatic genome finishing using reads corrupted by substitution noise. First, the assembler was tested on synthetic genomes, which were generated according to the genome model described previously. This demonstrates a proof-of-concept that one can achieve genome finishing with read length close to $\ell_{crit}$, as shown in Fig 2.8. The number on the line represents the number

of simulation rounds (out of 100) in which the reconstructed genome is a single contig with $\geq 99\%$ of its content matching the ground truth.

Second, the assembler was tested using synthetic reads, sampled from genome ground truth downloaded from NCBI. The assembly results are shown in Table 2.2. The observation from the simulation result is that we can assemble genomes to finishing quality with information requirement near the noiseless lower bound. More information about the detail design of the prototype assembler is presented in the Appendix.

| Index | Species | $G$ | $p$ | $\frac{NL}{G}$ | $L$ | $\tilde{\ell}_{max}$ | $\tilde{\ell}_{crit}$ | $\ell_{crit}$ | % match | Ncontig | $\frac{N}{N_{noiseless}}$ | $\frac{L}{\ell_{crit}}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | a | 1440371 | 1.5% | 37.36 X | 930 | 1817 | 803 | 770 | 100.00 | 1 | 1.57 | 1.21 |
| 2 | a | 1440371 | 1.5% | 33.14 X | 970 | 1817 | 803 | 770 | 99.95 | 1 | 1.67 | 1.26 |
| 3 | a | 1440371 | 1.5% | 29.60 X | 1000 | 1817 | 803 | 770 | 99.99 | 1 | 1.66 | 1.30 |
| 4 | b | 1589953 | 1.5% | 40.82 X | 2440 | 4183 | 2155 | 2122 | 100.00 | 1 | 1.30 | 1.15 |
| 5 | b | 1589953 | 1.5% | 21.31 X | 2752 | 4183 | 2155 | 2122 | 99.99 | 1 | 1.19 | 1.30 |
| 6 | b | 1589953 | 1.5% | 20.66 X | 2900 | 4183 | 2155 | 2122 | 99.99 | 1 | 1.35 | 1.37 |
| 7 | c | 1772693 | 1.5% | 30.03 X | 3950 | 5018 | 3234 | 3218 | 99.96 | 1 | 1.36 | 1.23 |
| 8 | c | 1772693 | 1.5% | 21.96 X | 4279 | 5018 | 3234 | 3218 | 99.97 | 1 | 1.33 | 1.33 |
| 9 | c | 1772693 | 1.5% | 17.03 X | 4700 | 5018 | 3234 | 3218 | 100.00 | 1 | 1.31 | 1.46 |
| 10 | d | 1006701 | 1.5% | 35.23 X | 6867 | 15836 | 10518 | 5494 | 99.05 | 1 | 1.72 | 1.25 |
| 11 | d | 1006701 | 1.5% | 19.88 X | 7500 | 15836 | 10518 | 5494 | 97.86 | 1 | 1.30 | 1.37 |
| 12 | d | 1006701 | 1.5% | 17.69 X | 9000 | 15836 | 10518 | 5494 | 98.10 | 1 | 1.68 | 1.64 |

Table 2.2: Simulation results on the assembly of several real genomes using reads corrupted by substitution noise ((a) *Prochlorococcus marinus* (b) *Helicobacter pylori* (c) *Methanococcus maripaludis* (d) *Mycoplasma agalactiae*) with $\ell_{crit} = \max(\ell_{int}, \ell_{tri})$ , $\tilde{\ell}_{crit} = \max(\tilde{\ell}_{int}, \tilde{\ell}_{tri})$ and $N_{noiseless}$ is the lower bound on number of reads in the noiseless case for $1 - \epsilon = 95\%$ confidence recovery

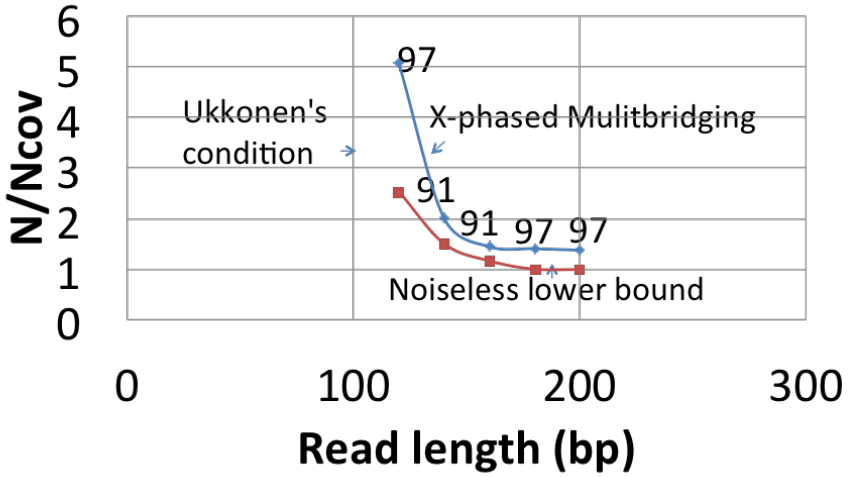
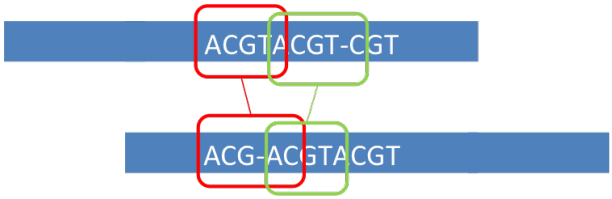
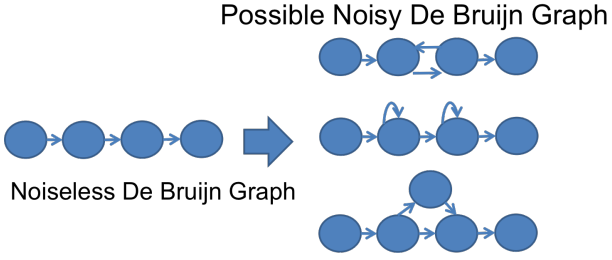



Figure 2.8: Simulation results on the assembly of synthetic genomes using reads corrupted by substitution noise. The parameters are as follows. $G = 10K; \tilde{\ell}_{max} = \ell_{max} = 500, \tilde{\ell}_{int} = 200, \ell_{int} = 100$ with two sites of polymorphism within the flanked repeat. $p = 1.5\%, \epsilon = 5\%$.




Group K-mers into clusters according to the alignment within the overlap region

(a) Form K-mer Clusters




(b) Abnormality in (indel) De Bruijn Graph

Figure 2.9: Treatment of reads corrupted by indel noise

## 2.9 Extension to handle indel noise

A further extension of the prototype assembler addresses the case of reads corrupted by indel noise. Similar to the case of substitution noise, tests were performed on synthetic reads sampled from real genomes and synthetic genomes. Simulation results are summarized in Table 2.3 where $p_i, p_d$ are insertion probability and deletion probability and rate is the number of successful reconstruction(i.e. simulation rounds that show mismatch $< 5\%$) divided by total number of simulation rounds. The simulation result for indel noise corrupted reads shows that X-phased Multibridging can be generalized to assemble indel noise corrupted reads. The information requirement for automated finishing is about a factor of two from the noiseless lower bound for both N and L.

We remark that one non-trivial generalization is the way that we form the noisy De Bruijn graph for K-mer clusters. In particular, we first compute the pairwise overlap alignment among reads, then we use the overlap alignment to group K-mers into clusters. Subsequently, we link successive cluster of K-mers together as we do in Alg 2. An illustration is shown in Fig 2.9a. However, due to the noise being indel in nature, the edges in the noisy De Bruijn graph may point in the wrong direction as shown in Fig 2.9b. In order to handle this, we traverse the graph and remove such abnormality when they are detected.

| Type | $G$ | $p_i$ | $p_d$ | $\frac{NL}{G}$ | $L$ | $\tilde{\ell}_{max}$ | $\tilde{\ell}_{crit}$ | $\ell_{crit}$ | $\frac{N}{N_{noiseless}}$ | $\frac{L}{\ell_{crit}}$ | Rate |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Synthetic | 50000 | 1.5% | 1.5% | 23.0 X | 200 | 500 | 200 | 100 | 2.25 | 2 | 28/30 |
| Synthetic | 50000 | 1.5% | 1.5% | 24.1 X | 180 | 500 | 200 | 100 | 2.33 | 1.8 | 27/30 |
| a | 1440371 | 1.5% | 1.5% | 28.53 X | 1000 | 1817 | 803 | 770 | 1.60 | 1.30 | 1/1 |
| b | 1589953 | 1.5% | 1.5% | 20.66 X | 2900 | 4183 | 2155 | 2122 | 1.35 | 1.37 | 1/1 |

Table 2.3: Simulation results on the assembly of real/synthetic genomes using reads corrupted by indel noise(Synthetic: randomly generated to fit $\tilde{\ell}_{max}, \tilde{\ell}_{crit}, \ell_{crit}$; (a) : *Prochlorococcus marinus* ; (b): *Helicobacter pylori*)

## 2.10 Conclusion

In this work, we show that even when there is noise in the reads, one can successfully reconstruct with information requirements close to the noiseless fundamental limit. A new assembly algorithm, X-phased Multibridging, is designed based on a probabilistic model of the genome. It is shown through analysis to perform well on the model, and through simulations to perform well on real genomes.

The main conclusion of this work is that, with an appropriately designed assembly algorithm, the information requirement for genome assembly is insensitive to moderate read noise. We believe that the information theoretic insight is useful to guide the design of future assemblers. We hope that these insights allow future assemblers to better leverage the high throughput sequencing read data to provide higher quality assembly.

# Chapter 3

# Towards computation, space, and data efficiency in de novo genome assembly

We consider the problem of de novo genome assembly from shot gun data, wherein an underlying (unknown) DNA sequence is to be reconstructed from several short substrings of the sequence. We propose a de novo assembly algorithm, OnlineOfflineAlgo. For noiseless reads of length at least twice the minimum length, OnlineOfflineAlgo requires minimum coverage, is time-efficient and is space-optimal under the i.i.d. genome model. The key idea to achieve space and time efficiency is to break the procedure into two phases, an online and an offline phase. We design the algorithm from an information theoretic perspective of using minimum amount of data. The key idea to achieve space and computational efficiency is to break the procedure into two phases, an online and an offline phase. We remark that this can serve as an evidence of the feasibility of using an information-theoretic perspective to guide practical algorithmic design in de novo genome assembly.

## 3.1 Introduction

De novo genome assembly is an important process that contributes greatly to the understanding and advancement of medicine and biology. The advent of next-generation sequencing technologies has drastically reduced the costs and the time associated to DNA sequencing in the last ten years [62]. Under these next-generation technologies, reads (i.e., substrings) of the underlying DNA sequence are first obtained, from which the underlying sequence is then inferred computationally. Due to the massive amounts of data generated, there is a great need to address the computational aspects of the sequencing process [51, 53]. Thus, algorithm design for assembly of such reads for DNA sequencing remains an important algorithmic challenge.

In this chapter, we consider the problem of designing efficient algorithms for de novo genome assembly from shot-gun data. We present a novel algorithm for de novo genome assembly that is efficient in terms of both (storage-)space and computations, and furthermore,

minimizes the number of reads required for recovering the underlying DNA sequence.

The approach we take towards algorithm design is based on an information-theoretic perspective [38]. This model, which we shall refer to as the 'i.i.d. model', is a simple generative model for the DNA sequence and the reading process. This model, per se, is not a very accurate description of real DNA sequences, but provides good insights with respect to this problem. In particular, this model has previously been employed to analyse the fundamental requirements of a sequencing process (e.g., the number of reads required for correct reconstruction) and to analytically compare the performance various existing algorithms. In this chapter, we first design an efficient sequencing algorithm based on this model, and subsequently use this as a building block towards de novo assembly in more realistic scenarios.

The contributions of this work thus are:

1. construction of a space, time and data efficient de novo sequencing algorithm based on the i.i.d. model,
2. construction of a framework towards building efficient de novo assembly algorithms for real data, and
3. providing evidence of the feasibility of a information-theoretic perspective to building practical algorithms for de novo assembly (as opposed to using it for the purposes of analysis alone).

The key idea behind this algorithm, that helps it retain both space and computational efficiency, is to divide the task into two phases: an online phase and an offline phase. Current algorithms (e.g., [50, 8, 17, 13, 53]) operate only under an offline phase, wherein they require all the reads at their disposal at the beginning of the execution of the algorithm. Such an approach leads to a considerably high storage space requirement since all the reads are required to be stored, and are then processed jointly. On the other hand, the online phase of our algorithm processes and combines reads on the fly, thereby offering a significant reduction in the storage space requirements. We provide additional algorithmic novelties to ensure that the two phases are executed in an efficient manner.

We note that while a lower computational complexity directly relates to lowering the sequencing time, a low space requirement can also contribute significantly to this cause. A lower storage requirement essentially enables the storage of *all the data* (i.e., all the necessary information required for assembly) in the faster but limited main memory. This results in a faster sequencing as compared to algorithms that have a larger storage footprint, and thus mandate either buying a greater amount of (expensive) main memory, or store and read part of the data from the (slower) disks.

The remainder of the chapter is organized as follows. Section3.2 describes the i.i.d. generative model of [38], along with the fundamental bounds on the various parameters for exact recovery. Section3.3 presents a space-and-computation-efficient and data-requirement-optimal assembly algorithm under the i.i.d. generative model. Section3.4 presents an analysis of this algorithm. Conclusion is drawn in Section3.5.

## 3.2 The basic generative model

### Notation

Assume that the underlying sequence is haploid, and is $G$ bases long. Denote the sequence by a $G$-length vector $\vec{x}_G$, with each of its elements being $A$, $C$, $G$, or $T$. We have at our disposal, $N$ reads (substrings) of $\vec{x}_G$, each of length $L$, and the goal is to reconstruct $\vec{x}_G$from these $N$ reads.

Throughout the chapter, we shall use the terms underlying sequence and underlying genome interchangeably.

### The i.i.d., Error-free Model [38]

The underlying DNA sequence $\vec{x}_G$and the set of reads are assumed to be generated in the following manner. Each entry of the underlying sequence $\vec{x}_G$is independently and identically distributed on the set of four alphabets $\{A, C, G, T\}$, with probabilities $\{p_1, p_2, p_3, p_4\}$ respectively. Here, $\min_{i=1}^{4} p_i \geq 0$ and $\sum_{i=1}^{4} p_i = 1$. The values of $\{p_1, p_2, p_3, p_4\}$ may or may not be known – our algorithm can handle both the cases. The $N$ reads are obtained by noiseless sampling from the long underlying sequence $\vec{x}_G$. The starting positions of the reads are unknown, and are assumed to be uniformly (and independently) distributed across the entire sequence $\vec{x}_G$of length $G$. Moreover, for simplicity of exposition, we assume that the reads may also be wrapped around $\vec{x}_G$, i.e., if the starting position of a read lies in the last $(L-1)$ bases of $\vec{x}_G$, then the $L$ bases of the read consist of the last few bases of $\vec{x}_G$followed by the first few bases of $\vec{x}_G$. Again, this condition of wrapping around is not fundamental to our algorithm, and only aids in the analysis. The reads are assumed to be free of errors or missing data. The setting is illustrated via an example in Fig.3.1. **The goal is to *exactly* recover the underlying sequence $\vec{x}_G$.**

### Fundamental bounds under the i.i.d. Model [38]

Motahari et al. [38] previously showed that for exact recovery of $\vec{x}_G$, there exists a tradeoff between the number of reads $N$ and the length $L$ of each read. In particular, in an asymptotic setting, they derived a precise characterization of the values of $(N,\ L)$ under which exact reconstruction is feasible, as described below.

The scheme for assembly provided in [38] is a greedy one. Under this scheme, the maximum overlap between every pair of reads is computed first. Pairs of reads are then merged in the decreasing order of overlaps. For this algorithm to fail, there are two possible sources of errors:

(a) Lack of coverage: There exists one or more bases in $\vec{x}_G$that are not covered by any of the $N$ reads.

ACTGACAGTGGA

(a) Original Sequence $\vec{x}_G$

GGAAC
CTGAC
ACAGT
AACTG
ACAGT
AGTGG

(b) Reads

Figure 3.1: Example illustrating the setting and notation of the model. (a) shows the original sequence $\vec{x}_G$of length $G = 12$, and (b) shows $N = 6$ reads, each of length $L = 5$. Observe how the reads $GGAAC$ and $AACTG$ wrap around $\vec{x}_G$. The sequence $\vec{x}_G$was generated with the base in each position being picked randomly and independently with a probability $p_1 = p_2 = p_3 = p_4 = 0.25$.The starting positions of each read in $\vec{x}_G$was picked uniformly at random from $\{1, \ldots, 12\}$.

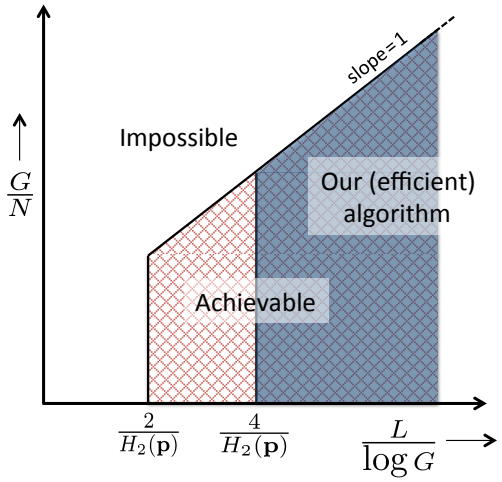



Figure 3.2: Plot showing the set of parameters achievable, impossible to decode for, and those which our algorithm operates under.

(b) Confounding due to repeats: Two reads, which do not overlap in $\vec{x}_G$, coincidentally have an overlap that is larger than their overlaps with the reads that are their true neighbours.

For instance, for the sequence depicted in Fig.3.1a, the set of reads $\{ACTGA, TGACA, CAGTG, AGTGG\}$ would cause an error due to lack of coverage since the last position is not covered by any of the reads. On the other hand, the reads $\{ACTGA, ACAGT, AGTGG, GAACT\}$ would confound the algorithm due to the repeats: the algorithm would append $GAACT$ at the end of $ACTGA$ (due to the overlap '$GA$'), while the correct merge would have been to append $ACAGT$ at the end of $ACTGA$ (they have an overlap of only '$A$').

It is shown in [38] that under the greedy algorithm, when

$$L \geq \frac{2}{H_2(p)} \ln G, \text{and} \tag{3.1}$$

$$N \geq \frac{G \ln G}{L}, \tag{3.2}$$

the probability of error decays exponentially with an increase in $G$. Here, $H_2(p)$ is the Renyi entropy of order 2 for the distribution $\{p_1,\ p_2,\ p_3,\ p_4\}$, and is given by

$$H_2(p) = -\ln \sum_{i=1}^{4} p_i^2. \tag{3.3}$$

When conditions(3.1) and(3.2) are satisfied, the probability of error under the greedy algorithm goes to zero asymptotically as $G \to \infty$. It is also shown in [38] that the conditions(3.1) and(3.2) are *necessary* for *any* algorithm to be able to correctly reconstruct the sequence $\vec{x}_G$.

The greedy algorithm of [38] described above is highly suboptimal in terms of space and computation efficiency. In particular, since it stores all the reads, it requires a space of $\Theta(NL) = \Theta(G \ln^2 G)$. Furthermore, the algorithm needs to make pairwise overlap computation for each pair of reads, thus resulting in a computation complexity lower bounded by $\Theta(N^2 L) = \Theta(G^2 \frac{\ln^2 G}{L})$. In the next section, we describe our algorithm that is efficient in terms of storage space and computation requirements, and is also optimal with respect to the number of reads required.

## 3.3 Main algorithm

The proposed algorithmic framework operates as follows. Depending on the (stochastic) model for the process, define a similarity metric between any two reads. Also find a threshold such that under the model considered, there is a vanishing probability of two non-adjacent reads having a similarity greater than that threshold. The first phase of the algorithm is an online phase, where two reads are merged whenever their similarity crosses the threshold. This phase thus allows the algorithm to require a small storage space, since all the reads now are not required to be stored separately. The second phase of the algorithm is an offline phase, where the remaining reads are merged in a greedy-yet efficient manner, exploiting the knowledge that no two of the remaining cotigs gave a similarity greater than the threshold. Finally, depending on the underlying model of errors, run a third 'consensus' phase that performs a consensus operation to obtain a final result from the scaffold construction (this third phase is not required in the absence of errors, or if missing data is the only form of errors). We now make this framework concrete by applying it to an i.i.d. model in the absence of errors.

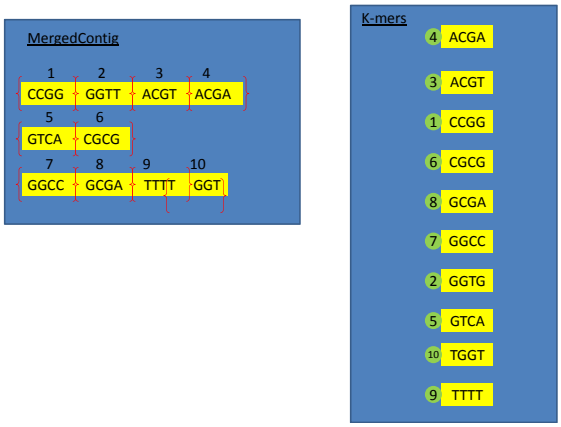



Figure 3.3: An example of MergedContig and K-mers data structures. Each row represents one distinct merged-contig/K-mer respectively. The entries of MergedContig are split into consecutive K-length segments represented via curly braces. These K-length segments are stored in the K-mers table in a sorted manner. Also depicted (through indices 1 to 10) is the one-to-one correspondence between the entries of the two tables, which are implemented via pointers both ways.

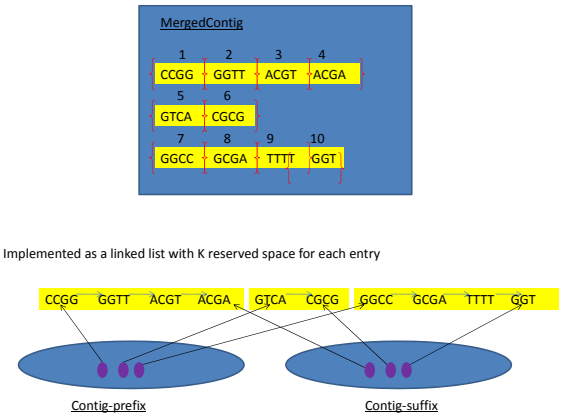



Figure 3.4: Implementation details of the MergedContig data structure. Each contig in the table is stored in the form of a linked list with each element of the list accommodating a sequence of length K, and each element having a pointer to the immediately succeeding length K sequence in that contig. Pointers to the beginning and end of each contig are stored separately. The set of contigs in the example are depicted on top of the figure, and its implemented as a linked list is depicted at the bottom of the figure. The parameters $L$ and $K$ take values 8 and 4 respectively in this example.

## Parameters

Our algorithm requires the read lengths to satisfy

$$L \geq \frac{4}{H_2(p)} \ln G. \tag{3.4}$$

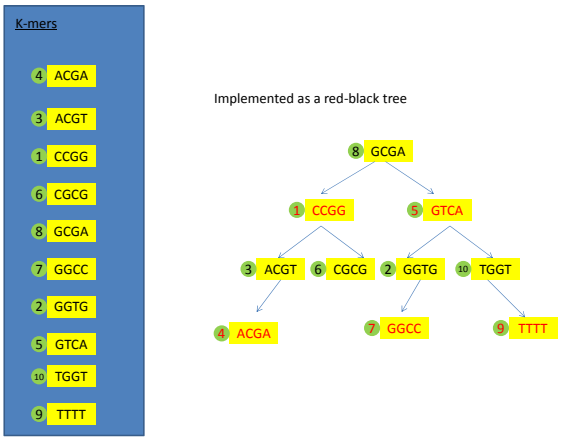



Figure 3.5: Implementation details of the of K-mers data structure. The K-mers are stored in a sorted manner as a red-black-tree. The list of K-mers in this example is displayed on the left, and the corresponding red-black-tree is depicted on the right of the figure. The parameters $L$ and $K$ take values 8 and 4 respectively in this example.

For any $L$ satisfying this condition, our algorithm is optimal with respect to the number of reads required for successful exact reconstruction, i.e., operates with

$$N = \frac{G \ln G}{L}. \tag{3.5}$$

Note that the value of $L$ required by our algorithm(3.4) is only a factor of 2 away from the lower bound(3.1). This may not be of a great concern in general, since for example, substituting $G = 10^{10}$ and assuming $p_i = 0.25 \forall i$, we get the requirement to be $L \geq 67$ which is certainly feasible with today's sequencing technologies. On the other hand, the optimality of $N$ (see(3.2) and(3.5)) is perhaps a significant advantage offered by this algorithm since this allows the sequencing to be carried out much faster.

Associated to our algorithm, we define an additional parameter $K$ as

$$K = \frac{2}{H_2(p)} \ln G. \tag{3.6}$$

This parameter shall be extensively employed in the algorithm.[1]

## Data structures employed

The algorithm employs two data-structures, 'MergedContig' and 'K-mers'. A high-level desription of these data structures is provided below; details will be clarified in the subsequent sections.

[1]The parameter K may also be chosen to have any higher value than3.6, and this will continue to possess theoretical correctness guarantees provided in Section3.4. The space and computation complexity of the algorithm remains the same (in an order sense) as long as K is chosen to be $\Theta(\ln G)$.

- **MergedContig**: This is a table that stores the set of contigs that have been merged at that point in time. The number of entries, and the length of each entry vary with time. To support these dynamics, the MergedContig table is implemented via a linked list.

- **K-mers**: This is a table storing a sorted list of entries, each of (equal) size K. The number of entries in K-mers changes over time, and the data structure needs to support efficient real-time search, (sorted) insertion and deletion. Thus, we implement K-mers as a self-balanced binary tree (e.g., a red-black tree).

The entries of the K-mers table are K-length substrings of the entries of "Merged-contig", and there are pointers pointing back and forth between the corresponding elements of the two data structures.

The MergedContig and K-mers data structures are initially empty.

The data structures are illustrated via an example in Figs.3.3,3.4 and3.5. In particular, Fig.3.3 depicts the relation between the MergedContig and the K-mers data structures, Fig.3.4 illustrates the implementation of MergedContig, and Fig.3.5 illustrates the implementation of K-mers. These illustrations are associated to the parameters $L = 8$ and $K = 4$.

## Algorithm

The main assembly algorithm (see Algorithm4) is divided into two phases: the first phase is an online phase and the second is an offline phase. At a higher level, in the online phase, reads with overlaps of $K$ or more contiguous positions are merged in real-time. (With respect to the general framework described previously, the distance metric is thus the amount of contiguous overlap and the threshold is K.) This phase thus allows the algorithm to require a small storage space, since all the reads now are not required to be stored separately. The comparisons and merges are made efficient by implementing the K-mers data structure as a red-black-tree, which enables fast searching, insertion and deletion. The offline phase merges the remaining set of contigs in a greedy-yet-efficient manner. This phase is made efficient by exploiting the fact that (following the online phase) no pair of contigs will have an overlap larger than K.

We also implement the following two subroutines that perform these tasks efficiently. Subroutine Combine() is used in the online phase for merging a new read with existing contigs. Subroutine Matching() is used in the offline phase for finding the best possible match (i.e., one with highest contiguous overlap) of a prefix contig among a set of suffix contigs, or vice versa. The precise algorithms of the two subroutines are provided in Algorithm5 and Algorithm6 respectively.

The operation of the online and offline phases of the main assembly algorithm are illustrated via an example in Figs.3.7 and3.8 respectively. These illustrations are associated to parameters $L = 4$ and $K = 2$, and assume an underlying sequence as $GCGTGGACCC$. The illustration of the online phase considers four newly arriving reads $GCGT$, $ACCC$, $GTGG$ and $GGAC$ (in that order).

**Algorithm 4** Main assembly algorithm

1. **Phase 1: online**
   Execute the following for every arrival of a new read $\vec{r}$ (of length $L$).

   a) Segment $\vec{r}$ into K-mers, i.e., all consecutive $K$-length subsequences (a total of $L - K + 1$ of them)

   b) Search each of these $(L - K + 1)$ subsequences in the K-mers table, and for each exact match, obtain the corresponding parent entry in "Merged-contig". There will be at-most two such parent entries in "Merged-contig" (see Fig.3.6).

      i. If there are no matching entries, then run subroutine Combine($\vec{r}$, null, new, null).

      ii. If there is exactly one matching entry $\vec{y}$ in "Merged-contig", and if $\vec{y}$ completely subsumes $\vec{r}$, then do nothing. Such a matching is depicted in Fig.3.6b.

      iii. If there is exactly one matching entry $\vec{y}$ in "Merged-contig", and if this entry has a partial overlap with $\vec{r}$, then (see Fig.3.6a)

         A. if prefix of $\vec{r}$ overlaps with suffix of $\vec{y}$, run subroutine Combine($\vec{r}$, $\vec{y}$, new, existing)

         B. if suffix of $\vec{r}$ overlaps with prefix of $\vec{y}$, run subroutine Combine($\vec{y}$, $\vec{r}$, existing, new).

      iv. If there are two matching entries, then $\vec{r}$ will have partial overlaps with both, as depicted in Fig.3.6c. Suppose the suffix of $\vec{r}$ overlaps with the prefix of $\vec{y}_1$ and the prefix of $\vec{r}$ overlaps with the suffix of $\vec{y}_2$, then run subroutine Combine($\vec{y}_1$, $\vec{y}_2$, existing, existing).

2. **Phase 2: offline**
   This phase merges all remaining contigs in MergedContig.

   a) Discard the K-mers table.

   b) Create two tables "pre-K-mer" and "suf-K-mer", both implemented in the same way as "K-mer", and populate them as follows (note that both these tables will be sorted).

   c) For each entry in MergedContig, store its K-length prefix as an entry in "pre-K-mer".

   d) For each entry in MergedContig, store its *reversed* K-length suffix as an entry in "suf-K-mer".

   e) If table "pre-K-mer" is empty, exit. Else, for the first entry $\vec{y}$ of "front-K-mer", do the following:

      i. $\vec{t_p} \leftarrow \vec{y}$ , $s_1 \leftarrow 0$, $s_2 \leftarrow -1$, $\vec{t_s} \leftarrow$ null

      ii. Execute subroutine Matching($\vec{t_p}$, suf, $s_2$). If "no match found" then go to (2b iv). Else, let $\vec{t_s}$ denote this best match, and let $s_1$ denote the overlap between $\vec{t_p}$ and $\vec{t_s}$, and continue to (2b iii).

      iii. Execute subroutine Matching($\vec{t_s}$, pre, $s_1$). If "no match found" then go to (2b iv). Else, let $\vec{t_p}$ denote this best match, and let $s_2$ denote the overlap between $\vec{t_s}$ and $\vec{t_p}$, and go back to (2b ii).

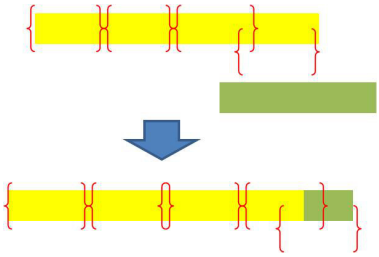
      iv. Merge the contigs corresponding to $\vec{t_p}$ and $\vec{t_s}$ in "MergedContigs". Delete $\vec{t_p}$ from "pre-K-mer" and $\vec{t_s}$ from "suf-K-mer". Go to (2e).

**Algorithm 5** Subroutine : Combine($\vec{t}$, $\vec{u}$, preType, sufType)

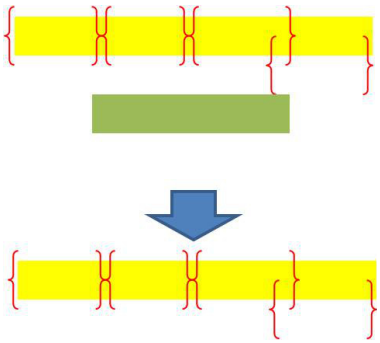
1. If preType=new and sufType=null then add $\vec{t}$ as a new entry in table "Merged-contig". Add the K-length prefix of $\vec{t}$ and the K-length suffix of $\vec{t}$ as two new entries in the K-mers table. Exit the subroutine.
2. If preType = existing, delete the K-mer corresponding to the prefix of $\vec{t}$ from K-mers
3. If sufType = existing, delete the K-mer corresponding to the suffix of $\vec{u}$ from K-mers
4. Merge the entries $\vec{t}$ and $\vec{u}$ in table "Merged-contig" as a single entry.
5. If a part of the merged entry is not included in K-mers (due to steps 2 and 3 of the subroutine, or if $\vec{t}$ or $\vec{u}$ is a new read), extract K-length substrings covering this part and add them as new entries into K-mers.

**Algorithm 6** Subroutine : Matching($\vec{t}$, searchTable, s)

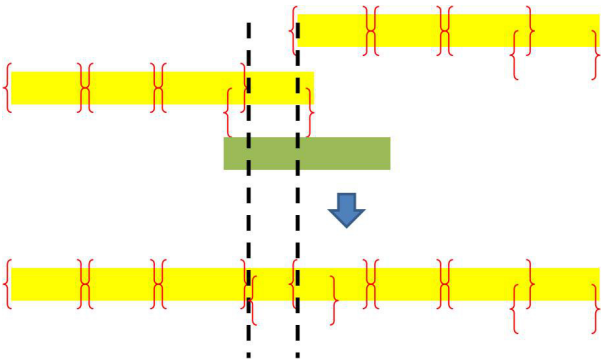
1. If searchTable=suf:
   a) For the $w = (\frac{L}{2} - 1)$ downto $(s + 1)$
      i. Let $\vec{v}$ be the first $w$ elements of $\vec{t}$. Reverse $\vec{v}$. Append $\vec{v}$ with $(K - w)$ zeros.
      ii. Search for $\vec{v}$ in "suf-K-mer".
      iii. If there is an entry in "suf-K-mer" that matches the first $w$ elements of $\vec{v}$, and that the parent (in MergedContig) of this entry is different from the parent of $\vec{t}$, then return this entry and the value of $w$. Exit the subroutine.
   b) Return "No match found"
2. If searchTable=pre:
   a) For the $w = (\frac{L}{2} - 1)$ downto $(s + 1)$
      i. Let $\vec{v}$ be the last $w$ elements of $\vec{t}$. Append $\vec{v}$ with $(K - w)$ zeros.
      ii. Search for $\vec{v}$ in "pre-K-mer".
      iii. If there is an entry in "pre-K-mer" that matches the first $w$ elements of $\vec{v}$, and that the parent (in MergedContig) of this entry is different from the parent of $\vec{t}$, then
   b) Return "No match found"



(a) New read matching the suffix an existing merged contig: append the new read at the end of the existing contig.



(b) New read completely overlapped by an existing merged contig: discard the new read since it is already covered.



(c) New read matching the prefix of one merged contig *and* the suffix of another merged contig: merge these two existing contigs.

Figure 3.6: Different ways a new read may match existing entries in MergedContig, and corresponding means of merging them. Existing reads in MergedConting are depicted yellow (lightly shaded when viewed in grayscale) and new reads in green (dark shaded when viewed in grayscale). Curly braces indicte the substrings that are entries of the K-mers table.

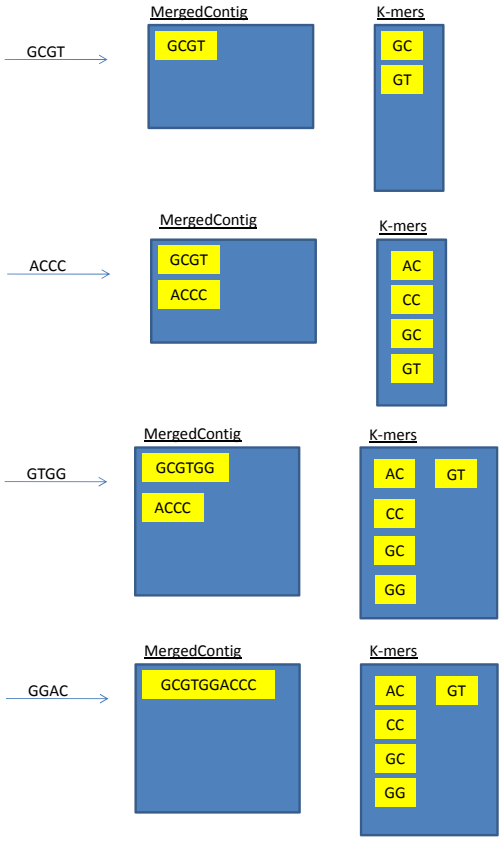



Figure 3.7: An example of the online phase of the main assembly algorithm, with $L = 4$ and $K = 2$. The four parts (from top to bottom) show the updates to the MergedContig and the K-mers tables under the algorithm upon arrival of four new reads.

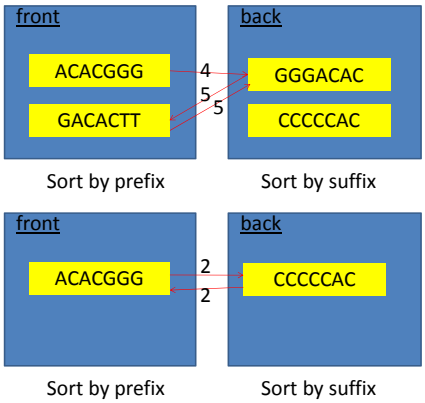



Figure 3.8: An example of the offline phase of the algorithm. The figures depicts two merges in this phase: the subfigure above shows one merge and the subfigure below shows the immediate next merge. The tables depicted are the pre-K-mer and the suf-K-mer tables (on the left and right respectively). The MergedContig table of the offline phase is not depicted here. Highest overlap counterparts of the contig being matched are indicated by an arrow.

## 3.4 Analysis

We analyse the performance of our algorithm in this section. Subsection3.4 describes simulations and experiments, while Subsections3.4,3.4 and3.4 provide a theoretical analysis of the algorithm.

To state the theoretical analysis in a nutshell, our algorithm is space-efficient in that it requires only $O(G)$ main memory and computation-efficient as it requires $O(G \ln^3 G)$ computations. The algorithm also minimizes the amount of data (i.e., the number of reads) required for successful reconstruction by achieving(3.5) which is (asymptotically) the minimum number of reads required for successful reconstruction of the sequence.

### Simulation and experiments

We have built a prototype of the assembler based on the algorithm provided, and the source code of the prototype has been made available. Using this prototype, we perform experiments to test the performance of the assembly algorithm. In particular, we consider an underlying DNA sequence of length $G = 1000$, and for various values of the read length $L$ and coverage $\frac{NL}{G}$, we plot the fraction of times the algorithm successfully reconstructed the underlying sequence exactly.[2] We experiment with three different underlying sequences: (a) a synthetic sequence generated from an i.i.d. uniform distribution at each base, (b) a segment of human chromosome 22 and (c) a segment of a bacteria genome Enterobacteria phage. The results are plotted in Fig.3.9.

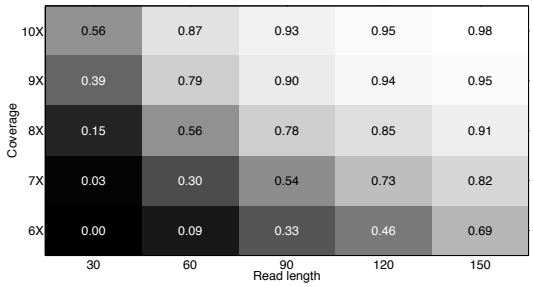
We remark that under the i.i.d. model with each base drawn from a uniform distribution, our algorithm requires (in theory, asymptotically) a coverage of at least $\frac{NL}{G} \geq 6.9078$ to entirely cover the underlying sequence and a read-length threshold of at least $L \geq 4\frac{\ln G}{\ln 2} = 39.8631$ for successful reconstruction. The empirical results of Fig.3.9 are fairly close: the genome is recovered correctly a reasonable fraction of times when $L \geq 60$ and coverage is $\geq 9$x. We reckon that the observed discrepancy is because the parameter $G = 1000$, and hence the algorithm is operating in a regime that is far from hitting asymptotic limits.[3] One can also observe from the plots that, as one would expect, the performance of the algorithm improves with an increase in coverage or with an increase in the read lengths.

### Theoretical correctness guarantees

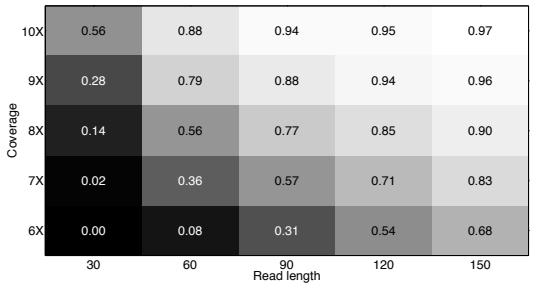
One can see that the algorithm described above will give a correct solution whenever the following three conditions are satisfied:

[2]For the simulations below, we employ the algorithm as in Algorithm4. However, the implementations of the individual data-structures have not been fine tuned yet (e.g., the K-mers table is implemented as an array instead of the more efficient red-black-tree). This preserves all the correctness guarantees of the algorithm, but does not provide as efficient a performance in terms of space and computation.

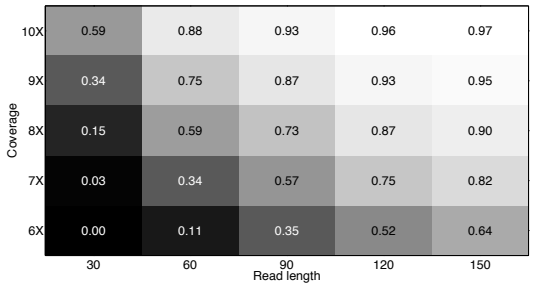
[3]We are also simulating the algorithm for much higher values of $G$, but the simulations are still running at the time of submission.




(a) An i.i.d. synthetic sequence




(b) First 1000 bases of human chromosome 22




(c) First 1000 bases of a bacteria genome Enterobacteria phage

Figure 3.9: Empirical variation of the proportion of instances of successful recovery with read length and coverage. The algorithm was run on three different underlying genomes, with the reads being sampled uniformly at random from the entire sequence. 500 simulations were performed for each triplet of (underlying genome, read length, coverage).

- coverage: every position in $\vec{x}_G$should be covered at-least once
- no-duplication: in $\vec{x}_G$, no sub-string of length $K$ should occur more than once
- no-confounding: for any read, the maximum overlap at its suffix (prefix) should be with the read that occurs next (previous) to it in $\vec{x}_G$.

The condition of no-duplication ensures that there are no errors in the merges performed in the online phase, the no-confounding condition prevents any incorrect merging in the offline phase, and the coverage condition ensures that each of the positions of $\vec{x}_G$are covered in the reads.

Under the i.i.d. model, one can show using [38, Theorem 1] that when $L \geq \frac{4}{H_2(\mathbf{p})} \ln G$ and when $N \geq \frac{G \ln G}{L}$, the first and the third conditions are satisfied with a probability that decays exponentially in the length $G$ of the underlying sequence. Furthermore, Theorem [38, Theorem 1] also ensures that with a high probability (decaying exponentially in $G$), there are no two identical substrings of length $\frac{2}{H_2(\mathbf{p})} \ln G$ in $\vec{x}_G$. Thus the conditions(3.4) and(3.5) guarantee successful reconstruction of $\vec{x}_G$. In particular, in the asymptotic setting as $G \to \infty$, the correctness is guaranteed with probability 1.

We note that our algorithm is guarantees the correct output (with probability 1) whenever the three conditions listed above (coverage, no duplication, no confounding) are satisfied. This makes the algorithm robust to the assumptions on $\vec{x}_G$as well as to the starting positions of the $N$ reads, as long as these three conditions are satisfied.

## Space complexity

We first analyze the space complexity (i.e., the storage space requirements) of the algorithm. The data-structures "MergedConting" and K-mers encompass all the storage space requirements under our algorithm.

We begin with an analysis of the MergedContig data structure, which is employed in both the online and offline phases. As we saw previously in Section3.4, since $K \geq \frac{2}{H_2(p)} \ln G$, all merge operations performed in the online phase are correct (with a high probability). Furthermore, observe that since $L = 2K$, any position in the underlying sequence $\vec{x}_G$can be a part of at-most two contigs in MergedContig (otherwise two of the contigs in MergedContig containing this position would have had an overlap of at least $K$, and would have been merged previously with each other). It follows that in MergedContig, every position appears at most twice, and hence the total size of the data stored is no more than $\leq 2G$ bases. This requires a storage space of at most $4G$ bits. Furthermore, it follows from a similar argument that the total number of contigs stored in MergedContig is at most $\frac{2G}{L}$. We implement MergedContig as a linked list, and the overhead caused by the pointers employed is no more than $\frac{2G}{L} \frac{\ln G}{\ln 2}$. For each entry of MergedContig, we also employ pointers to keep track of its prefix and suffix that are stored in K-mers, and this consumes a space of at most $\frac{2G}{L} \frac{\ln G}{\ln 2}$. From(3.4) we have $L \geq \frac{4}{H_2(\mathbf{p})} \ln G$ and hence the total space required to store these pointers is upper bounded by $\frac{3}{2} \frac{H_2(\mathbf{p})}{\ln 2} G$. Summing these items up, memory of size no larger than $(4 + \frac{3H_2(p)}{2 \ln 2})G$ bits is required for MergedContig.

Next we analyse the space requirements of the K-mers data-structure. The analysis encompasses all three tables using this data-structure, i.e., K-mers, pre-K-mers and suf-K-mers. Since K-mers is allowed to be deleted at the end of the online phase, the storage requirements for these tables is max(storage required by K-mers table, sum of storage required by

pre-K-mer and suf-K-mer tables). Thus, each of the parameters (e.g., storage requirement, number of entries) that are analysed below actually correspond to the max(parameter under K-mers table, sum of parameter under pre-K-mer and suf-K-mer tables). The aggregate length of all the entries in the table is upper bounded by $4G$, and hence storing this data requires a space no more than $8G$ bits. We also need pointers referring the entries of K-mers back to their respective parents in the MergedContig table. The number of entries in this table is at most $\frac{4G}{K}$, while the number of entries in MergedContig is at most $\frac{G}{K}$. Thus these pointers require a space of at most $\frac{4G \ln G}{K \ln 2}$ bits. We implement the K-mers data-structure as a self-balanced binary tree, and since the K-mers contains at most $\frac{4G}{K}$ entries, the pointers for this implementation require a space of at most $\frac{4G \ln G}{K \ln 2}$ bits. Summing up these quantities, and substituting the value of K from(3.6), we see that a memory of size no more than $\left(8 + \frac{4H_2(p)}{\ln 2}\right) G$ bits is required for the K-mers data-structure.

Thus the total space requirements for the algorithm is upper bounded by $\left(11 + \frac{5.5H_2(p)}{\ln 2}\right) G$ bits.

### Computational complexity

We now analyze the computational complexity of the algorithm. We start with an analysis of the complexity of the online phase. The online phase requires processing of $N = \frac{G \ln G}{L}$ reads. For each read, we obtain $O(L)$ number of K-mers. For each of these K-mers, an exact match is searched for in the K-mers table. Since this table is implemented as a red-black-tree, each search takes a worst case of $O(\ln \frac{G}{K})$ instances of matching two length $K$ strings. Thus the search complexity is $O(K \ln \frac{G}{K})$ for each K-mer. Furthermore, one needs to update the MergedContig and the K-mer tables, which also takes at most $O(K \ln G)$ computations. Aggregating these quantities and substituting the value of K from(3.6), we obtain that the online phase requires an aggregate computation upper bounded by $O(G \ln^3 G)$.

Next we analyse the computational-complexity of the offline phase. To this end, note that the total number of entries in MergedContig can be at most $O(\frac{G}{K})$, and hence the same also applies to the pre-K-mer and the suf-K-mer tables. Finding the best match in suf-K-mer for an entry of pre-K-mer takes $O(K \ln \frac{G}{K})$. Since there are $O(\frac{G}{K})$ entries in MergedContig, the subroutine of finding the best match is run a total of $O(K \frac{G}{K})$ times. Aggregating these quantities and substituting the value of K from(3.6), we obtain that the offline phase requires an aggregate computation upper bounded by $O(G \ln^3 G)$.

Thus the overall computational complexity is $O(G \ln^3 G)$.

## 3.5 Conclusion

In this chapter, we presented a new algorithm for de novo genome assembly that is efficient with respect to the storage space and computation requirements, and is optimal with respect to the number of reads required for reconstruction. The algorithm is based on an

i.i.d. generative model for the underlying DNA sequence. Our assembly algorithm operates in two phases, an online and an offline phase. The online phase merges reads with high overlaps on the fly, while the subsequent offline phase merges the remaining reads greedily. The operations in both phases are designed to be efficient in terms of space and time. In particular, by combining reads at run-time, the online phase significantly reduces the storage space requirements, which can also be exploited to achieve a much faster assembly by storing a greater proportion of the data in the fast-but-expensive main memory. This also serves as an algorithmic framework for designing algorithms in the future addressing various other scenarios and models. Furthermore, this also provides evidence of the feasibility of using the information-theoretic perspective for constructing practical algorithms for de novo assembly (and not merely for analysis and comparison of different algorithms).

# Chapter 4

# FinisherSC : A repeat-aware tool for upgrading de-novo assembly using long reads

We introduce FinisherSC, a repeat-aware and scalable tool for upgrading de-novo assembly using long reads. Experiments with real data suggest that FinisherSC can provide longer and higher quality contigs than existing tools while maintaining high concordance. The tool and data are available and will be maintained at http://kakitone.github.io/finishingTool/

## 4.1 Introduction

In de-novo assembly pipelines for long reads, reads are often trimmed or thrown away. Moreover, there is no evidence that state-of-the-art assembly pipelines are data-efficient. In this work, we ask whether state-of-the-art assembly pipelines for long reads have already used up all the available information from raw reads to construct assembly of the highest possible quality. To answer this question, we first collect output contigs from the HGAP [12] pipeline and the associated raw reads. Then, we pass them into our tool FinisherSC to see if higher quality assemblies can be consistently obtained after post-processing.

## 4.2 Methods

### Usage and pipeline

FinisherSC is designed to upgrade de-novo assembly using long reads (e.g. PacBio® reads). It is especially suitable for data consisting of a single long reads library. Input to FinisherSC are contigs (contigs.fasta) constructed by an assembler and all the raw reads with adaptors removed (raw_reads.fasta). Output of FinisherSC are upgraded contigs (improved3.fasta) which are expected to be of higher quality than its input (e.g. longer N50, longer longest

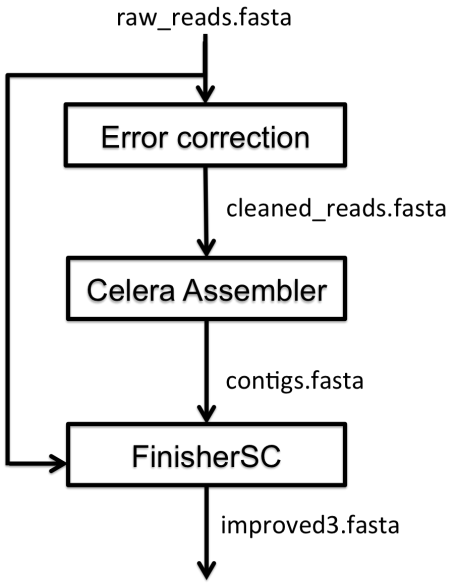



Figure 4.1: Pipeline where FinisherSC can fit in

contigs, fewer number of contigs, high percentage match with reference, high genome fraction, etc). In Fig 4.1, we show an example pipeline in which FinisherSC can fit. As shown in Fig 4.1, FinisherSC can be readily incorporated into state-of-the-art assembly pipelines (e.g. PacBio® HGAP).

## Algorithm and features

The algorithm of FinisherSC is summarized in Alg 7. Detailed description of the algorithm is in the supplementary materials. We summarize the key features of FinisherSC as follows.

- Repeat-aware: FinisherSC uses a repeat-aware rule to define overlap. It uses string graphs to capture overlap information and to handle repeats so that FinisherSC can robustly merge contigs. There is an optional component, X-phaser [29], that can resolve long approximate repeats with two copies by using the polymorphisms between them. There is also an optional component, T-solver, that can resolve tandem repeat by using the copy count information.

- Data-efficient: FinisherSC utilizes all the raw reads to perform re-layout. This can fill gaps and improve robustness in handling repeats.

- Scalable: FinisherSC streams raw reads to identify relevant reads for re-layout and refined analysis. MUMMER [26] does the core of the sequence alignment. Although MUMMER is single threaded, we provide an option to segment the files and run multi-

ple MUMMER jobs in parallel. These techniques allow FinisherSC to be easily scalable to high volume of data.

**Algorithm 7** Main flow of FinisherSC

**Input** : contigs.fasta, raw_reads.fasta
**Output**: improved3.fasta

1. Filter completely embedded contigs
2. Form a string graph with the BEST successors/predecessors as edges
3. Condense the string graph by contracting edges with both in-degree and out-degree being 1
4. Use raw reads to declare potential successors/predecessors of dangling contigs
5. Merge contigs (with gaps filled by reads) when they respectively only have 1 successor/1 predecessor
6. Form a string graph with ALL successors/predecessors as edges
7. Merge contigs with only 1 predecessor or 1 successor and each has no more than two competing edges

## 4.3 Results and discussion

### Experimental evaluation on bacterial genomes

We evaluated the performance of FinisherSC as follows. Raw reads were processed according to the pipeline in Fig 4.1. They were first error corrected and then assembled into contigs by an existing pipeline (i.e. HGAP). Contigs were upgraded using FinisherSC and evaluated for quality with Quast [19]. The data used for assessment are real PacBio® reads. These include data recently produced at JGI and data available online supporting the HGAP publication. We compared the assembly quality of the contigs coming out from the Celera assembler [42] of HGAP pipeline, the upgraded contigs by FinisherSC and the upgraded contigs by PBJelly [16]. A summary of the evaluation is shown in Fig 4.2. More details can be found in the supplementary materials. We find that FinisherSC can upgrade the assembly from HGAP without sacrifice on accuracy on these test cases. Moreover, the upgraded contigs by FinisherSC are generally of higher quality than those upgraded by PBJelly. This suggests that there is extra information from the reads that is not fully utilized by state-of-the-art assembly pipelines for long reads.

## Experiments on scalability

We tested the scalability of FinisherSC by applying it to handle larger genomes. The data used are the benchmark data available on PacBio Devnet. We run FinisherSC with the option of using 20 threads (-par 20) on a server computer. The server computer is equipped with 64 cores of CPU at clock rate of 2.4-3.3GHz and 512GB of RAM. The running time is tabulated in Table 4.1.

## Discussion

Although FinisherSC was originally designed to improve de-novo assembly by long reads, it can also be used to scaffold long contigs (formed by short reads) using long reads. For that use case, we note that the contigs formed by short reads can sometimes have length shorter than the average length of long reads. Therefore, we suggest users to filter out those short contigs before passing them into FinisherSC.

| Genome name | Genome size (Mbp) | Size of reads (Gbp) | Running time (hours) |
|---|---|---|---|
| Caenorhabditis elegans | 104 | 7.65 | 23 |
| Drosophila | 138 | 2.27 | 9.4 |
| Saccharomyces cerevisiae | 12.4 | 1.40 | 0.66 |

Table 4.1: Summary of running time for the experiments on scalability

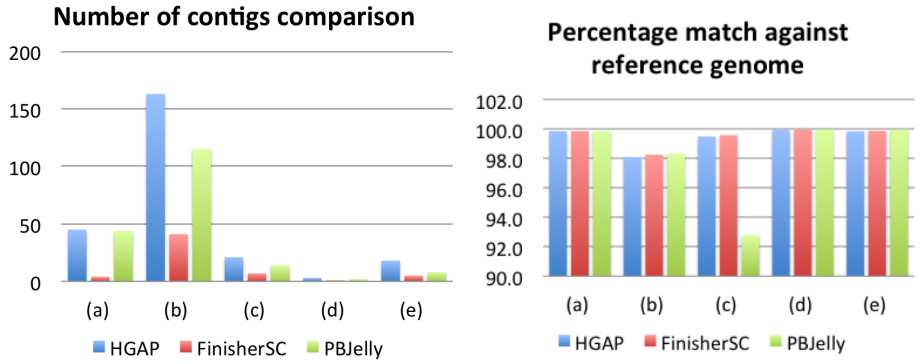



Figure 4.2: Experimental evaluation on bacterial genomes. (a,b) : Pedobacter heparinus DSM 2366 (PacBio® long reads from JGI) (c, d, e) : Escherichia coli MG 1655, Meiothermus ruber DSM 1279, Pedobacter heparinus DSM 2366 (PacBio® long reads supporting the HGAP publication).

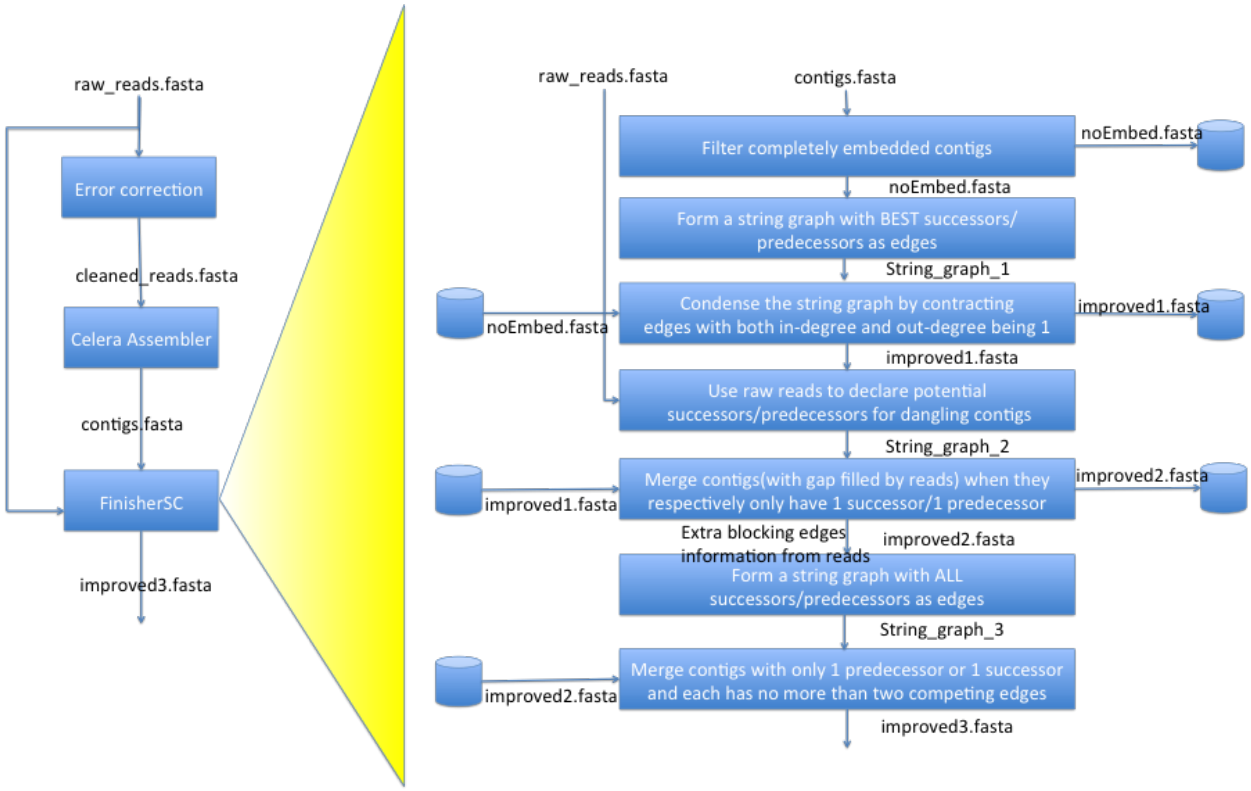



Figure 4.3: Summary of data flow of FinisherSC

## 4.4 Algorithm details, theoretical justification and more data analysis

### Summary of dataflow in FinisherSC and key experimental results

We visualize in Figure 4.3 the flow of FinisherSC. Moreover, we summarize the key experimental results in Table 4.2.

### Typical use cases

In this section, we describe example use cases of FinisherSC. Below are several scenarios that FinisherSC is helpful to you.

#### Low coverage data

There are many reasons that you end up having low coverage reads. You may want to save chemicals, the genome may be too long, some parts of the experimental setup may just malfunction or you do not want to overwhelm the assembler with huge amount of data. In

| Genome name | (a) | (b) | (c) | (d) | (e) |
|---|---|---|---|---|---|
| Genome size | 5167383 | 5167383 | 4639221 | 3097457 | 5167383 |
| Coverage of raw reads | 50 | 30 | 50 | 55 | 53.1 |
| Coverage of corrected reads | 32.93 | 17.74 | 10.1 | 11.3 | 9.5 |
| Coverage of input to Celera | 32.93 | 17.74 | 10.1 | 11.3 | 9.5 |
| N50 of HGAP output | 4097401 | 89239 | 392114 | 1053479 | 1403814 |
| N50 of FinisherSC upgraded output | 5168551 | 215810 | 1525398 | 3099349 | 2913716 |
| N50 of PBJelly upgraded output | 4099674 | 145441 | 1200847 | 1715191 | 3343452 |
| Number of contigs of HGAP output | 45 | 163 | 21 | 3 | 18 |
| Number of contigs of FinisherSC upgraded output | 4 | 41 | 7 | 1 | 5 |
| Number of contigs of PBJelly upgraded output | 44 | 115 | 14 | 2 | 8 |
| Length of the longest contig of HGAP output | 4097401 | 254277 | 1241016 | 1390744 | 2103385 |
| Length of the longest contig of FinisherSC upgraded output | 5168551 | 637485 | 2044060 | 3099349 | 2913716 |
| Length of the longest contig of PBJelly upgraded output | 4099674 | 495596 | 1958341 | 1715191 | 3343452 |
| Percentage match of HGAP output against reference | 99.87 | 98.11 | 99.51 | 99.96 | 99.85 |
| Percentage match of FinisherSC upgraded output against reference | 99.87 | 98.27 | 99.60 | 99.99 | 99.89 |
| Percentage match of PBJelly upgraded output against reference | 99.86 | 98.34 | 92.77 | 99.98 | 99.97 |
| Total length of HGAP output | 5340498 | 5536634 | 4689701 | 3102769 | 5184825 |
| Total length of FinisherSC upgraded output | 5212355 | 5139491 | 4660679 | 3099349 | 5167414 |
| Total length of PBJelly upgraded output | 5383836 | 5821106 | 4718818 | 3106774 | 5210862 |

Table 4.2: Experimental evaluation results. (a,b) : Pedobacter heparinus DSM 2366 (recent real long reads from JGI) (c, d, e) : Escherichia coli MG 1655, Meiothermus ruber DSM 1279, Pedobacter heparinus DSM 2366 (real long reads supporting the HGAP publication). Detailed analysis by Quast is shown in the supplementary material.

any of these situations, you want to utilize as much information from the reads as possible because of the scarcity of read data.

**Simple setup for assemblers**

There are normally a lot of parameters that can be tuned for modern assemblers. It is also often not clear what parameters work best for your data. However, you do not want to waste time in repeatedly running the assembler by varying different combinations of parameters/setting. In this case, you need a tool that can efficiently and automatically improve your assemblies from the raw reads without rerunning the assembler.

**Scaffolding**

You may have long contigs prepared from one library and long reads prepared from the other. In this case, you want to robustly and seamlessly combine data from two libraries through scaffolding.

## Instructions on using FinisherSC

Our software, FinisherSC, is helpful for the use cases discussed above. It processes long contigs with long reads. You only need to supply the input data files and issue a one-line

command as follows to perform the processing. Let us assume that mumP is the path to your MUMMER and destP is the location where the input and output files stay.

- Input : raw_reads.fasta, contigs.fasta
- Output : improved3.fasta
- Command :

```
python finisherSC.py destP/ mumP/
```

We provide a sandbox example linked in our webpage. Besides the standard usage, there are extra options with details in our webpage.Specifically, we note that users can run FinisherSC in parallel by using the option of [-par numberOfThreads].

## Detailed description of the algorithm

We adopt the terminology in [29]. Random flanking region refers to the neighborhood of a repeat interior. A copy of a repeat being bridged means that some reads cover the copy into the random flanking region. Subroutine 1 removes embedded contigs that would otherwise confuse the later string graph operations. Subroutines 2, 3, 6, 7 are designed to handle repeats. Subroutines 2, 3 resolve repeats whose copies are all bridged by some reads. Subroutines 6, 7 resolve two-copies repeats of which only one copy is bridged. Subroutines 4, 5 utilize neglected information from raw reads. They define merges at locations which are not parts of any long repeats. [1]

**Algorithm 8** Subroutine 1: Filter completely embedded contigs

**Input** : contigs.fasta
**Output**: noEmbed.fasta

1. Obtain alignment among contigs from contigs.fasta
2. For any (x,y) contig pair, if x is completely embedded in y, then we add x to removeList
3. Remove all contigs specified in removetList from contigs.fasta. The resultant set of contigs are outputted as noEmbed.fasta

[1] To simplify discussion, the subroutines described are based on the assumption that reads are extracted from a single-stranded DNA. However, we remark that we have implemented FinisherSC by taking into account that reads are extracted from both forward and reverse strands.

**Algorithm 9** Subroutine 2: Form a string graph with the BEST successors/predecessors as edges

**Input** : noEmbed.fasta
**Output**: String_graph_1

1. Initialize the nodes of G to be contigs from noEmbed.fasta
2. Obtain alignment among contigs from noEmbed.fasta
3. **for** *each contig x* **do**
   Find predecessor y and successor z with the largest overlap with x
   **if** *such y exists* **then**
   add an edge $y \rightarrow x$ to G
   **end**
   **if** *such z exists* **then**
   add an edge $x \rightarrow z$ to G
   **end**
   **end**
4. Output G as String_graph_1

**Algorithm 10** Subroutine 3: Condense the string graph by contracting edges with both in-degree and out-degree being 1

**Input** : String_graph_1, noEmbed.fasta
**Output**: improved1.fasta

1. **for** *each edge* $u \rightarrow v$ *in String_graph_1,* **do**
   **if** *out-deg (u) = in-deg (v) =1* **then**
   condense (u,v) into a single node and concatenate the node labels
   **end**
   **end**
2. **for** *each node x in the transformed String_graph_1* **do**
   output the concatenation of contigs associated with node x to be a merged contig
   **end**
3. Output all the merged contigs as improved1.fasta

**Algorithm 11** Subroutine 4: Use raw reads to declare potential successors/predecessors of dangling contigs

**Input** : improved1.fasta, raw_reads.fasta
**Output**: String_graph_2

1. Initialize nodes of G to be contigs from improved1.fasta
2. Divide raw_reads into batches $B_S$
3. Stream the data in $B_S$.

   **for** $b \in B_S$ **do**
   i) align b with contigs from improved1.fasta
   ii) record the overlap information in $I$
   **end**

4. **for** *each pair of nodes* $u, v$ *in* $G$ **do**
   **if** $u \rightarrow v$ *is a predecessor-successor pair* **then**
   Add an edge $u \rightarrow v$ to G
   **end**
   **if** *there exists read* $R$ *such that* $(u, R)$ *and* $(R, v)$ *are predecessor-successor pairs according to* $I$ **then**
   Add an edge $u \rightarrow v$ to G
   **end**
   **end**

5. Output graph G as String_graph_2

**Algorithm 12** Subroutine 5: Merge contigs (with gaps filled by reads) when they respectively only have 1 successor/1 predecessor

**Input** : improved1.fasta, String_graph_2
**Output**: improved2.fasta, connectivity_info

1. **for** *each edge $u \rightarrow v$ of String_graph_2* **do**
   **if** *out-deg (u) = in-deg (v) =1* **then**
   condense (u,v) into a single node and concatenate the node labels
   **end**
   **end**
2. **for** *each node in the transformed String_graph_2* **do**
   output concatenated contigs as new contigs (with reads filling the gaps) and connectivity information to connectivity_info
   **end**

**Algorithm 13** Subroutine 6: Form a string graph with ALL successors/predecessors as edges

**Input** : improved2.fasta, connectivity_info
**Output**: String_graph_3

1. Use connectivity_info to form a graph G with nodes from improved2.fasta. All predecessor-successor pairs are edges in G.
2. Output the corresponding graph as String_graph_3

**Algorithm 14** Subroutine 7: Merge contigs with only 1 predecessor or 1 successor and each has no more than two competing edges

**Input** : improved2.fasta, String_graph_3
**Output**: improved3.fasta

1. Traverse the String_graph_3 for pattern of $u1 \rightarrow u3$, $u2 \rightarrow u3$ , $u2 \rightarrow u4$ and that out-deg (u1) =1, out-deg (u2) = 2, in-deg (u3) =2, in-deg (u4) =1, if found, then,
   a) Delete the edge $u2 \rightarrow u3$
   b) Condense the graph
   c) Continue until the whole graph is traversed
2. Output the merged contigs as improved3.fasta

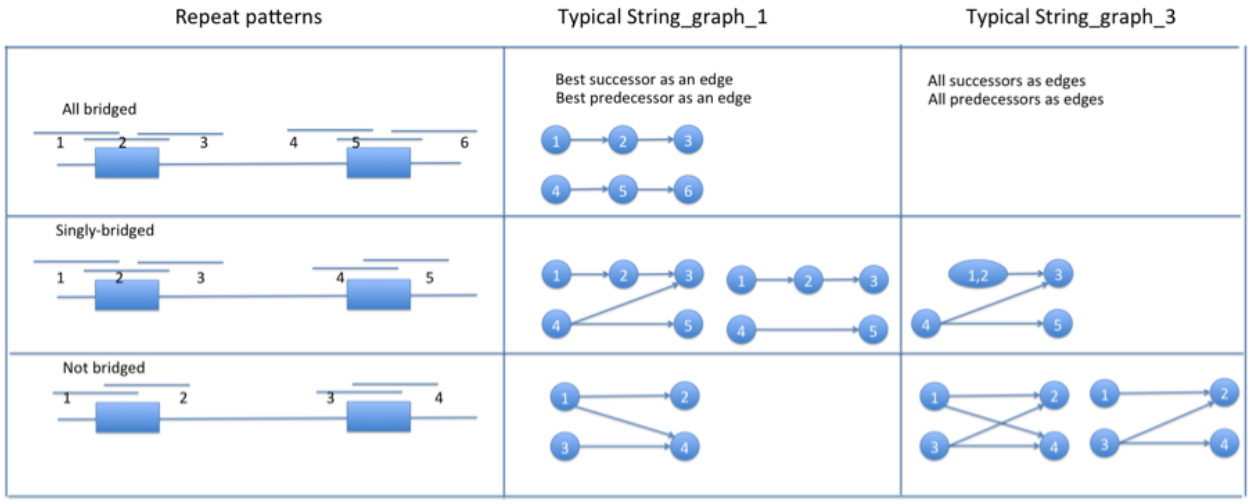



Figure 4.4: Repeat patterns, typical String_graph_1, typical String_graph_3

## Justification of the algorithm

### Big picture

There are two main parts of the algorithm underlying FinisherSC. They are

1. Gap filling
2. Repeat resolution

With uniform sampling assumption, the gaps are unlikely to land on the few long repeats on bacterial genomes. Therefore, subroutines 4, 5 can close most of the gaps. For repeat resolution, subroutines 1, 2, 3, 6, 7 robustly define merges using various transformations of string graphs. Detailed discussion is in the coming section.

**Detailed justification on repeat resolution**

We focus the discussion on a long repeat with two copies. To simplify discussion, we further assume that each base of the genome is covered by some reads and the read length is fixed. The goal here is to correctly merge as many reads as possible in the presence of that repeat. The claim is that Subroutines 2, 3, 6, 7 collectively can achieve this goal. In the case of one repeat, we only need consider the reads either bridging the repeat copies/ reads at the interior of repeats/ touching the repeat copies of that repeat. We separate the discussion on each of the cases depicted in the rows of Fig 4.4. They are listed as follows.

1. Both copies are bridged
2. Only one copy is bridged
3. Both copies are not bridged

Let us first clarify some terminologies before proceeding. A read y is called a successor of another read x if they have significant head-to-tail overlap in the order of $x \rightarrow y$. y is called the best successor of x if the overlap length is the largest among all the successors of x. y is called the true immediate successor of x if y is the closest read to x's right side in the sequencing process. Similarly, we can also define the corresponding notion for predecessors.

In the first case, without loss of generality, let us consider any read R emerging from the left flanking region of the left copy. It will get merged with its best successor when condensing String_graph_1. Moreover, the best successor is also the true immediate successor. It is because reads from the other copy of the repeat either have smaller overlap length or are not successors.

Now, let us move to the second case. Since there is a bridging read, there are no reads completely embedded in the interior of the repeat. Without loss of generality, we consider the case that the left copy is bridged and the right copy is not. Now we label R2 as the bridging read, R1/R3 respectively as the true immediate predecessor/successor of the bridging read, R4/R5 as the most penetrating reads into the second copy of the repeat. For all other reads, they get merged with their true immediate successors/predecessors when condensing in String_graph_1. For the remaining five items of interest, the main question is whether there is an edge between R4 and R5 in String_graph_1 (i.e. whether the best successor of R4 is R3). If not, then condensing in String_graph_1 will merge R4 with R5, which is the true immediate successor. If such an edge exists, then we end up with the pattern shown in Fig 4.4 for String_graph_3. This means that only R1 is merged to R2 when condensing String_graph_1. However, given the existence of the Z-shape pattern, graph operations on String_graph_3, the subroutine 7 will merge R2 and R3, and also will merge R4 and R5.

Finally, consider the third case, when both repeat copies are not bridged. For reads that are not closest to the repeat copies, they get merged correctly when condensing String_graph_1. Without loss of generality, we consider a read x closest to the left flanking region of the left copy of the repeat. An illustration of this situation in String_graph_1 is shown in Fig 4.5. Let its true immediate successor be T. We are going to show that it will not get merged with

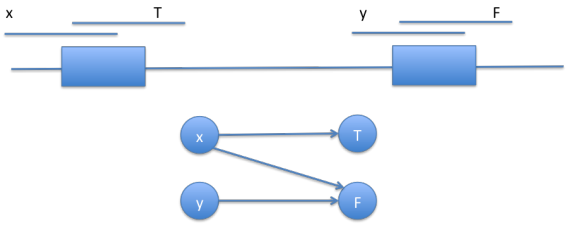



Figure 4.5: Z-pattern in string graph

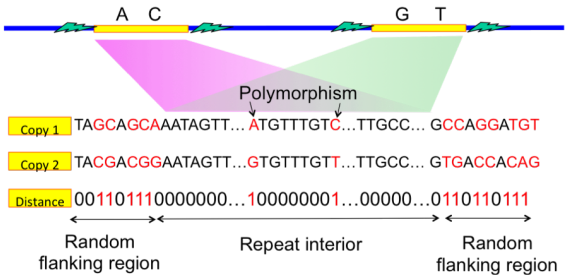



Figure 4.6: Approximate repeat with two copies

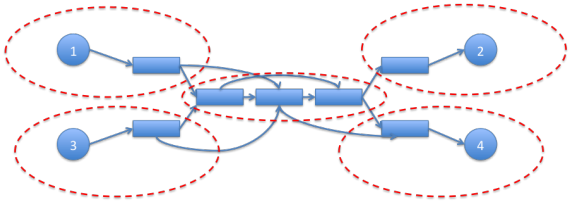



Figure 4.7: Using string graph to define repeat interiors and flanking regions

the wrong read in String_graph_1 through a proof by contradiction. If x got merged with some wrong F, then $x \rightarrow F$ would be an edge. Let y be the read closest the left flanking region of the right copy of the repeat. Then, $y \rightarrow F$ is also an edge. Therefore, there should be no merges of $x \rightarrow F$, which results in contradiction. Now we consider String_graph_3, if x has only 1 successor, then it should be T. Otherwise, it is connected to both T and some F. Then, we consider the y coming from the left flanking region of the right copy. There must be an edge from y to F. If there is also an edge from y to T, then both x and y are not merged in String_graph_3. However, if there is no edge from y to T, then x is merged with T and y with F correctly.

## Approximate repeat phasing option

FinisherSC provides an optional component, X-phaser to resolve approximate repeat with two copies, which cannot be bridged by any reads. An example of such an approximate repeat is shown in Fig 4.6. The algorithm behind X-phaser involves two main parts.

Figure 4.8: Pattern of back-to-back tandem repeat

Figure 4.9: Pattern of string graph that corresponds to a typical back-to-back tandem repeat

1. Identify repeat interior and its flanking regions
2. Merge contigs by phasing the polymorphisms within the repeat

Algorithm 15 achieves the first part by performing various operations on a string graph. The nodes of the string graph are either contigs or reads near the end points of the contigs. An illustration of a typical string graph is shown in Fig 4.7. The contigs are indicated by solid circles and reads are indicated by rectangles. The dotted circles specify the random flanking region and repeat interior that we want to infer through Algorithm 15. The X-phasing step in [29] achieves the second part. It utilizes the polymorphisms within the repeat interior to help distinguish the repeat copies. Interested readers may refer to Xphased-Multibridging in [29] for more details. In FinisherSC, we use the implementation of the Xphasing step in [29].

## Tandem repeat resolution option

The optional tandem repeat resolution step can resolve back-to-back tandem repeat. A back-to-back tandem repeat refers to repeat whose copies directly follow one another. An example is given in Fig 4.8. FinisherSC provides a component, T-solver, to resolve that. The key idea is to detect cycles in the string graph that join the reads and contigs. An illustration of such a graph is shown in Fig 4.9, where the circles are contigs and rectangles are reads associated with the end points of the contigs. The algorithm behind T-solver is in Alg 16.

**Algorithm 15** Repeat phasing option

**Input** : improved3.fasta, raw_read.fasta
**Output**: improved4.fasta

1. Stream reads to identify reads that are associated with end points of contigs

2. Form a 2-color graph with black nodes as reads and red nodes as contigs. Name it as G

3. **for** *each red node in G* **do**
   Perform graph search to determine other red nodes that are reachable by it through path consisting of black nodes only
   **end**

4. Form a bipartite graph $B = (B_L, B_R)$ with nodes on left/right side representing left/right ends of the contigs. An edge $L \rightarrow R$ exists, where $L \in B_L, R \in B_R$, if there exists a path of black nodes in G such that $R$ is reachable from $L$

5. Find connected components in $B$

6. **for** *each connected component at* $b \in B$ **do**
   **if** $|b \cap B_L| = 2$ *and* $|b \cap B_R| = 2$ **then**
   define b as a two-copies repeat
   **end**
   **end**

7. **for** *each two-copies repeat[ with associated nodes (L1,L2; R1, R2) ]* **do**
   a) Go back to graph G, and label nodes reachable from each of L1/L2 and to each of R1/R2 through black paths.
   b) For each black node in G connected to all L1/L2/R1/R2, name it as inside nodes. If it only misses one of the four, then label it as miss_x node where x is the missed item
   c) Define start node S as an inside node connected to some miss_L1 nodes and miss_L2 nodes. Similarly, define end node E as an inside node connected to some miss_R1 nodes and miss_R2 nodes
   d) Define repeat interior and flanking region
   i) Find a black path between S and E and label it as a repeat path (this is the repeat interior)
   ii) Find paths from L1 to S, L2 to S , E to R1, E to R2 respectively (these are the flanking regions)
   e) For each black node involved with this repeat,
   i) If it is connected to nodes on paths L1 to S, add it to L1 to S read set. Similarly, do it for L2 to S, E to R1 and E to R2 read sets
   ii) If it is connected to inside nodes only, then add it to repeat read set
   iii) Use the separated read sets to perform repeat phasing as described in [29]
   iv) Declare merging of contigs based on the phasing results
   **end**

**Algorithm 16** Tandem repeat resolution option

**Input** : improved3.fasta, raw_read.fasta
**Output**: tandem_resolved.fasta

1. Classify contigs into two clusters in which one contains long contigs ($S_L$) and the other contains short contigs($S_S$). We only focus on the treatment on $S_L$.

2. Form a 2-color read-contig string graph $G$ with contigs from $S_L$ and raw reads associated to the endpoints of these contigs. This step is similar to step 2 in Alg 15

3. **repeat**

   a) Identify local graph pattern in $G$ consisting of 1 incoming contig, 1 outgoing contig and cycles linking intermediate reads. Declare that as a tandem repeat target $t$.

   b) For such target $t$, find the repetitive pattern of the tandem repeat through finding a cycle on the intermediate reads.

   c) Estimate the copy count of the repetitive pattern by using overall coverage and amount of reads associated with $t$.

   d) Connect the contigs associated with $t$ with appropriate tandem repeat copies filled in.

   **until** *there is no back-to-back tandem repeat pattern detected in* $G$;

## 4.5 Conclusion

In this chapter, we introduce FinisherSC which improves assembly quality for haploid genome assembly with long read only data. The key idea is to reuse the original data to perform scaffolding in a repeat-aware manner. FinisherSC has shown significant quality improvement on assembly quality for haploid genome assembly on all the tested datasets. Moreover, it has shown better performance when compared to another state-of-the-art long read scaffolding tool, PBJelly. We note that FinisherSC takes the post-processing approach and thus is much simpler to implement than a brand-new assembler. This thus serves as an evidence that post-processing is a good technique to bring sophisticated techniques to assembler building process. Because of its great potential, we further explore the post-processing approach in the coming two chapters by studying more complex cases involving metagenomes and hybrid data.

# Chapter 5

# BIGMAC : Breaking Inaccurate Genomes and Merging Assembled Contigs for long read metagenomic assembly

The problem of de-novo assembly for metagenomes using only long reads is gaining attention. We study whether post-processing metagenomic assemblies with the original input long reads can result in quality improvement. Previous approaches have focused on pre-processing reads and optimizing assemblers. BIGMAC takes an alternative perspective to focus on the post-processing step. Using both the assembled contigs and original long reads as input, BIGMAC first breaks the contigs at potentially mis-assembled locations and subsequently scaffolds contigs. Our experiments on metagenomes assembled from long reads show that BIGMAC can improve assembly quality by reducing the number of mis-assemblies while maintaining/increasing N50 and N75. The software is available at https://github.com/kakitone/BIGMAC

## 5.1 Introduction

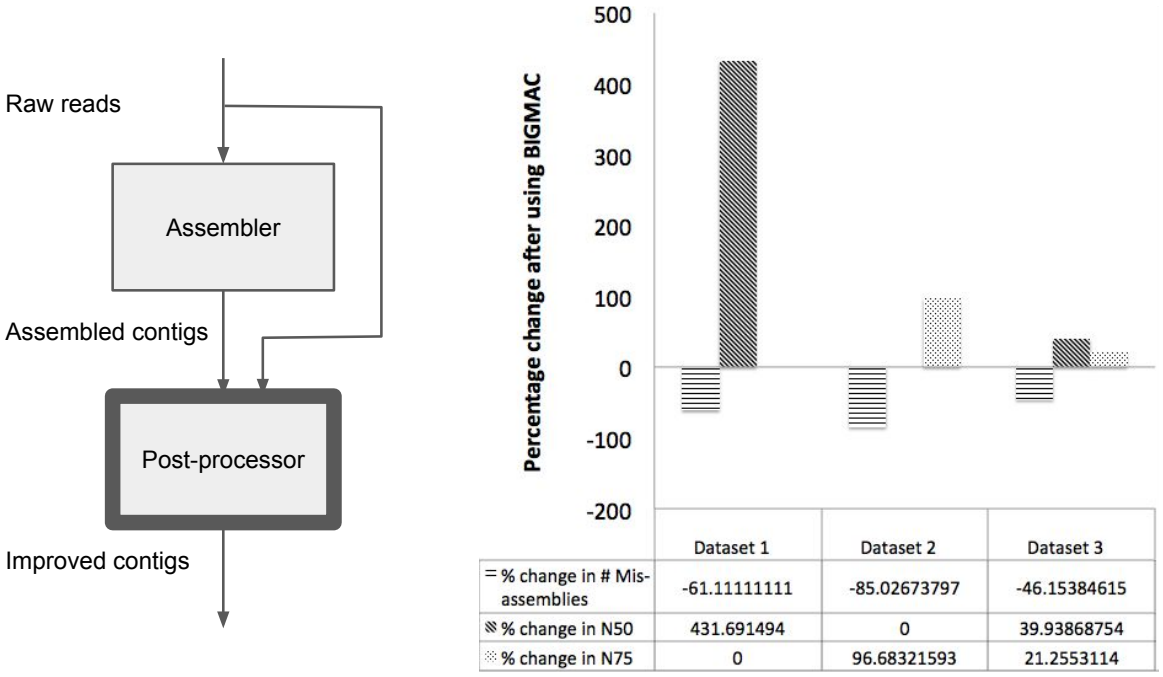
De-novo assembly is a fundamental yet difficult [11] computational problem in metagenomics. Indeed, there is currently an open challenge for metagenomic assembly using short reads, titled "Critical Assessment of Metagenomic Interpretation (CAMI [59])." On the other hand, emerging sequencing technologies can provide extra information and make the computational problem more tractable. For example, long reads are increasingly being shown to be valuable in the de-novo assembly of single genomes[24]. Therefore, it is natural to study metagenomic assembly using long reads. Current assembly approaches for long reads focus on developing more optimized assemblers to leverage the power of the data. However, significant engineering effort is usually involved to build an end-to-end assembler.

We take a different perspective, focusing the design effort on a post-processor that im-

proves assembled contigs using the original long read data (Fig 5.1). The main question is whether we can achieve quality improvement with this approach using typical long reads. This post-pocessing approach is attractive because it leverages existing software. Consequently, we can focus design effort and computational resources to specifically address thorny issues arising from the nature of new data, complex repeats, varying abundances and noise. Moreover, since the long reads are part of the input, the post-processor can bring back information missed upstream. In single genome assembly, FinisherSC [30] has demonstrated the effectiveness of this approach. In this paper, we investigate the effectiveness of this post-processing approach for metagenomic assembly.

BIGMAC is a post-processor to improve metagenomic assemblies with long read only data. It first breaks contigs at potentially mis-assembled locations and subsequently scaffolds contigs. In our experiments, BIGMAC demonstrates promising results on several mock communities using data from the Pacific Biosciences long read sequencer. Inputs to BIGMAC include assembled contigs from HGAP [12] and the original raw reads with adapters removed. After assembly and post-processing, we use QUAST [19] to evaluate and compare the assembly quality of contigs before and after using BIGMAC. As shown in Fig 5.1, BIGMAC improves the quality of the contigs by reducing the number of mis-assemblies while maintaining/increasing N50 and N75. This demonstrates the effectiveness of the post-processing approach for metagenomic assembly with long reads.

Figure 5.1: Position of post-processor in an assembly pipeline (left). Improvement in assembly quality using post-processor BIGMAC on three different datasets (right). BIGMAC shows the effectiveness of the post-processing approach for long read metagenomic assembly.

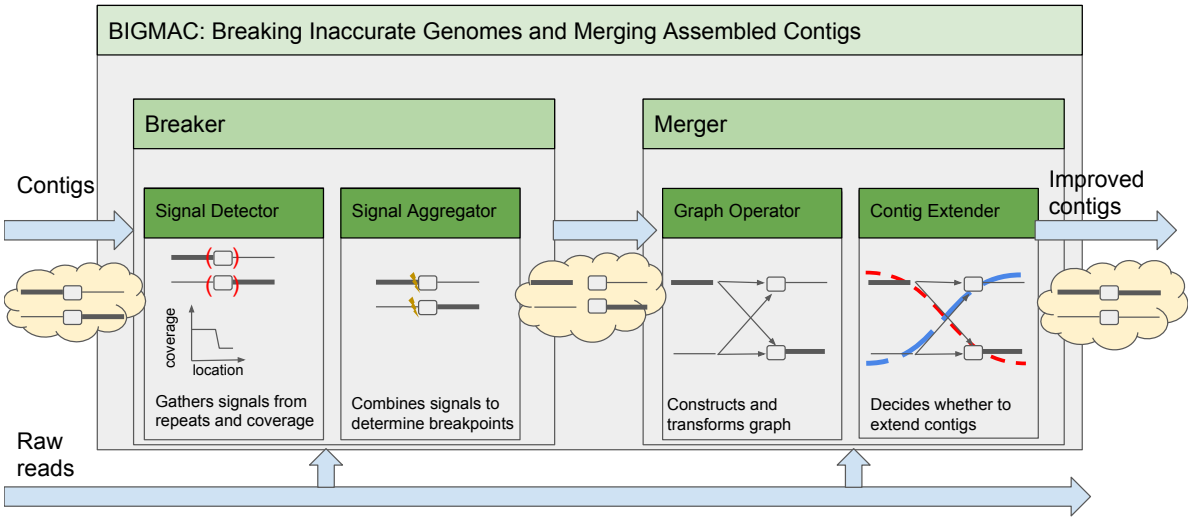



## 5.2 A top-down design of BIGMAC

We use a hypothetical yet representative set of input data to illustrate the design of BIGMAC in a top-down manner. Let $g_1, g_2$ be two genomes of abundances $\lambda_1, \lambda_2$ respectively. Assume that they share a long repeat in the middle, that is, $g_1 = x_1ry_1, g_2 = x_2ry_2$. Unfortunately, an upstream assembler mis-assembles the reads and produces two contigs $c_1, c_2$ with incorrect joins at the repeat. That is, $c_1 = x_1ry_2, c_2 = x_2ry_1$. As an assembly post-processor, BIGMAC takes the mis-assembled contigs $c_1, c_2$ and original reads as input. Its goal is to reproduce $g_1, g_2$. To achieve this, we immediately recognize that there should be components for fixing mis-assemblies and scaffolding contigs. In BIGMAC, they are respectively Breaker and Merger. An illustration is given in Fig 5.2.

Breaker is further divided into two subcomponents: Signal Detector and Signal Aggregator. Signal Detector collects signals that indicates mis-assemblies and Signal Aggregator subsequently makes decisions on breaking contigs based on the collected signals. In our example, the coverage fluctuation between $\lambda_1, \lambda_2$ along the contigs $c_1, c_2$ and the presence of long repeat $r$ are useful signals that Signal Detector collects. After aggregating these signals, Signal Aggregator decides on breaking both the contigs $c_1$ and $c_2$ at the starting points of

Figure 5.2: Top-down design of BIGMAC with an example of how BIGMAC improves a pair of mis-assembled contigs.




the repeat $r$. Therefore, the output contigs of Breaker are $x_1, x_2, ry_1, ry_2$.

Merger is also divided into two subcomponents: Graph Operator and Contig Extender. With information from the original reads, Graph Operator establishes connectivity of the input contigs using string graphs. Then, based on the evidence from spanning reads and contig coverage, Contig Extender extends input contigs into longer contigs. In our example, the input contigs to Merger are $x_1, x_2, ry_1, ry_2$. Graph Operator forms a string graph with edges $x_1 \to ry_1, x_1 \to ry_2, x_2 \to ry_1$ and $x_2 \to ry_2$. Contig Extender analyzes the coverage depth of the related contigs and decides to merge contigs to form $x_1ry_1$ and $x_2ry_2$, thus reproducing the correct genomes.

## 5.3 Breaker: Breaking Inaccurate Genome

After the functional decomposition of BIGMAC in the previous section, we are ready to investigate its first component: Breaker. We note that the goal of Breaker is to fix mis-assemblies. In order to achieve that, we need to collect sensible signals related to mis-assemblies and subsequently aggregate the signals to make the contig breaking decisions.

### Signal Detector

Signal Detector collects three important signals related to mis-assemblies.

**Palindrome:** We are interested in palindromes because of their relationship to a form of chimeric reads, the adaptor-skipped reads, which are common in today's long read technology. Since assemblers get stuck at these chimeric reads, the palindrome pattern in reads prop-

agates to the corresponding contigs. Thus, the pattern of palindrome is a strong signal indicating mis-assemblies, particularly when the palindrome is long. A string tuple $(a, b)$ is called a wrapping pair if the reverse complement of $a$ is a prefix of $b$ or the reverse complement of $b$ is a suffix of $a$. $x$ is called a palindrome if it is the concatenation of a wrapping pair $(a, b)$, that is $x = ab$. The wrapping length of $x$ is $\max_{x=ab,(a,b) \text{ is a wrapping pair}} \min(a.length, b.length)$. For example, $x = ACGGCCG$ is a palindrome of wrapping length 3; $(a, b) = (ACGG, CCG)$ is a wrapping pair because the reverse complement of $b$ is $CGG$, which is a suffix of $a$. Since the minimum length of $a$ and $b$ is $\min(4, 3) = 3$ and the wrapping length of $x$ cannot exceed 3, the wrapping length for $x$ is 3.

Signal Detector collects information about palindromes by aligning each contig against itself. It then identifies palindromes with long wrapping length because that indicates mis-assemblies. The corresponding palindromes' information is then put into $S_{palindrome}$. To improve sensitivity, Signal Detector allows approximate match when searching for palindromes.

**Repeat:** Since long repeats confuse assemblers, their endpoints are possible candidates for mis-assemblies. Let $s_1 = axb, s_2 = cxd$, a repeat between $s_1, s_2$ is specified by the endpoints of $x$ in $s_1, s_2$. For example, $s_1 = CAAAAT, s_2 = GAAAAG$, the endpoints of the repeat $AAAA$ are the position specified by ! in $C!AAAA!T, G!AAAA!G$. Signal Detector aligns contigs against themselves to find the repeats. It then marks down the positions of the endpoints and puts them in a set called $S_{repeat}$. To improve sensitivity, Signal Detector allows approximate matches when searching for repeats. Moreover, it only considers repeats that are maximal and are of significant length.

**Coverage:** Significant coverage variation along contigs can correspond to false joins of sequences from different genomes with different abundances. Coverage profile is the coverage depth along the contigs. For example, the coverage profile along a string $s = ACGT$ is $(1, 2, 2, 1)$ if the reads are $AC, CG, GT$. Signal detector aligns original reads to the contigs to find the coverage profile, which is called $S_{coverage}$.

## Signal Aggregator

After Signal Detector collects signals regarding palindromes $S_{palindrome}$, repeats $S_{repeat}$ and coverage profile $S_{coverage}$, Signal Aggregator uses them to determine the breakpoints on the input contigs $C$. The algorithm is summarized in Alg 17.

### ChimericContigFixing

The goal of ChimericContigFixing is to fix the contigs formed from chimeric reads. We simply break the palindromes in $S_{palindrome}$ at locations corresponding to their wrapping lengths. After removing redundant contigs, ChimericContigFixing returns the broken palindromes with the unchanged contigs.

**Algorithm 17** Signal Aggregator

**Input:** Input contigs $C$ and signals from Signal Detector $S_{palindrome}$, $S_{repeat}$ and $S_{coverage}$
**Output:** Contigs $C''$ with less mis-assemblies
**procedure** SIGNALAGGREGATION($S_{palindrome}, S_{repeat}, S_{coverage}, C$)
    $C'$ = ChimericContigFixing($S_{palindrome}$, $C$) ▷ Fix chimeric contigs
    $S_{filter}$ = LocatePotentialMisassemblies($S_{repeat}$, $C'$) ▷ Locate mis-assemblies caused by repeats
    $C''$ = ConfirmBreakPoints($S_{filter}$, $S_{coverage}$, $C'$) ▷ Confirm mis-assemblies using coverage
    **return** $C''$

**LocatePotentialMisassemblies**

The goal of the subroutine LocatePotentialMisassemblies is to locate potential mis-assemblies caused by repeats. We study the design of this subroutine in this section.

**Motivating question and example:** We can declare all the endpoints of approximate repeats in $S_{repeat}$ to be potential mis-assemblies. While this is a sensible baseline algorithm, it is not immediately clear whether it is sufficient or necessary. It is thus natural to consider the following question.

Given a set of contigs, how can we find the smallest set of locations on contigs to break so that the broken contigs are consistent with any reasonable ground truth? To illustrate our ideas, we consider an example with contigs $x_1 = abcde, x_2 = fbcg, x_3 = hcdi$ with $\{a, b, c, d, e, f, g, h, i\}$ being strings of equal length $L$.

The baseline algorithm of breaking contigs at the starting points of all the long($\geq 2L$) repeats breaks the contigs 4 times(i.e. $a|b|cde, f|bcg, h|cdi$). However, interestingly, we will show that one only need to break the contigs 3 times to preserve consistency (i.e. $x_1 = ab|cde, x_2 = fb|cg, x_3 = h|cdi$) and that is optimal.

**Modelling and problem formulation:** We will formalize the notions of feasible break points, feasible ground truth, consistency between sets of contigs, sufficiency of break points to achieve consistency and the optimiality criterion.

We use a graph theoretic framework. Specifically, we study a directed graph $G = (V, E)$ with $m$ sources $S$ and $m$ sinks $T$ where $\forall v \notin S \cup T, indeg(v) = outdeg(v)$ and parallel edges between two vertices are allowed. This is used to model a fully contracted De Bruijn graph formed by successive K-mers of the contigs. Vertices $V$ are substrings of the contigs and edges $E$ correspond to potentially mis-assembled locations on contigs. In our example, the set of vertices is $V = \{a, b, c, d, e, f, g, h, i\}$ and the set of edges is $E = \{ab, fb, bc_1, bc_2, hc, cd_1, cd_2, cg, de, di\}$. We use subscripts to differentiate parallel edges joining the same vertices. The graph corresponding to our running example is shown in Fig 5.3.

Given such a graph $G$, we note that $E$ is the set of all feasible break points because each edge in the graph corresponds to a potentially mis-assembled location on contigs. A

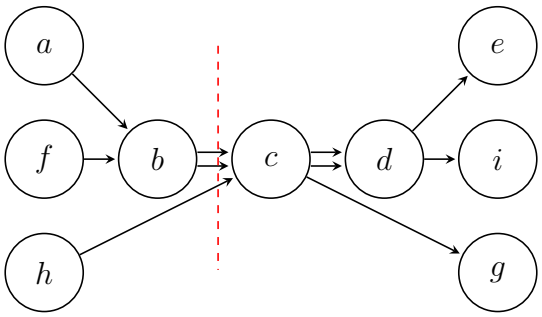



Figure 5.3: The graph corresponding to our example contig set $x_1 = abcde, x_2 = fbcg, x_3 = hcdi$ is shown. We note the optimal set of break points by the red dotted line.

feasible ground truth corresponds to a set of $m$ edge-disjoint source-to-sink trails that partitions the edge set $E$. For simplicity, we represent a trail as a sequence of the vertices in $G$, where the edges linking the vertices are ignored. For example, $\{abcde, fbcdi, hcg\}$ is a feasible ground truth while $\{abcg, fgde, hcdi\}$ is another feasible ground truth. The set of all feasible ground truths is $GT$.

We recall that our high level goal is to choose a set of feasible break points $R \subseteq E$ so that the broken contigs are consistent with any feasible ground truth. So, we need to define the notion of broken contigs and consistency between two sets of contigs under the current framework. Let $gt \in GT$, we denote $gt \backslash R$ be a set of trails after the removal of the edge set $R$. In particular, for the original contig set $C \in GT$, $C \backslash R$ is the set of broken contigs for the set of feasible break points $R$. For example, if $R = \{bc_1, bc_2, hc\}$ and $C = \{abcde, fbcdi, hcg\}$, $C \backslash R = \{ab, cde, fb, cdi, h, cg\}$. To capture consistency between two sets of contigs, we use the following definition. Given two sets of trails $T_1, T_2$, we say that $T_1$ is consistent with $T_2$ if $\forall x \in T_1, \exists y \in T_2 \ s.t. \ x \subseteq y$ and $\forall x' \in T_2, \exists y' \in T_1 \ s.t. \ x' \subseteq y'$. We call $R$ a sufficient breaking set with respect to $(C, GT)$ if $\forall gt \in GT, C \backslash R$ is consistent with $gt \backslash R$. In other words, $R$ is a set of feasible break points that allows the broken contigs to be consistent with any feasible ground truth. Although this notion of sufficient breaking set is a natural model of the problem, it depends on the original contig set $C$, which is indeed not necessary. Specifically, we show that we have an equivalent definition of sufficient breaking set without referring back to the original contig set. Let us call $R$ a sufficient breaking set with respect to $G$, or simply a sufficient breaking set, if $\forall gt_1, gt_2 \in GT, gt_1 \backslash R$ is consistent with $gt_2 \backslash R$.

**Proposition 5.3.1.** *$R$ is a sufficient breaking set with respect to $(C, GT)$ if and only if $R$ a sufficient breaking set with respect to $G$.*

*Proof.* The backward direction is immediate because $C \in GT$. We will show the forward direction as follows. Let $g_1, g_2 \in GT$ and we want to show that $g_1 \backslash R$ is consistent with $g_2 \backslash R$. Since $R$ is a sufficient breaking set with respect to $(C, GT)$, $g_1 \backslash R$ is consistent with $C \backslash R$. Therefore, $\forall x \in g_1 \backslash R \exists y \in C \backslash R \ s.t. \ x \subseteq y$. But since $g_2 \backslash R$ is consistent with $C \backslash R$ , we have

$\exists z \in g_2 \backslash R$ *s.t.* $y \subseteq z$. By transitivity, we have $x \subseteq y \subseteq z \in g_2 \backslash R$. By symmetry, we can also show that $\forall x' \in g_2 \backslash R \exists y' \in g_1 \backslash R$ *s.t.* $x' \subseteq y'$. Thus, $g_1 \backslash R$ is consistent with $g_2 \backslash R$. □

Now, we state our optimization criterion. The goal here is to minimize the cardinality of the sufficient breaking set, formally as Eq 5.1.

$$OPT = \min_{R \subseteq E} |R| \text{ } s.t. \text{ } R \text{ is a sufficient breaking set with respect to } G \tag{5.1}$$

We will show that if we solve Eq 5.1 for our running example, the answer is 3. This thus shows that the baseline algorithm of breaking contigs at all starting points (in our example, there are 4 of them) of all long repeats is not optimal.

**Proposition 5.3.2.** *For our running example, OPT = 3.*

*Proof.* First, by inspecting the 6 feasible ground truths in $GT$, we note $R = \{bc_1, bc_2, hc\}$ is a sufficient breaking set with respect to $G$. Second, we note that the edge set need to disconnect sources and sinks, otherwise, we can find $g_1, g_2 \in GT$ such that $g_1 \backslash R, g_2 \backslash R$ are inconsistent. This means $|R|$ need to be $\geq$ mincut of the graph, which is 3. □

**Algorithm development and performance guarantee:** Next we describe an algorithm that finds a sufficient breaking set with respect to $G$. Let us denote a boolean variable $b_e$ on each edge $e \in E$, with $\vec{b} = (b_e)_{e \in E}$. For $v \in V$, $InEdges(v), OutEdges(v)$ are the sets of incoming edges and outgoing edges at $v$ respectively. $Prev(v), Succ(v)$ are the sets of predecessor vertices and successor vertices of $v$ respectively. Our algorithm solves the following minimization problem (Eq 5.2) as a proxy to (Eq 5.1).

$$\min_{r \subseteq \vec{b}: r = True \Rightarrow \Phi(\vec{b}) = True} |r| \tag{5.2}$$

where,

$$\Phi(\vec{b}) = \wedge_{v:|Prev(v)| \geq 2} \text{ and } {}_{|Succ(v)| \geq 2}(\wedge_{e \in InEdges(v)} b_e \vee \wedge_{e \in OutEdges(v)} b_e) \tag{5.3}$$

In other words, it includes either all the incoming edges or all the outgoing edges for every vertices with at least 2 successors and at least 2 predecessors to $R$. We then seek $R$ with minimum cardinality among the choices available. If $G$ can be decomposed into connected components, we can optimize $\Phi(\vec{b})$ independently on each connected component. In our implementation, if the size of the connected component is not too large, we optimize the objective function by exhaustion. Beyond a certain threshold, we simply output a feasible solution without optimizing. Indeed, in our experiments on real data, most of the connected components are not that large and this step typically takes a few minutes. But we remark that for more complex applications, one can further optimize the algorithm. For example, one can first topologically sort the vertices and then use dynamic programming to solve Eq 5.2 along the sorted vertices.

We note that the algorithm described gives an optimal solution for our running example. Moreover, we show performance guarantee of the algorithm as follows.

**Proposition 5.3.3.** *Solving Eq 5.2 gives a sufficient breaking set $R$ if the graph $G$ is fully contracted.*

*Proof.* Let $R$ be the set of edges selected by the algorithm. For any two $gt_1, gt_2 \in GT$, we want to show that $gt_1 \backslash R$ and $gt_2 \backslash R$ are consistent. By symmetry, it suffices to prove that if $x \in gt_1 \backslash R$, then $\exists y \in gt_2 \backslash R$ *s.t.* $x \subseteq y$. If $|x| = 2$, it is immediate because every edge other than those in $R$ are covered. If $|x| \geq 3$, we will show that it is also true using proof by contradiction. If $\forall y \in gt_2 \backslash R, x \not\subseteq y$, we can find a sub-trail $x' = (a_1, a_2, ..., a_k, a_{k+1})$ of $x$ such that $\exists y' \in gt_2 \backslash R$ *s.t.* $x'' = (a_1, ..., a_k) \subseteq y'$ but $\forall y \in gt_2 \backslash R, x' \not\subseteq y$. This implies $\exists a^* \neq a_{k+1}$ *s.t.* $(x'', a^*) \subseteq z$ for some $z \in gt_2 \backslash R$. Since the edge $(a_k, a_{k+1}) \subseteq x \in gt_1 \backslash R$, we know that $(a_k, a_{k+1})$ is not in $R$. Similarly, $(a_k, a^*) \notin R$ because $(a_k, a^*) \subseteq y' \in gt_2 \backslash R$. But since $|Succ(a_k)| \geq 2$, the fact that our algorithm does not include $(a_k, a^*), (a_k, a_{k+1})$ in $R$ implies that $|Pred(a_k)| = 1$, namely $Pred(a_k) = \{a_{k-1}\}$. Note that we are considering a fully contracted graph. So, the fact that $a_{k-1}$ exists implies that $|Succ(a_{k-1})| \geq 2$. But our algorithm has not removed edge $(a_{k-1}, a_k)$. This gives $|Pred(a_{k-1})| = 1$. Inductively, we get $|Pred(a_i)| = 1 \forall 2 \leq i \leq k$. But we know that $(a_k, a_{k+1}) \subseteq w$ for some $w \in gt_2 \backslash R$. Since the edges along $(a_1, ..., a_{k+1})$ are not in $R$, this gives, $x' = (a_1, ..., a_{k+1}) \subseteq w \in gt_2 \backslash R$. This contradicts the assumption about $x'$. □

**Proposition 5.3.4.** *If the graph $G$ is fully contracted DAG without parallel edges, then solving Eq 5.2 returns a sufficient breaking set that is of optimal cardinality, $OPT$.*

*Proof.* It suffices to show that for any sufficient breaking set $R$, $\forall v \in V$ where $|Succ(v)| \geq 2, |Pred(v)| \geq 2$, we have either $InEdges(v) \subseteq R$ or $OutEdges(v) \subseteq R$. We prove it by contradiction. If not, $\exists v \in V$ where $|Succ(v)| \geq 2, |Pred(v)| \geq 2$ but $InEdges(v) \not\subseteq R$ and $OutEdges(v) \not\subseteq R$. Because it is a DAG, it means we can find $gt_1 \in GT$ such that $\exists x, y, x', y'$ such that $(x, v, y) \in gt_1$ and $(x', v, y') \in gt_1$. Now consider $gt_2$ to be a clone of $gt_1$ except that it has $(x, v, y'), (x', v, y)$ instead of $(x, v, y'), (x', v, y)$. We note that $gt_2 \in GT$. And because there are no parallel edges, $(x, v, y)$ is not a subset of any of the trails in $gt_2$. So, we find two distinct $gt_1, gt_2 \in GT$ such that $gt_1, gt_2$ are not consistent. This contradicts the fact that $R$ is a sufficient breaking set. □

It is noteworthy that if we are given any set of contigs from any $gt \in GT$, we still obtain the same set of broken contigs after the operation of removal of redundant trails, RemoveRedundant (i.e. we eliminate the contigs in a set $A$ to form a minimal subset $B \subseteq A$ in which $\forall x \neq y \in B, x \not\subseteq y$). This can be formalized as follows.

**Proposition 5.3.5.** *If $R$ is a sufficient breaking set, then for any $gt_1, gt_2 \in GT$,*

$$RemoveRedundant(gt_1 \backslash R) = RemoveRedundant(gt_2 \backslash R)$$

*Proof.* Let $s_i = RemoveRedundant(gt_i \backslash R)$ for $i \in \{1, 2\}$. By symmetry, it suffices to prove that $s_1 \subseteq s_2$ $\forall x \in s_1 \subseteq gt_1 \backslash R, \exists y \in gt_2 \backslash R$, such that $x \subseteq y$. Note that we can also find some $x^* \in s_2$ such that $y \subseteq x^*$. This gives $x \subseteq y \subseteq x^*$. However, since we have no redundant trails in $s_1$, we get $x = x^*$. Thus $x = x^* \in s_2$. □

To apply BIGMAC to real data, we have to implement the described algorithm with some further engineering. This includes methods to tolerate noise, to handle double stranded nature of DNA, and to construct De Bruijn graph from the repeats. Interestd readers can refer to the Appendix for these implementation details.

**ConfirmBreakPoints**

The goal of ConfirmBreakPoints is to utilize the coverage profile $S_{coverage}$ to confirm breaking decisions at potentially mis-assembled locations specified in $S_{filter}$. For the sake of simplicity, we now consider a string $s$ of length $L$, and a set of positions $Y = \{y_i\}_{1 \leq i \leq k}$ of $s$ which are potential mis-assemblies. Furthermore, let us assume that all mis-assemblies are caused by mixing genomes of different abundances. Using $Y$, we can partition $s$ into segments $\{s_i\}_{0 \leq i \leq k}$ of lengths respectively as $\{\ell_i\}_{0 \leq i \leq k}$. We let $x_i$ be the number of reads that start in segment $s_i$, which can be estimated from $S_{coverage}$. The question is how to classify the points in $Y$ as true mis-assemblies or as fake mis-assemblies.

One can use heuristics, like comparing coverage depth difference before and after each $y_i$. However, this is not ideal. For example, if we have coverage depth before and after $y_1$ differing by 1X, we would expect it to be a much stronger signal for true mis-assembly if the lengths $\ell_0, \ell_1$ are of order of 100K rather than of 100 and this cannot be seen by considering coverage depth difference alone. For the case of just two segments of equal length and if we assume the $x_i$'s are independent Poisson random variables, there is a popular statistical test, C-Test[52], that can make the decision. Formally, if $x_1 \sim Poisson(m_1)$ and $x_2 \sim Poisson(m_2)$, then C-Test is a test to decide between the hypothesis of $H_0 : m_1 = m_2$ against $H_1 : m_1 \neq m_2$. C-Test first considers $x_1 + x_2$ to compute the decision boundary and it then decides whether to reject $H_0$ based on $x_1/x_2$ and the previously derived decision boundary. The intuition is that $x_1 + x_2$ corresponds to the amount of data, which determines the confidence of a statistical test. Thus, if $x_1 + x_2$ is large, a small perturbation of $x_1/x_2$ from 1 can still be very significant and can correspond to a confident rejection of $H_0$.

However, we still need to think carefully about how to apply C-Test to our problem. One method is to directly apply $k$ independent C-Test on each of the junctions $y_i$. However, that method cannot take advantage of the information from neighboring segments to boost the statistical power at a junction. Therefore, we develop the algorithm IterativeCTest. IterativeCTest performs traditional C-Test but in multiple iterations. Initially, it directly applies $k$ independent C-Test on each of the junctions $y_i$. Some of the segments are merged and we aggregate the counts from the merged segments to continue to the next C-Test at the remaining unmerged junctions in $Y$. This process is repeated until the algorithm converges. Finally, we use the breaking decisions from IterativeCTest to break the incoming contigs and return the fixed contigs.

## 5.4 Merger: Merging Assembled Contigs

After fixing mis-assemblies, we are ready to study another pillar of BIGMAC: Merger. Merger establishes connectivity among contigs and subsequently makes decisions to extend contigs.

### Graph Operator

The goal of Graph Operator is to identify candidates for contig extension. It forms and transforms a string graph using the overlap information among original reads and contigs. It subsequently analyzes the graph to find candidates for contig extension. The overall algorithm is summarized in Alg 18.

**Algorithm 18** Graph Operator

1: **Input:** Contigs $C$ and original reads $R$
2: **Output:** String graph $G$ with information about candidates for contig extension
3: **procedure** GRAPHOPERATOR($R, C$)
4: M = Mapping(R, S) ▷ Obtain mapping among contigs and reads
5: G = FormGraph(M) ▷ Form string graph to represent connectivity
6: G.GraphSurgery(M) ▷ Simplify graph
7: G.FindExtensionCandidates() ▷ Identify candidates for contig extension
8: **return** G

**Mapping:** We obtain mapping among contigs and reads. This provides the fundamental building block to construct the connectivity relationship among contigs and reads.

**FormGraph:** The goal of FormGraph is to establish connectivity among contigs. We use contig-read string graph as our primary data structure. Contig-read string graph is a string graph[41] with both the contigs and the reads associated with their endpoints as nodes. The size of the graph is thus manageable because most reads are not included in the graph. Then, we construct an overlay graph (which we call the contig graph) such that the nodes are the contigs with weights on nodes being the coverage depth of contigs. In the contig graph, there is an edge between two nodes if there is a sequence of reads between the associated contigs. With these data structures, we can switch between macroscopic and microscopic representation of the contig connectivity easily.

**GraphSurgery:** GraphSurgery simplifies the contig graph. This includes removal of transitive edge and edge contraction.

For nodes $u, v, w$, if we have edges $u \to v, u \to w$ and $w \to v$, then we call $u \to v$ a transitive edge. We perform transitive reduction[41] on the contig graph to remove transitive edges. Removing these transitive edges prevents us from finding false repeats in the next stage. To improve robustness, there is a pre-processing step before transitive reduction. If the contig $w$ is too short and there are no reads that form head-to-tail overlap between $w, u$ or $w, v$, then we can have a missing edge for transitive reduction to operate properly. Thus,

we add the missing edge (either from $u$ to $w$ or $w$ to $v$) back when we find contigs and related reads that suggest the missing edge might be there.

An edge $u \to v$ is contractable when the outgoing degree of $u$ and the incoming degree of $v$ are both 1. We contract edges to fill gaps. Our experience with FinisherSC is that data are dropped in the assembler and so reconsidering them as a post-processing step can potentially fill some gaps. However, there is a caveat. In establishing connectivity in contig-read string graph, we only use reads close to each contig's endpoints (as a way to save computation resources), we may miss some legitimate connections. Therefore, very long repeats prevent the detection of linkage of contigs, thereby allow contigs to be erroneously merged. To address that, if there exists contig $w$ that is connected (by some reads) to a region close to the right end of $u$/left end of $v$, then we avoid contraction of $u \to v$.

**FindExtensionCandidates:** FindExtensionCandidates identifies candidates for contig extension by identifying local patterns in the contig graph. One form of extension candidates is a pair of contigs that are connected without competing partners. This corresponds to the contigs joined by a contractable edge. Another form of extension is a set of contigs that are connected with competing partners. This corresponds to the contigs linked together in the presence of repeats. In the contig graph, the repeat interior can either be represented as a separate node or not. If the repeat interior is represented as a separate node, the local subgraph is a star graph with the repeat node at the center. Otherwise, the local subgraph is a bipartite graph consisting of competing contigs. After identifying the contigs associated with a specific repeat, we can then merge contigs in the next stage.

## Contig Extender

After the operations by Graph Operator, we have extracted the potential contig extension candidates from the contig graph. It remains to decide whether and how to merge the corresponding contigs. In a high level, Contig Extender aims at solving the Contig Merging Decision Problem.

**Contig Merging Decision Problem.** *Given a set of incoming contigs $I$ and a set of outgoing contigs $O$ whose coverage depth and read connectivity information is known. Decide how to form an appropriate bipartite matching between $I$ and $O$.*

While we do not intend to rigorously define the statement of Contig Merging Decision Problem now, it is clear that appropriate matching corresponds to one that achieves high accuracy. Contig Extender carefully analyzes the read connectivity and contig coverage to make decisions on merging contigs. In the coming discussion, we first study an effective heuristic that captures the essence of the problem. After that, we will study how to rigorously define the Contig Merging Decision Problem in a mathematical form and suggest an EM-algorithm in solving that.

**Heuristic in solving the Contig Merging Decision Problem**

When the cardinality of the set of incoming contigs $I$ and the set of outgoing contigs $O$ are both 1, a natural decision is to merge them. Thus, the focus here is to study the case when $|I| > 1$ or $|O| > 1$. We select the reads that uniquely span one contig $a$ in the incoming set and one contig $b$ in the outgoing set. If there are multiple such reads, then we decide that $a, b$ should be joined together provided that there does not exist any paths of reads that lead $a$ to other contigs in the outgoing set and similarly for $b$. Moreover, we perform similar tests using contig coverage. If the coverage depth between two contigs is very close, they will be declared to be a potential merge pair. Then, we test whether there are any close runner-ups. If not, they should be merged. To combine the decisions made using spanning reads and coverage depth, we have a subroutine that discards all conflicting merges. We find that this heuristic for decision making works quite well in our datasets. However, in the coming discussion, we will study how to make the extension decisions in a more principled and unified manner.

**General framework in solving the Contig Merging Decision Problem**

The challenge for the Contig Merging Decision Problem is the tradeoff for many physical quantities (e.g. abundance, edit distance of reads, noise level, number of spanning reads, etc). We address this by defining a confidence score based on parameter estimation. For simplicity of discussion, we assume that $k$ is the cardinality of both the set of incoming contigs and the set of outgoing contigs. The goal is to find the best perfect matching with respect to a confidence score.

Let $M$ be a perfect matching of contigs among incoming and outgoing groups $I$ and $O$. Under $M$, there are $k$ merged contigs. Let the observation be the set of related reads $X = \{R_i \mid 1 \leq i \leq n\}$. We define the parameters $\theta = \{(\lambda_j, I_j)_{1 \leq j \leq k}\}$, where $\lambda_j$ is the normalized abundance of contig $j$ and $I_j$ is genomic content of the contig $j$. Note that $\sum_{1 \leq j \leq k} \lambda_j = 1$. So, the parameter estimation problem can be cast as $s_M = \max_\theta P_\theta(X \mid M)$, where $s_M$ is the confidence score of the matching $M$. Finally, the best perfect matching can be found by comparing the corresponding confidence score.

Note that we have hidden variables $Z = (Z_i)_{1 \leq i \leq n}$ which indicate the contigs that reads $X$ are extracted from (i.e. $Z_i \in \{1, 2, ..., k\}$). If we assume the length of the contig $j$ to be $\ell_j$ and $q$ to be the indel noise level (i.e. probability of $1 - 2q$ to be remained unaltered at each location), then we can use an EM-algorithm to obtain an estimate of $\theta$. Specifically, the expected value of the log likelihood function $E_{q(Z|X,\theta^{(t)})}[\log P_{\theta^{(t)}}(X, Z, \theta^{(t+1)}]$ is

$$\sum_{1 \leq i \leq n} \sum_{1 \leq j \leq k} M^j(R_i, I^{(t)})[\log \frac{\lambda_j^{(t+1)}}{\ell_j} + |R_i| \log(1 - 2q) + d(R_i, I^{(t+1)}) \log \frac{q}{1 - 2q}] \tag{5.4}$$

where $M^j(R, I^{(t)}) = \delta_{j = argmin_j d(R, I_j^{(t)})}$ is the assignment of reads to the most similar contig (with tie breaking using $\lambda^{(t)}$), $d(A, B)$ is the best local alignment score, $I^{(t)} = (I_j^{(t)})_{1 \leq j \leq k}$

are the genomic contents of the contigs at iteration $t$ and $\lambda^{(t)} = (\lambda_j)_{1 \leq j \leq k}$ are the estimated abundances at iteration $t$. By maximizing the log likelihood function with respect to $\theta^{(t+1)}$, we have the update formulas as

$$\lambda_{j^*}^{(t+1)} = \frac{\sum_{1 \leq i \leq n} M^{j^*}(R_i, I^{(t)})}{\sum_{1 \leq j \leq k} \sum_{1 \leq i \leq n} M^{j}(R_i, I^{(t)})} \tag{5.5}$$

$$I_{j^*}^{(t+1)} = \underset{I}{\operatorname{argmin}} \sum_{1 \leq i \leq n} M^{j^*}(R_i, I^{(t)}) d(R_i, I) \tag{5.6}$$

Note that the update of $I_{j^*}$ requires multiple sequence alignment. In general, the problem is NP-hard[15]. However, for noisy reads extracted from several contigs, the problem is not as difficult. For example, in the case of pure substitution noise, an efficient optimal solution is a column-wise majority vote. Despite the elegance and feasibility of this approach, it is still computationally more intense than the heuristic. Therefore, in our implementation of BIGMAC, the heuristic is the default method used in Contig Extender. But we also provide an experimental version of the EM-algorithm which can be used when users specify --option emalgo=True in their commands.

## 5.5 Experiments

### End-to-end experiments

#### Synthetic data

We verify that BIGMAC correctly improves the incoming contigs using the following synthetic data. We generate two synthetic species of different abundances $(\frac{2}{7}, \frac{5}{7})$. They are composed of random nucleotide sequences of length 5M each and share a common segment of length 12K. We uniformly sample indel noise corrupted reads of length 6K from the genomes, with coverage depth of 20X and 50X respectively. We also artificially construct mis-assembled contigs by switching the genome segments at the 12K repeat.

The contigs and the reads are passed through BIGMAC. BIGMAC breaks the contigs at the mis-assembled point and joins them back correctly. This is also the example that we discuss in the top-down design of BIGMAC.

#### Real data

We test the performance of BIGMAC in improving metagenomic assembly on PacBio only data. We use different datasets of different characteristics. Dataset 1 consists of a mock community of 5 species [20], with genomes of high similarity. Dataset 2 consists of a mock community of 23 species [46], with genomes of diverse abundances. We also remark that we have tested BIGMAC on a third PacBio only dataset (Dataset 3): a real experiment involving Desulfuromonas biwabikus, D. soudanensis and some other unknown genomes. We

note that we know the complete ground truth for the metagenomes in both Dataset 1 and 2 but only know part of the ground truth for Dataset 3. We take the output of HGAP and use the raw reads to improve them using BIGMAC. We observe that in these datasets, BIGMAC reduces the number of mis-assemblies while maintaining/increasing N50 and N75. The results of BIGMAC is summarized in Table 5.1, where the quantity of mis-assemblies is obtained from the QUAST reports. By default, QUAST's statistics are based on contigs of size >= 500 bp. Interested readers can refer to the Appendix for more details of the assessment. The data is provided in our online distribution and users can reproduce the results by issuing a single command `python reproduce.py`

Table 5.1: Performance evaluation of BIGMAC on several mock communities is shown. BIGMAC consistently improves assembly quality by simultaneously increasing contig contiguity and decreasing the number of mis-assemblies.

| Dataset | Method | NContig | # Mis-assembly | N50 | N75 |
|---|---|---|---|---|---|
| 1 | HGAP | 130 | 18 | 818655 | 274801 |
| 1 | BIGMAC | 129 | 7 | 4352719 | 274801 |
| 2 | HGAP | 477 | 187 | 397611 | 38471 |
| 2 | BIGMAC | 344 | 28 | 397611 | 75666 |
| 3 | HGAP | 185 | 26 | 257044 | 82370 |
| 3 | BIGMAC | 140 | 14 | 359704 | 99878 |

## Comparison with other post-processing tools

### Synthetic data

We use the synthetic data in Section 5.5 to evaluate and benchmark BIGMAC, FinishERSC[30], SSPACE_LongRead[5] and PBJelly[16]. BIGMAC is the only tool among the tested tools that fix the mis-assembled contigs and merge them back correctly. Other tested tools output the same mis-assembled contigs.

### Real data

We perform end-to-end testing to compare performance of different tools. The comparison is shown in Table 5.2. BIGMAC shows the largest N75/# Mis-assemblies on all three datasets and largest N50/# Mis-assemblies on two out of three datasets. Indeed, in the only dataset that BIGMAC does not have the largest N50/#Mis-assemblies, the number of contigs(i.e. L50) beyond N50 is 7. Due to the small number of contigs, this particular measurement on that dataset may not be significant. We also remark that BIGMAC is the only tool that improves contiguity (N50 and N75) and the number of mis-assemblies.

Table 5.2: Comparison of performance of BIGMAC with other post-processing tools for long read assemblies is shown. BIGMAC shows the largest N75/# Mis on all three datasets and largest N50/# Mis on two out of three datasets. We also remark that BIGMAC is the only tool that can improve N50 and N75 while reducing the number of mis-assemblies. Note that # Mis is an abbreviation for the number of mis-assemblies.

| Data | Method | # Mis | N50 | N75 | N50/# Mis | N75/# Mis |
|---|---|---|---|---|---|---|
| 1 | HGAP | 18 | 818655 | 274801 | 45481 | 15267 |
| | BIGMAC | 7 | 4352719 | 274801 | **621817** | **39257** |
| | FinisherSC | 32 | 2531294 | 415024 | 79103 | 12970 |
| | PBJelly | 19 | 4642330 | 418480 | 244333 | 22025 |
| | SSPACE_LR | 32 | 4657611 | 493683 | 145550 | 15428 |
| 2 | HGAP | 187 | 397611 | 38471 | 2126 | 206 |
| | BIGMAC | 28 | 397611 | 75666 | **14200** | **2702** |
| | FinisherSC | 192 | 654163 | 43018 | 3407 | 224 |
| | PBJelly | 271 | 1585584 | 61775 | 5851 | 228 |
| | SSPACE_LR | 255 | 1568442 | 95133 | 6151 | 373 |
| 3 | HGAP | 26 | 257044 | 82370 | 9886 | 3168 |
| | BIGMAC | 14 | 359704 | 99878 | 25693 | **7134** |
| | FinisherSC | 25 | 996532 | 97964 | 39861 | 3919 |
| | PBJelly | 27 | 1103847 | 128718 | **40883** | 4767 |
| | SSPACE_LR | 43 | 1266912 | 290104 | 29463 | 6747 |

## Simulation and testing on independent components

We perform micro-benchmarking on individual components of BIGMAC. The settings and results are summarized as follows.

**Analysis of LocatePotentialMisassemblies** : We test our break point finding algorithm used in LocatePotentialMisassemblies of Breaker on the synthetic data of $x_1 = abcde, x_2 = fbcg, x_3 = hcdi$ discussed in the previous section. The algorithm succeeds in identifying the minimum number of break points as 3. Also, it succeeds in identifying the minimum number of break points as 2 in the presence of double stranded DNA, in the test case of $x_1 = abcd, x_2 = ec'b'f$, where $b', c'$ are the reverse complement of $b, c$ respectively.

**Analysis of ConfirmBreakPoints**: We test IterativeCTest used in ConfirmBreakPoints of Breaker on synthetic data. We simulate mis-assemblies and variation on abundances by generating a sequence of Poisson random variables and compare the performance of the algorithms on the simulated data as follows. We generate a sequence of 100 numbers by 100 independent Poisson random variables. The Poisson random variables have parameters of either 20 or 50. To select the parameters, we simulate 100 steps of a two-states Markov chain with transition probability of 0.1. We then evaluate the performance of C-Test and IterativeCTest on finding the true transition points, which correspond to the junctions of mis-assemblies. Taking average from 100 rounds of simulation, the recall of both C-Test and

IterativeCTest are 0.93, meaning that they both can identify almost all transition points. C-Test has precision of 0.75 while the precision of IterativeCTest is of 0.87, meaning that IterativeCTest can avoid more fake mis-assemblies.

**Analysis of Merger** : We compare Merger with other tools on synthetic data as follows. We use a synthetic contig set $\{x_1, x_2, r, y_1, y_2\}$ where the ground truth genomes are $(x_1, r, y_1), (x_2, r, y_2)$. The coverage depth of $(x_1, y_1)$ and $(x_2, y_2)$ are 20X and 50X respectively. We pass the reads together with the contig set to FinisherSC, PBJelly, SSPACE_LongRead to perform scaffolding. We note that BIGMAC is the only tool the can scaffold the contigs correctly into 2 contigs by using the abundance information among the tested tools. Other tools either do not change the input or simply merge $r$ with some of $\{x_1, x_2, y_1, y_2\}$, resulting in 4 contigs.

### Runtime of BIGMAC

The runtime of BIGMAC is summarized in Table 5.3. We use 20 threads to run the tool on a server computer. The server computer is equipped with 64 AMD Opteron(tm) Processor 6380(8 cores) with frequency 2500 MHz and 362GB RAM. We note that the majority of the time is spent on alignment of contigs and reads by MUMmer.

Table 5.3: Runtime of BIGMAC and the corresponding file size

| Dataset | Component | Contig file size (MB) | Read file size (GB) | Running time (sec) |
|---|---|---|---|---|
| Synthetic | Breaker | 9.6 | 0.335 | 164 |
| Synthetic | Merger | 9.6 | 0.335 | 123 |
| 1 | Breaker | 30 | 5.7 | 6646 |
| 1 | Merger | 29 | 5.7 | 6998 |
| 2 | Breaker | 32 | 5.8 | 4865 |
| 2 | Merger | 29 | 5.8 | 5087 |
| 3 | Breaker | 17 | 7.6 | 7099 |
| 3 | Merger | 14 | 7.6 | 6887 |

## Discussion

There are two main implications from the experiments performed. First, we show that post-processing assemblies is a feasible alternative in improving assembly quality to building another assembler from scratch. This is demonstrated by BIGMAC showing simultanous improvement in terms of number of mis-assembly and contiguity. We note that this is a non-trivial feature because all other tested tools fail to achieve it. Second, BIGMAC is competitive when compared to the existing post-processing tools. In particular, it shows better N75/# Mis-assemblies than all other tested tools in all tested datasets. Moreover,

BIGMAC is also the only tool that can handle abundance information, which makes it an attractive candidate for improving metagenomic assembly.

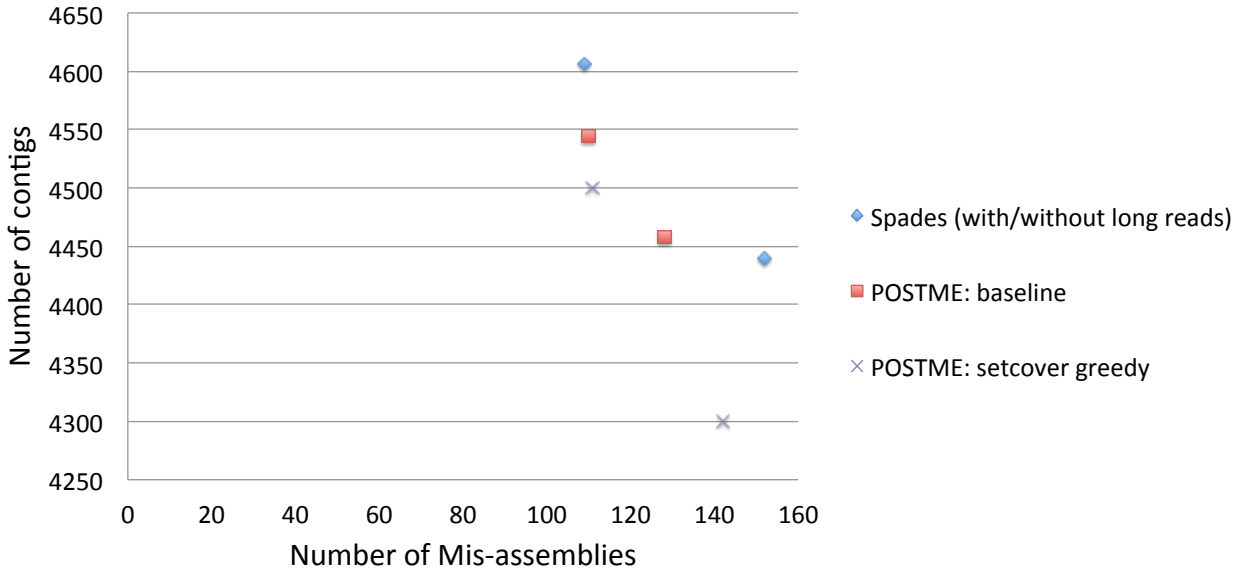
# Chapter 6

# POSTME : POSTprocessing MEtagenomics assembly with hybrid data by highly precise scaffolding

## 6.1 Problem statement

We study data efficiency of de novo metagenomic assembly using hybrid data. Initial inputs are short reads (e.g. Illumina reads) and long reads (e.g. PacBio reads). We investigate whether state-of-the-art assemblers are data-efficient for these data. We use our previously established post-processing paradigm to guide this study. To quantify the assembly quality, we focus on two dimensions, which are the number of contigs and the number of mis-assemblies. As a preliminary evaluation, we use Spades to assemble the short reads. Let the resultant contigs be $C_1$. We also use Spades-Hybrid to assemble both the short reads and the long reads. Let the resultant contigs be $C_2$. On a mock community of 26 genomes, it is observed that there are less contigs but more mis-assemblies in $C_2$ than $C_1$. To study data efficiency, we build a post-processor that takes in $C_1$, short reads and long reads to produce improved contigs $C_3$. We note that our post-processor only merges contigs, so the number of contigs is a good proxy for contiguity. It turns out that $C_3$ has higher quality than $C_2$ both in terms of number of contigs and number of mis-assemblies. On the other hand, we can also tune parameters to produce more conservative merges. In that case, when compared with $C_1$, we can significantly reduce the number of contigs while keeping the number of mis-assemblies almost unchanged. Details can be seen in Fig 6.1. This shows that improvement on the state-of-the-art hybrid assembler Spades-Hybrid can be made.

Figure 6.1: Performance of POSTME




## 6.2 Baseline algorithm

Let us study a baseline post-processing algorithm that captures the essential features of the hybrid data. Hybrid data involves long reads with low coverage and short reads with high coverage. Our baseline algorithm uses long reads to establish connectivity among contigs and uses short reads to estimate the coverage depth of the contigs. The algorithm is as follows. We find contig pairs that are connected by $m$ uniquely spanning reads. A spanning read is connected to some contigs on either end. A uniquely spanning read is connected to exactly one contig on either end. We then use coverage depth information to confirm merges for the contig pair using C-Test at confidence level $q$. We implement this algorithm and test it on the previously described dataset. We have a few interesting observations. First, by requiring $m = 3$ uniquely spanning reads and a confidence level of $q = 0.95$ in C-Test, the baseline algorithm can generate contigs of higher quality than the input contigs $C_1$ (i.e. a significant decrease in the number of contigs with almost the same number of mis-assemblies). Second, if we further relax the parameters of $q$ and $m$, we have a tradeoff in quality improvement. One can see detail results in Fig 6.1. However, despite having fewer mis-assemblies, the baseline algorithm produces more fragmented contigs than Spades-Hybrid. So, whether Spades-Hybrid is already competitive enough is still an open question.

## 6.3 Data analysis

To figure out the bottleneck of the baseline algorithm, we perform explorative data analysis. We study the input contigs $C_1$ produced by Spades using a scatter plot in Fig 6.2. In the

figure, each contig corresponds to a point whose y-coordinate is the coverage depth from short reads and x-coordinate is the length of the contig. It is noteworthy that there is a lot of contigs of short length in $C_1$. Since they constitute a significant fraction of the total contig count, proper merging involving these contigs has a significant impact on the overall quality. We further note that these contigs have diverse coverage depth, which suggests that they can easily pass the C-Test. Since the long reads are uniformly sampled from the genomes, connectivity for these contigs should be comparable with those of the other contigs (if it is done properly). However, there is a caveat in establishing connectivity: a single long read can indeed span multiple contigs formed from short reads. We confirm this hypothesis by performing further data analysis, which is shown in Fig 6.3. In the figure, the x-axis is the number of distinct contigs embedded in the long read and the y-axis is the number of such long reads. According to the figure, there is a significant number of short contigs that are embedded in the long reads. The baseline algorithm overlooks this characteristic of the data because it assumes contigs to be long enough and thus uses long reads for scaffolding only. Thus, it is a potential bottleneck over which we can optimize.

Figure 6.2: Data analysis of the contigs. Y-axis is the coverage depth and X-axis is the contig length.

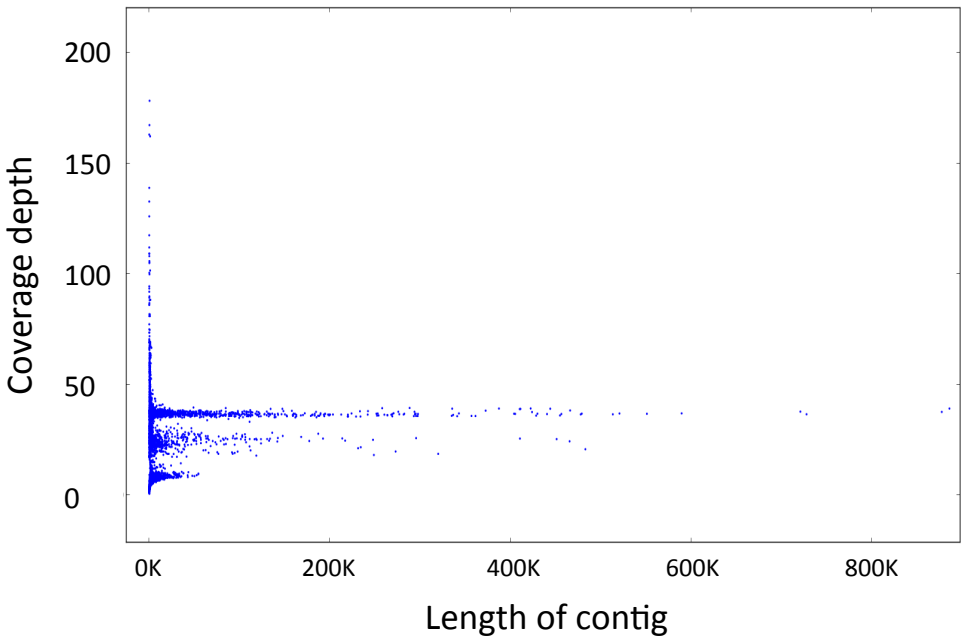
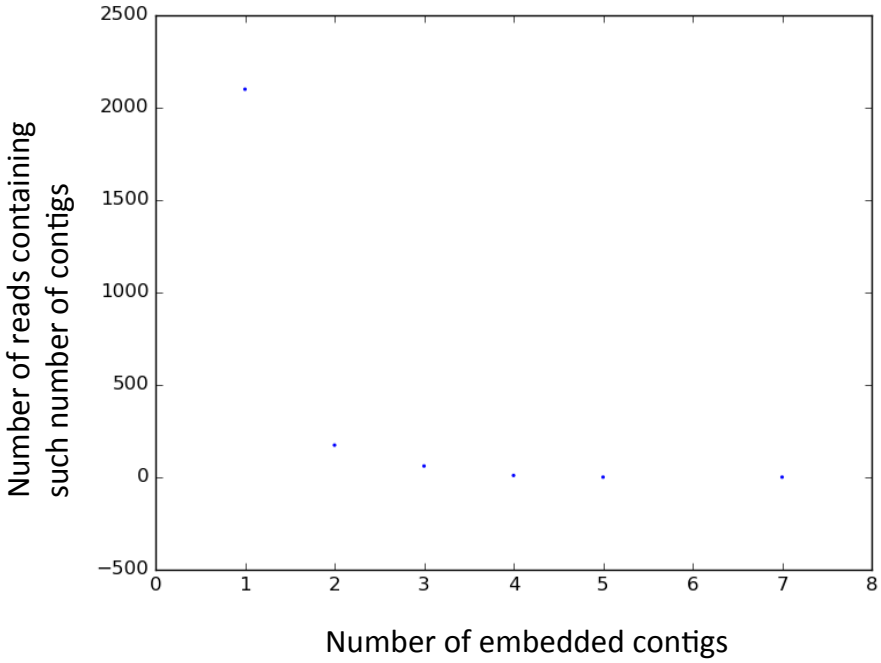


Figure 6.3: Data analysis on the number of short contigs embedded in long reads.




## 6.4 Set cover minimization

We use set cover minimization formulation. The short contigs are treated as elements to be covered. The set containers are the long reads, in which the short contigs are embedded. To include all the contigs, we append set containers containing only one contig. We note that some contigs (i.e. elements) can simultaneously be embedded in multiple reads (i.e. set containers). The goal is to find the minimum set cover, that is, to find the set containers that include all the short contigs and are of the smallest overall cardinality. We note that finding minimum set cover is NP-hard in general and it is known that greedy algorithm gives an optimal approximation ratio unless P = NP. For the contigs embedded in a long read, we consider the neighboring contig to be potential merging pairs. C-Test is then used to decide on the final merging. We note that POSTME is a modification of the baseline algorithm with this extra step on set cover minimization. We implement POSTME and test it on the example data. Interestingly, with this single optimization, we can produce contigs with higher quality than those produced by Spades-Hybrid, both in terms of the number of contigs and the number of mis-assemblies.

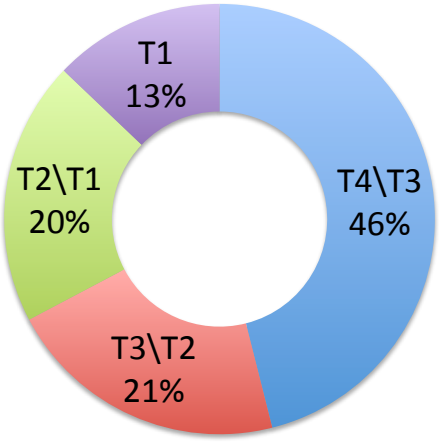
## 6.5 Benchmarking

Besides Spades-Hybrid, we also benchmark the quality improvement made by POSTME with other post-processing tools. The goal is to explore other dimensions for improvement or to verify that POSTME has already obtained good enough quality improvement. For example, we compare the performance of POSTME with Minimus2. Minimus2 is a tool for merging assemblies, which is also used to merge the error-corrected long reads with the contigs formed from the short reads. We note that despite a significant drop in contig count, there is also a significant drop in the captured genome fraction for Minimus2. This means that Minimus2 cannot provide conclusive answer on quality improvement in terms of data efficiency. Indeed, a trade-off is observed when we test POSTME with other tools, which cannot provide conclusive answer to whether further improvement is possible. Thus, in additional to benchmarking, we need another approach to check whether POSTME is good enough.

## 6.6 Further optimization feasibility test

To study the question on how good is good enough, we analyze the characteristics of all possible merging pairs to see if we are already close to optimal (i.e. having minimum contig count while not making further mis-assemblies). We define an ascending hierrachy ($T1 \subseteq T2 \subseteq T3 \subseteq T4$) of merging pairs. $T1$ includes the merging pairs that are connected by three or more uniquely spanning reads and do not have competing contigs. $T2$ includes the merging pairs that are connected by one or more uniquely spanning reads and do not have competing contigs. $T3$ includes the merging pairs that are connected by one or more uniquely spanning reads. $T4$ includes the merging pairs that are connected by one or more spanning reads. The proportion of merging pairs in the example data is shown in Fig 6.4. We also measure the success rate of merges in our example data. It turns out that all merges in $T1$ are correct whereas 90% of the merges in $T2$ are correct. However, $T2$ only corresponds to around one-third of all the merges and they are the only merges considered in POSTME. For the other two-thirds, overlap information is clearly not adequate because of competing contigs. However, it is still not clear whether the abundance information can come to the rescue. Thus, we rank the merges by the ratio of C-Test on the best match versus the second best match in each conflicting cluster. Surprisingly, in order to make no errors, the best threshold can only be made to include the top two merging pairs. This suggests that POSTME is performing reasonably well in the dataset.

Figure 6.4: Relative proportion of each class of merging pairs




## 6.7 Discussion

As an explorative research, POSTME effectively demonstrates the potential to improve the assembly quality on hybrid data. Moreover, we also remark that the design of POSTME demonstrates the agility of the post-processing approach in the study of data efficiency for emerging data. We hope that this case study can spark more progress in improving the quality of metagenomic assembly involving hybrid data.

# Appendix A

# Appendix of Near-optimal Assembly for Shotgun Sequencing with Noisy Reads

# A.1 Appendix: Proof on performance guarantee

Here we use the short hand of $l_{repeat}$, $l_{interleaved}$ and $l_{triple}$ to represent the corresponding approximate repeat length of longest simple, interleaved, triple repeat respectively.

## Greedy algorithm

Let us define a $\theta-$ neighborhood of the repeat specified by $x[a:b]$ and $x[c:d]$ to be the loci of repeat which are $x[a-\theta:b+\theta]$ and $x[c-\theta:d+\theta]$.

We say a repeat is $\theta-$bridged if there exists a read that cover the $\theta$-neighborhood of at least one copy of the repeat. For simplicity of arguments, we assume $l_{repeat} >> \max(l_{intereleave}, l_{triple})$.

**Lemma A.1.1.** *We first note the following sufficient conditions for Noisy Greedy to succeed.*

1. *Merging at stages from $L$ to $\ell_{\ddot{\imath}id}(p,\epsilon,G)$ are merging successive reads*
2. *Every successive reads have overlap with length at least $\ell_{\ddot{\imath}id}(p,\frac{\epsilon}{3},G)$*

**Theorem A.1.2.** *Under the generative model on genome, with $\ell_{\ddot{\imath}id} = \ell_{\ddot{\imath}id}(p,\frac{\epsilon}{3},G)$, $\alpha = \alpha(p,\frac{\epsilon}{3},G)$, if*

$$L > l_{repeat} + 2\cdot\ell_{\ddot{\imath}id}$$
$$G > N > \max(\frac{G\ln\frac{3}{\epsilon}}{L - l_{repeat} - 2\cdot\ell_{\ddot{\imath}id}}, \frac{G\cdot\ln\frac{N}{\epsilon/3}}{L-\ell_{\ddot{\imath}id}})$$

*, then $\mathcal{P}(\mathcal{S}^C) \leq \epsilon$.*

*Proof.* In order to prove that claim, let us break down into several subparts

Let $E_1$ be the event that condition 1 in Lemma (A.1.1) is not satisfied. $E_2$ be the event that condition 2 in Lemma (A.1.1) is not satisfied. $E_3$ be the event that the long/interleave/triple repeat is not $\ell_{\ddot{\imath}id}-$bridged.

Now we claim that with the chose $(N, L)$ in the range,

1. $\mathcal{P}(E_1) \leq \frac{\epsilon}{3}$
2. $\mathcal{P}(E_3) \leq \frac{\epsilon}{3}$
3. $\mathcal{P}(E_2 \mid E_1^C \cap E_3^C) \leq \frac{\epsilon}{3}$

We first see how we can use these to obtain the desired claim and proceed to prove each of the above sub-claims.

$$\begin{aligned}\mathcal{P}(S^C) =& \mathcal{P}(E_1) + \mathcal{P}(E_2 \cap E_1^C) \\ =& \mathcal{P}(E_1) + \mathcal{P}(E_2 \cap E_1^C \cap E_3^C) + \mathcal{P}(E_2 \cap E_1^C \cap E_3) \\ \leq& \mathcal{P}(E_1) + \mathcal{P}(E_2 \mid E_1^C \cap E_3^C) + \mathcal{P}(E_3) \\ \leq& \frac{\epsilon}{3} + \frac{\epsilon}{3} + \frac{\epsilon}{3} \\ =& \epsilon\end{aligned}$$

Now, we proceed to prove each of the sub-claims.

1. With $N > \frac{G \cdot \ln \frac{N}{\epsilon/3}}{L - \ell_{\text{iid}}}$, we have,

$$\begin{aligned}\mathcal{P}(E_1) \leq& N \exp(-\frac{N}{G}(L - \ell_{\text{iid}})) \\ \leq& \frac{\epsilon}{3}\end{aligned}$$

2. With $N > \frac{G \cdot \ln \frac{3}{\epsilon}}{L - l_{repeat} - 2 \cdot \ell_{\text{iid}}}$we have,

$$\begin{aligned}\mathcal{P}(E_3) \leq& \exp(-\frac{N}{G}(L - 2\ell_{\text{iid}} - l_{repeat})) \\ \leq& \frac{\epsilon}{3}\end{aligned}$$

3. With the choice of $\ell_{\text{iid}} = \ell_{\text{iid}}(p, \frac{\epsilon}{3}, G)$, we have,

$$\begin{aligned}\mathcal{P}(E_2 \mid E_1^C \cap E_3^C) \leq& N^2 \cdot \exp(-\ell_{\text{iid}} D(\alpha || \frac{3}{4})) \\ &+ 2N \exp(-\ell_{\text{iid}} D(\alpha || \eta)) \\ \leq& G^2 \cdot \exp(-\ell_{\text{iid}} D(\alpha || \frac{3}{4})) \\ &+ 2G \exp(-\ell_{\text{iid}} D(\alpha || \eta)) \\ \leq& \frac{\epsilon}{3}\end{aligned}$$

Here we use the fact that there are indeed 4 types of overlap as in Fig A.1. And given the bridging condition, we are only left with 2 types, namely, both ending segments outside/exactly one ending segment outside the longest repeat repeat region.

□

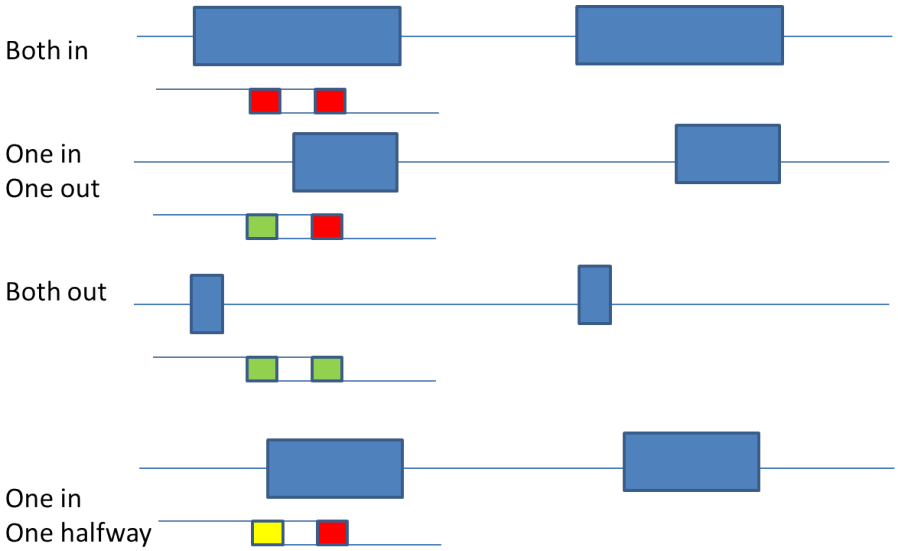



Figure A.1: Overlap Type

**Algorithm 19** Noisy Simple De Bruijn

0. Choose K to be $\max(l_{interleave}, l_{triple})$
1. Extract Kmers from reads
2. Clusters Kmers
3. Form Kmer Graphs
4. Condense the graph
5. Clear Branches
6. Condense graph
7. Find Euler Cycle

## Simple De Bruijn algorithm

Before continuing proving the performance of Multibridging Algorithm, it is instructive to analyze the following Simple De Bruijn Algorithm (Alg 19) because this is closely related to the Multibridging Algorithm.

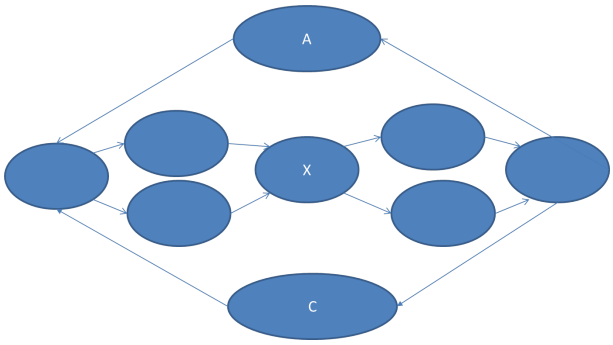
Here we first define several genomic region of interest which we will refer to in the proofs below.

a) $S_0$ = set of K-mers that are completely inside $\ell_{iid}-$ neighborhood of the longest repeat
b) $S_1$ = set of K-mers that are completely inside the longest repeat
c) $S_2 = S_0 \backslash S_1$

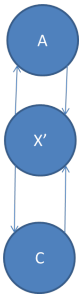
**Lemma A.1.3.** *Here we provide several deterministic conditions that guarantee the success of the algorithm.*

*1. Successive reads overlap with length at least K*

*2 K-mers are almost correctly clustered, that is,*




(a) Before branch clearing




(b) After branch clearing

Figure A.2: Branch clearing

a) K-mers from the same locus but not merged

*b)* $x$ not in $S_0$ s.t. $x$ get clustered with wrong K-mers

*c)* $x$ in $S_0$ s.t.$x$ get clustered with elements other than its own cluster/mirror cluster(mirror cluster is defined to be the cluster for the other copy of the repeat)

*3) Repeat at both circle are at least* $2 \cdot \ell_{\text{ıid}}$ *separated(the interleaving segments between the repeat differ with at least* $2\ell_{\text{ıid}}$ *in length)*

*Proof.* We note that every length K segments $x \notin S_0$, they are represented as a distinct node in the K-mer graph because of the length K that we pick and the condition that successive reads overlap at least K bases. Moreover, for K-mers $x \in S_1$, they are condensed into the repeat as 'X' in Fig A.2a. However, for the K-mers $x \in S_2$, they have chances not to merge properly, thus they form into the branches surrounding 'X' in Fig A.2a. Because of condition 3, branch clearing will not eliminate the 'A' or 'C' in Fig A.2a, further after condensing, we get the desired K-mer graph as in Fig A.2b and this can be successfully read by a Eulerian Walk. □

**Theorem A.1.4.** *If* $G > \frac{6}{\epsilon \ell_{\text{ıid}}}, G \geq N \geq \frac{G \cdot \ln \frac{3N}{\epsilon}}{L - \max(l_{int}, l_{triple}) - 2\ell_{\text{ıid}}}$ *, with* $\ell_{\text{ıid}} = \ell_{\text{ıid}}(p, \frac{\epsilon}{3}, G)$ $\alpha = \alpha(p, \frac{\epsilon}{3}, G)$*, then,* $\mathcal{P}(S^C) \leq \epsilon$

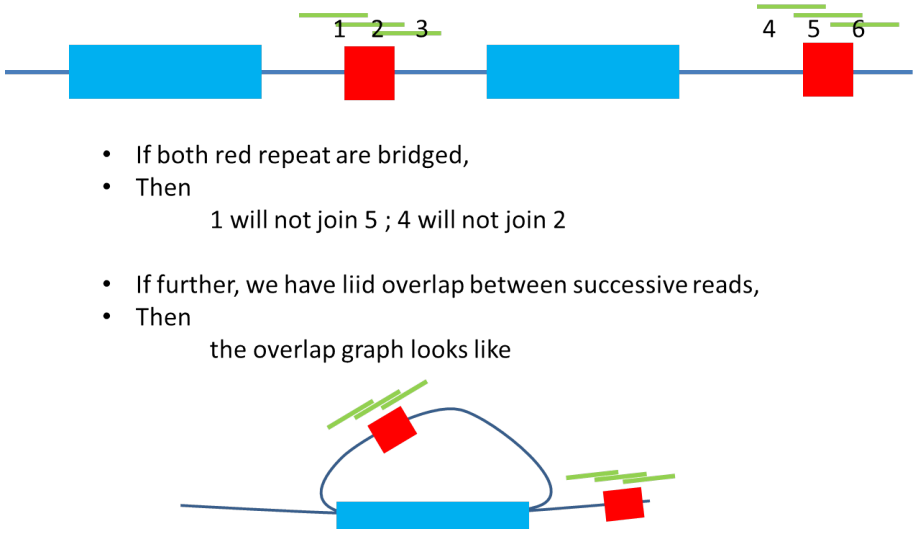



Figure A.3: An illustration of the Noisy Multi Bridging

*Proof.* We first note that in order to obtain a bound on the error probability, we only need to separately bound the probability that each of the conditions in Lemma A.1.3 fail,which are $\leq \frac{\epsilon}{3}$ each. Thus, combining, we get, $\mathcal{P}(S^C) \leq \epsilon$. □

## Multibridging algorithm

An illustration of noisy Multibridging Algorithm is shown in Fig (A.3).

**Lemma A.1.5.** *Here are the deterministic conditions for the algorithm to succeed.*
*1) Every successive reads overlap at least* $\ell_{iid}(p, \frac{\epsilon}{3}, G)$
*2) K-mers are almost correctly clustered, that is,*
a) K-mers from the same locus but not merged
*b)* $x$ not in $S_0$ s.t. $x$ get clustered with wrong K-mers
*c)* $x$ in $S_0$ s.t. $x$ get clustered with elements other than its own cluster/mirror cluster
*3) Repeat at both circle are at least* $2 \cdot \ell_{iid}$ *separated*
*4) When finding successors/predecessors, they are the real successors and predecessors*

*Proof.* Along the same lines as the proof in Lemma (A.1.3), we only note that in this algorithm, we have an extra step of finding predecessor/successors. Moreover, the overlap here is significantly reduced to only $\ell_{iid}$ instead of K in the Noisy Simple De Bruijn case. □

**Theorem A.1.6.** *With* $G > \frac{6}{\epsilon \ell_{iid}}$, $G \geq N \geq \max(\frac{G}{L-2\ell_{iid}} \ln \frac{N}{\epsilon/3}, \frac{G \ln \frac{3}{\epsilon}}{L-\max(l_{triple}, l_{interleave})-2\ell_{iid}})$ *,with* $\ell_{iid} = \ell_{iid}(p, \frac{\epsilon}{3}, G)$ $\alpha = \alpha(p, \frac{\epsilon}{3}, G)$ *, then* $\mathcal{P}(S^C) \leq \epsilon$.

*Proof.* Here we note that with the given coverage, bridging conditions of the interleave repeat and the triple repeat are satisfied. And when this is true, then Condition 4 in Lemma A.1.5 is true with high probability. Following the arguments in Theorem A.1.4, we get desired. □

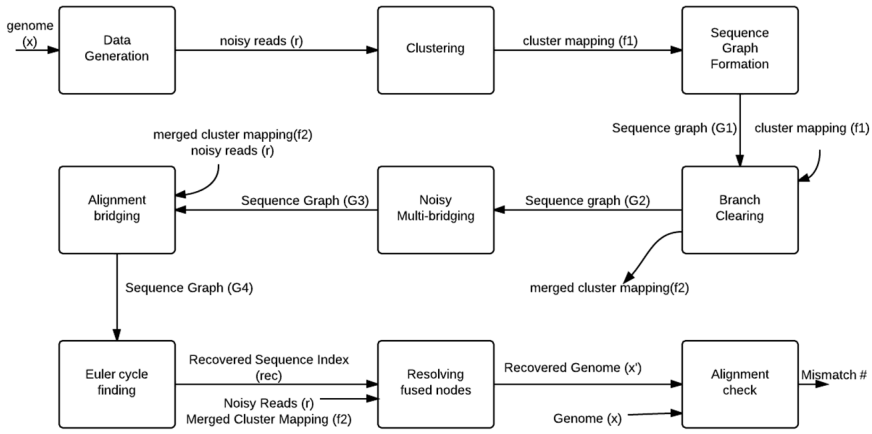



Figure A.4: Pipeline of the prototype assembler

## A.2 Appendix: Design and additional algorithmic components for the prototype assembler

### Pipeline of the prototype assembler

The pipeline of the prototype assembler is shown in Fig A.4. With a ground truth genome as input, the output is the performance of the whole pipeline by giving the mismatch rate.

### A more robust branch clearing step

Since we employ a speed up step in the clustering and there may be K-mers that are not completely clustered correctly in the clustering step of Multibridging Algorithm. Regarding that, we need to have a more robust branch clearing step. In particular, we first classfy nodes as “big” or “small” nodes based on the size of the nodes in the sequence graph. The key idea is to merge the small nodes together while keeping the big nodes unchanged. Starting from each big nodes, we tranverse the graph to detect all the small nodes that link the current big node to other big nodes. Then, we classify the small nodes into levels(depending on its distance from the current big node). After that, the small nodes in the same level are merged. Finally, we note that we keep the reachability among each big nodes.

### Enhanced Multibridging algorithm that can resolve middle range repeats

We note that the ideas presented here can also be found in the prior work on the treatment of noiseles sreads. It is stated here for completeness. In the noisy setting, instead of considering

the alphabet set to be $\Sigma = \{A, C, G, T\}$, one can consider the alphabet set as the cluster index of the K-mers.

**Algorithm 20** Enhanced Multibridging Algorithm

Resolution of repeats:
0. Intially the weight of the edge are set to be 1.
1. While there is a X-node $v$ :
a) For each edge ($p_i$,$v$) with weight $a_{p_i,v}$, create a new node $u_i = {}^{p_i\rightarrow} v$ and an edge $(p_i, u_i)$ with weight $1 + a_{p_iv}$. Similarly, for each edge ($v$,$q_j$), create a new node $w_j = v^{\rightarrow q_j}$ and an edge ($w_j$, $q_j$)
b) If $v$ has a self-loop ($v$, $v$) with weight $a_{v,v}$, add an edge ($v^{\rightarrow v}$,${}^{v\rightarrow} v$) with weight $a_{v,v} + 2$
c) Remove node $v$ and all incident edges
d) For each pair of $u_i$, $w_j$ adjacent in a read(extending to at least length of $\ell_{iid}$ on both sides of the X-node), add edge $(u_i, w_j)$. If exactly on each of the $u_i$and $w_j$nodes have no added edge, add the edge.
e) Condense the graph

## A.3 Appendix: Treatment of indel noise

### Formation of K-mer De Bruijn graph for indel corrupted reads

In order to form K-mer De Bruijn graph for indel corrupted reads, we first need to have a clear notion of K-mers. We define K-mers to be the length K segments in the genome ground truth (as opposed to the usual definition from the reads). Although we mostly work on the reads themselves, the definition of the Kmers are based on the ground truth. In order to successfully cluster K-mers, we need to do the following steps.

1. We first compute the pairwise alignement of the reads.

2. Based on the pairwise alignment, for each length K-segments, we know which should be aligned to which. We then group them together using the alignment result.

3. Finally, we end up with the length K segments from the reads clustering together, and now we use it as an operational way to identify the Kmers since each cluster will naturally correspond to a K-mers originated from the genome groundtruth(though there are a few discrepancy, mostly this is correct).

4. After we identify the K-mers clusters, we add an edge between them if there exists a read such that there are two consecutive Kmers originate from it.

#### Graph surgery to clear abnormality of the noisy De Bruijn graph

Due to indel noise and runs of the same alphabet, the way that we form K-mers graph may need to abnormality of the graph. We thus perform a graph tranversal and identify the abnormality that are of short length(i.e. resulted from noise but not the genome structure).

After that, we remove such abnormality. This step also involves transitive edge reduction and removal of small self loops.

## X-phased step tailored for indel noise type

### Generalization to handle indel error

When dealing with indel noise, the neighborhood of reads can also affect consensus of the base. There we have to do sequence alignment in order to find the appropriate posterior probability in order to do a maximum likelihood estimate of whether a particular given genomic location is a site of polymorphism or not. In order to do that, we formulate the problem as a ML problem as follows.

$$\max_{T\in\Omega} \quad \Pi_{i\in S} P(R_i \mid T), \qquad P_{err} = P_{opt} \tag{A.1}$$

$$\max_{T\in\Omega'} \quad \Pi_{i\in S} P(R_i \mid T), \qquad P_{err} = P_{opt} + \delta_1 \tag{A.2}$$

$$\max_{T\in\Omega'} \quad \Pi_{i\in S'} P(R_i \mid T), \qquad P_{err} = P_{opt} + \delta_1 \tag{A.3}$$

$$\max_{T\in\Omega'} \quad \Pi_{(j,k)}\Pi_{i\in S'_{jk}} P(R_i \mid T_j^{j+k}), \qquad P_{err} = P_{opt} + \delta_1 \tag{A.4}$$

$$\max_{T\in\Omega'} \quad \Pi_{(j,j+1)}\Pi_{i\in S'_{j,j+1}} P(R_i \mid T_j^{j+1}), \quad P_{err} = P_{opt} + \delta_1 + \delta_2 \tag{A.5}$$

Here we also discuss about the places that we take approximation to enhance the computational efficiency in the steps of the previous reduction. From (A.1) to (A.2), we use some heuristics to find out the possible location of SNPs within the whole repeat in which disagreement is observed after several rounds of error correction. From (A.2) to (A.3), we remove all the reads that only span one single SNPs and it has no effect on the error of the detection problem that we are trying to solve. From (A.3) to (A.4), we further partition the reads into group in which $S'_{jk}$ is the set of reads that only span the SNPs j to j+k. Doing this can decompose the ML problem into smaller subproblems with no effect on the accuracy. Finally, in practice, we take a first order approximation of (A.4) to (A.5) by only onsidering two SNPs for each subproblem.

As for each of the marginal probability distribution, the best way is to run Sum-Product algorithm to compute in a dynamic programming fashion similar to S-W alignment. But as pointed out in Quiver, this steps can be significanly speeded up using a Viterbi approximation and this is also what we implemented in the simulation code.

### Simulation study

We simulated on both synthetic and real data set with indel noise and on a double stranded DNA. In the simulation, we assume that the reads from the neighborhood of a repeat is given and our goal is to decide how to extend the reads to span the repeat copies into the

| Repeat Type | $C_s$ | $L_s$ | $C_l$ | $L_l$ | $p_{del}$ | $p_{ins}$ | $G$ | Homology | $l^{approx}$ | $l^{exact}$ | Success % |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Randomly generated | 50X | 100 | 50X | 240 | 10% | 10% | 10000 | 0.67% | 300 | 150 | 99% |
| A repeat of Ecoli-K12 | – | – | 80X | 3000 | 10% | 10% | 4646332 | 0.48% | 5182 | 1507 | 89% |
| A repeat of Bacillus anthracis | – | – | 80X | 3500 | 10% | 10% | 5227293 | 0.23% | 4778 | 2305 | 85% |
| A repeat of Meiothermus ruber | – | – | 80X | 750 | 10% | 10% | 3097457 | 1.40% | 1217 | 257 | 94% |

Table A.1: Simulation results on long contig creator($C_s, L_s$ are coverage and readlength for short reads. $C_l, L_l$ are coverage and readlength for long reads. $p_{del}, p_{ins}$ are the probability of insertion and deletion. $G$ is the length of the genome. Homology is the number of SNPs divided by the length of the approximate repeat. $l^{approx}, l^{exact}$ are the length of the approximate and exact repeat being studied. Success % is the percentage of success in 100 rounds)

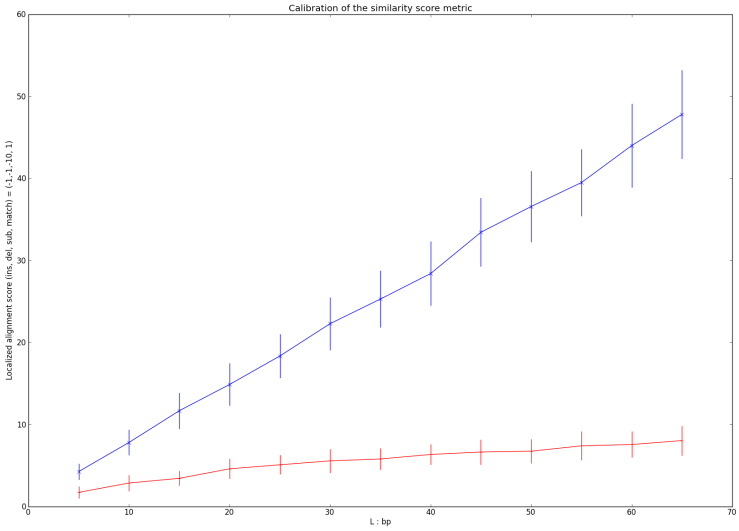



Figure A.5: A calibration for similarity score using global alignment computation.

flanking region correctly. The correctness is evaluated based on whether they can correctly extend the correct reads into the flanking region.

## Edit distance metric calibration

We also do a study on whether we can use alignment score to differentiate between segments from being extracted from the same locus or not. In Fig A.5, the upper curve is the score for segment extracted from the same locus while the bottom curve is for completely iid randomly(irrelevant) generated segment. And we simulate it for 100 times at each length and the bar indicate 1 standard deviation from the mean.

### Tolerance in the Multibridging step

We note that due to the indel noise and the graph surgery that we perform, an X-node of the graph may be p times longer than the usual size of the approximate repeat, thus we should have a corresponding higher tolerance to use the reads to bridge across the repeats.

### Computation speed up of alignment step

The key bottleneck in computation speed of the indel extension is on the pairwise alignment of the reads, which can be speeded up using the ideas in BLAST. We use sorting to identify exact matching fingerprint that identify the starting and ending location of the segment that need to be aligned with. After that, we do a local search instead of the whole dynamic programming search.

## A.4 Appendix: Evidence behind model

### Approximate repeat

We let the underlying genome be $\vec{x}$ and use the short hand that $x[a:b]$ be the $a^{th}$ to $(b-1)^{th}$ entries of $\vec{x}$.

Let $\vec{v_1} = x[s_1 : s_1+l]$ and $\vec{v_2} = x[s_2 : s_2+l]$ be two length $l$ substrings of the genome with starting positions at $s_1$ and $s_2$ respectively. We call $\vec{v_1}$ and $\vec{v_2}$ be an approximate repeat of length $l$ if

$d(x[s_1 - W : s_1], x[s_2 - W : s_2]) \geq 0.7W$

$d(x[s_1 + l : s_1 + l + W], x[s_2 : s_2 + l + W]) \geq 0.7W$

$d(x[s_1 - W + k : s_1 + k], x[s_2 - W + k : s_2 + k]) < 0.7W$ for all $0 < k < l$

To understand approximate repeat better, we plot the Hamming distance for consecutive disjoint window of length 10 as shown in Fig A.6 .

### Classification of approximate repeat

While repeats are studied in the literature[2], they are not investigated by looking at the ground truth. This is partially due to the insufficiency of data in the early days of genome assembly development. Therefore, based on the ground truth genome, we define several quantities that allow us to classify approximate repeat and understand the approximate repeat spectrum of genome. Here we define stretch and mutation rate. Stretch is defined to be the ratio of the length ($l^*$) of the longest exact repeat within an approximate repeat divided by the length ($l_{approx}$) of the approximate repeat. Mutation rate is defined to be number of mutation within approximate repeat divided by ($l_{approx} - l^*$). An illustration is shown in Fig A.6.

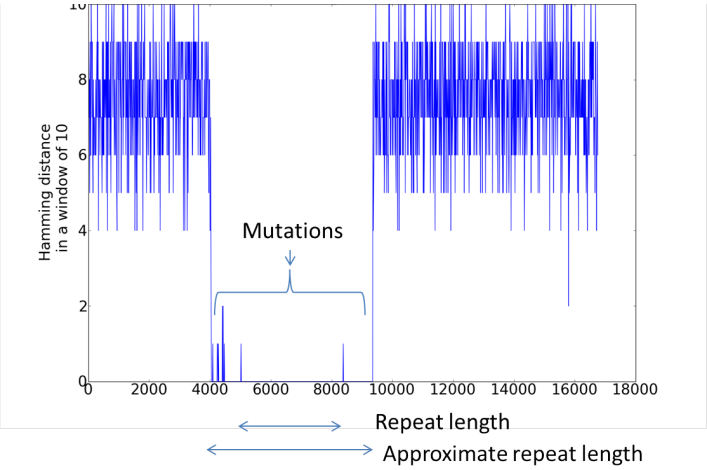



Figure A.6: Example of how to define stretch and mutation rate

Moreover, we do a scatter plot to classify the approximate repeats(approximate repeat having exact repeat length within top 20) and we have a plot of approximate repeat spectrum as in Fig A.7.

From the plots in Fig A.7, we classify approximate repeat as homologous repeat if the stretch is bigger than 1.25 and as non-homologous repeat if the stretch is less than 1.25.

For the scatter plot, every approximate repeat is a dot there with x coordinate and y coordinate being mutation rate and stretch respectively. And the color represents the length of that approximate repeat. For the approximate repeat spectrum plot, the red bar represent non-homogeneous repeat while the blue bar represent homologous repeat. The green dotted line indicates the length of the longest repeat.

We focus on genomes when the non-homologous repeat dominates, namely the longest interleave and the longest triple repeats are non-homologous because the stretch is relatively short which can be captured by our generative model. We do not distinguish between the length of approximate or exact repeat are considered to be the same and we do not distinguish between the two in the discussion because of the small stretch.

## Stopping criterion for defining approximate repeat by MLE estimate

### Parametric model

Let $L_k$ be the number of bases between the $(k-1)^{th}$ and the $k^{th}$ SNPs starting from the right end-point of a repeat.

We consider the following probabilistic model for the $L_k$. $\{L_k\}_{k=1}^{n}$is taken as an indepen-

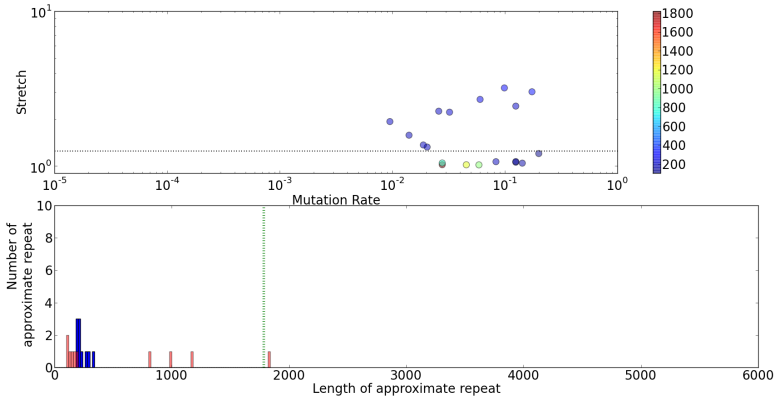



Figure A.7: Classification of approximate repeats and approximate repeat spectrum. The upper plot is scatter plot to classify approximate repeat. The lower plot is the approximate repeat spectrum

dent sequence of geometrically distributed random variables with parameter $\Theta = \{p_1, p_2, r\}$ defined as follows.

$$L_k \sim \begin{cases} Geo(p_1) & \text{if } 1 \le k \le r \\ Geo(p_2) & \text{if } r < k \le n \end{cases}$$

**MLE estimate of parameters**

We now would like to estimate $\Theta$ given the observation of $\{\hat{L}_k\}_{k=1}^n$ by maximum likelihood estimation. Consider the log-likelihood function $L(\Theta) = \log \mathcal{P}(\{\hat{L}_k\}_{k=1}^n \mid \Theta)$.

$$\begin{aligned} L(\Theta) &= \log \mathcal{P}(\{\hat{L}_k\}_{k=1}^n \mid \Theta) && \text{(A.6)} \\ &= \log\{[\Pi_{k=1}^r (1-p_1)^{\hat{L}_k} p_1] \cdot [\Pi_{k=r+1}^n (1-p_2)^{\hat{L}_k} p_2]\} && \text{(A.7)} \\ &= r \log p_1 + [\sum_{k=1}^r \hat{L}_k] \cdot \log(1-p_1) + (n-r) \cdot \log p_2 + [\sum_{k=r+1}^n \hat{L}_k] \cdot \log(1-p_2) && \text{(A.8)} \end{aligned}$$

And we want to find $\hat{\Theta} = \arg\max_\Theta L(\Theta)$.

Observe that, if we fix $1 \le r \le n$, then the optimal $\hat{p_1}$ and $\hat{p_2}$ can be readily obtained by taking derivative on $L(\Theta)$ with respect to $p_1$and $p_2$, specifically,

$$\hat{p_1} = \frac{1}{1 + \frac{\sum_{k=1}^r \hat{L}_k}{r}} \quad \text{(A.9)}$$

$$\hat{p_2} = \frac{1}{1 + \frac{\sum_{k=r+1}^n \hat{L}_k}{n-r}} \quad \text{(A.10)}$$

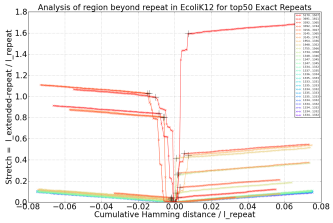



Figure A.8: An example plot that define the stopping point of approximate repeat by Algorithm 21

$\hat{\Theta}$ can then be obtained by running over all integral $1 \leq r \leq n$ and use the corresponding optimal $\hat{p}_1$ and $\hat{p}_2$ to obtain the $L(\Theta)$, and finally we use the $r$ that gives the highest value of $L(\Theta)$ as the MLE estimate given the observation.

**Linear time algorithm to estimate the stopping criterion**

Moreover, this can be done by the following algorithm Algo 21, which run in linear time $\Theta(n)$ with respect to the number of observations $n$.

**Algorithm 21** Linear time algorithm to estimate the stopping criterion

1. a)$C_0 \leftarrow 0$ ; b) for r = 1 to n : $C_r \leftarrow C_r + \hat{L}_r$
2. a)$D_0 \leftarrow C_n$ ; b)for r = 1 to n : $D_r \leftarrow C_n - \hat{L}_r$
3. for r = 1 to n

$\hat{p}_1^{(r)} \leftarrow \frac{1}{1+\frac{C_r}{r}}$

$\hat{p}_2^{(r)} \leftarrow \frac{1}{1+\frac{D_r}{n-r}}$

$\Theta_r \leftarrow (r, \hat{p}_1^{(r)}, \hat{p}_2^{(r)})$

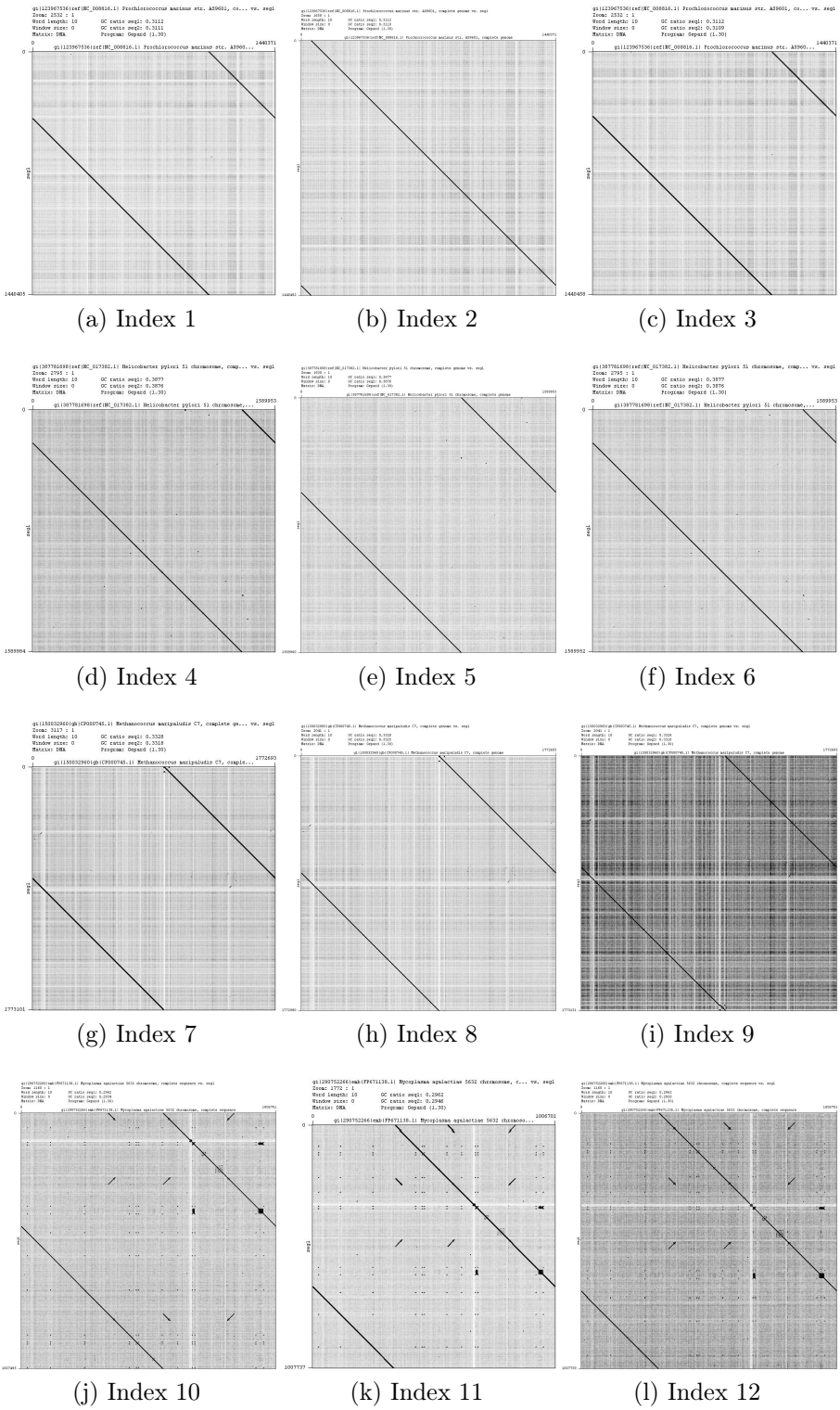
$X_r \leftarrow L(\Theta_r)$

4. find maximum among $\{X_r\}_{r=1}^n$, and the corresponding $\Theta_r$ is the MLE estimate.
5. (Differentiate between homologous and non-homologous repeat)

If the optimal $\hat{p}_1^{(r)}, \hat{p}_2^{(r)}$are too close (i.e. $\hat{p}_1^{(r)} > 0.2$), then claim $r = 1$; else, claim $\hat{r} = r$.

A sample plot is of who we can use the critierion to accurately define the ending of approximate repeat is shown in Fig A.8.

## A.5 Appendix: Dot plots of finished genomes



(a) Index 1 (b) Index 2 (c) Index 3

(d) Index 4 (e) Index 5 (f) Index 6

(g) Index 7 (h) Index 8 (i) Index 9

(j) Index 10 (k) Index 11 (l) Index 12

Figure A.9: Dot plot of recovered genomes against ground truth(according to index in Table 2.2)

# Appendix B

# Appendix of Towards Computation, Space, and Data Efficiency in de novo Genome Assembly

# B.1 Appendix: Extensions to realistic data

The algorithm presented here is based on the simplistic i.i.d. model, and some preliminary results on its extensions towards handling long repeats and errors are provided. Our immediate future goal is to build on this work and develop algorithms that can perform space, time and data efficient sequencing on real data. The two main departures of the i.i.d. model considered here, from the realistic data are that (a) the reads are assumed to be error free here, while in practice they may have indel or substitution errors or missing data, and (b) the underlying sequence may have much longer repeats that what is predicted by the i.i.d. model. The following two subsections describe preliminary thoughts on handling these issues. Also provided are bounds on the system parameters in the presence of substitution errors in the reads.

## Handling repeats

A significant difference between the i.i.d. model and real data is the presence of long repeats in real DNA sequences, which is a very low probability event under the i.i.d. model. In this section, we describe our preliminary work on extending the algorithmic framework described in Section3.3 to handle the case of having long repeats. In a nutshell, the framework of Section3.3 can detect repeats in the online phase. This enables one to avoid any confounding due to repeats in the online phase, and the greedy-yet-efficient algorithm of the offline phase can then handle the repeats. Details follow.

When the underlying sequence has multiple repeats of length $K$ or more, the algorithm of Section3.3 is likely to get confounded, and produce an incorrect output. The following toy example illustrates the kinds of errors that will result in such a situation. Let us assume $G = 34$ and $L = 8$, which gives $K = 4$. Suppose the underlying sequence is $TTTTTAAAAACCCCCAAAAAGGGGGAAAAATCGA$. Then, there is a chance that the algorithm will (incorrectly) output $TTTTTAAAAAGGGGGAAAAACCCCCAAAAATCGA$. This sequence has three repeats $AAAAA$ of length greater than $K = 4$, due to which the algorithm may interchange the data $GGGGG$ and $CCCCC$ between two of the long repeats.

The framework developed in Section3.3 inherently has the ability to *detect* long repeats. To see this, let us first consider an instance of running the algorithm on the toy sequence considered above. Let us assume that the first read is $TTTTAAAA$. The online algorithm will insert this read as the first entry in MergedContig table, and insert the extracted K-mers $AAAA$ and $TTTT$ in the K-mers table. Now suppose the next read is $AAAATCGA$. Then, the K-mer $AAAA$ of this new read will match the $AAAA$ K-mer from the K-mer table. Since the online algorithm detects a length K overlap, it will merge these two reads, and replace the entry of the MergedContig table with $TTTTAAAATCGA$ (and the K-mers table will now have $AAAA$, $TCGA$ and $TTTT$). This is what will subsequently lead to an erroneous output. Let us now see a modification of the algorithm to handle such cases. Suppose the algorithm subsequently observes the read $AAAAGGGG$. The K-mer $AAAA$ from the new read will match the K-mer $AAAA$ in the MergedContig. Thus, the algorithm would try to

merge the two parent contigs $AAAAGGGG$ and $TTTTAAAATCGA$ (treating $AAAA$ as a common sub-string), and would have failed in this process. However, observe that in this process, the algorithm does manage to detect the presence of a repeating $AAAA$, and we shall now exploit this ability of detecting long repeats.

Upon detection of any long repeat ($AAAA$ in this case), the algorithm now separates out the strings in the MergedContig table that contain this repeat as follows. The merged contig containing this repeat is split into two strings, one of which contains the prefix of the repeat followed by the repeat, and the other contains the repeat followed by its suffix. E.g., the string $TTTTAAAATCGA$ in this case would be split into $TTTTAAAA$ and $AAAATCGA$. The MergedContig and the K-mers tables are now updated with these two new strings, and the older string $TTTTAAAATCGA$ is removed. Also, the new read $AAAAGGGG$ that triggered this is also not merged, and is stored as a separate string in MergedContig. At any subsequent time, a new read involving $AAAA$ has the K-mer $AAAA$ matching with multiple entries in MergedConting, and even in this case, a multiple repeat is safely detected. These contigs are subsequently merged in the offline phase in a greedy manner.

We note that these are preliminary ideas, and have not yet been implemented or thoroughly analysed.

## Handling errors

As mentioned previously, in general, our framework operates as follows. Depending on the (stochastic) error model, define a similarity metric between any two reads. Also find a threshold such that under the model considered, there is a vanishing probability of two non-adjacent reads having a similarity greater than that threshold. Now in the online phase, merge two reads whenever their similarity crosses the threshold. (In the algorithm of Section3.3, the distance metric is the amount of contiguous overlap and the threshold is K.) In the offline phase, merge the reads in a greedy-yet efficient manner, exploiting the knowledge that no two of the remaining cotigs gave a similarity greater than the threshold. Finally, run a third phase that performs a consensus operation to obtain a final result from the scaffold construction (this third phase is not required in the absence of errors, or if missing data is the only form of errors). While we leave the general case for future work, below we present (fairly tight) bounds on the system parameters for exact reconstruction in the presence of substitution errors in the reads.

Let us consider the case when the reads might have random substitution errors. As in Section3.2, we assume that the underlying sequence $\vec{x}_G$is generated with each base drawn randomly in an i.i.d. fashion. For simplicity, we assume for now that the distribution for each base is uniform over $\{A,\ C,\ G,\ T\}$. Each read is an $L$ length contiguous substring of $\vec{x}_G$, and is drawn uniformly at random from the entire sequence $\vec{x}_G$. We introduce errors into the model in the following manner. For each read, we assume that each base is randomly and independently substituted by a different base. In particular, we associate a new parameter $p$ to the model, and assume that each base flips to any one of the three other bases with a probability $\frac{p}{3}$ each.

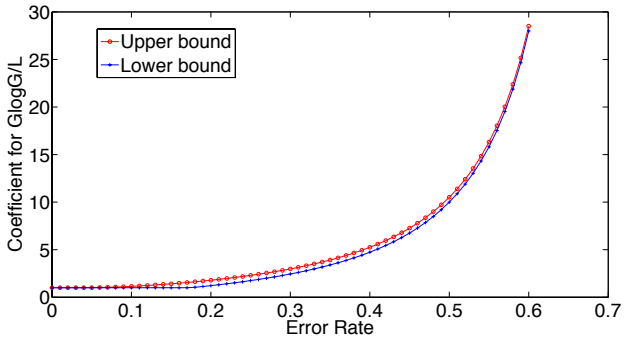



Figure B.1: A graph of the lower and upper bounds to the coefficient of growth, plotted against the error rate. Notice how closely the two bounds follow each other.

Our goal is to find necessary and sufficient conditions on the number of reads $N$ required to allow for exact recovery of $\vec{x}_G$with a high probability. More formally, if $\vec{x}_G$is the actual underlying sequence, and $\vec{x}_G$' is the sequence reconstructed, we wish to have

$$\lim_{G\to\infty} \mathcal{P}(\vec{x}_G \neq \vec{x}'_G) = 0.$$

We obtain a lower bound and an upper bound on minimum data requirement $N^*$ for exact recovery in an asymptotic setting as follows. Let

$$f(p) = \frac{2p(9/8-p)}{(p-3/4)^2}$$

and

$$g(p) = \frac{1}{1-exp(-\frac{1}{f(p)})}.$$

Then,

$$\max\{1, f(p)\} \cdot \frac{G\log G}{L} < N^* < \max\{1, g(p)\} \cdot \frac{G\log G}{L} \tag{B.1}$$

The proof of this result, along with a more detailed discussion on it, is provided in the Appendix. A plot of the bounds is provided in Fig.B.1.

Preliminary thoughts on incorporating substitution errors in the algorithmic framework described in Section3.3 are as follows. We could choose a higher value of the parameter K, and in the online phase merge reads with an overlap of K but allowing for a small fraction of differences between the reads. Likewise in the offline phase, the merge of contigs can be performed allowing for a small fraction of errors. The positions of misconsensus are marked, and a consensus is enforced at the end of the assembly algorithm.

## B.2 Appendix: Bounds on the data requirements in presence of substitution errors

### Previous work

The authors in [38] find the conditions for having the correct alignment on reads. We extend their results by considering a more stringent metric, exact recovery. We also provide an explicit coefficient factor on the coverage requirement for this more stringent metric.

In [38], authors consider a greedy algorithm. We reproduce this algorithm in Algorithm22, along with an additional consensus requirement since our aim is to achieve exact recovery of the underlying sequence.

Let $\alpha^*$ be the value of $\alpha$ that satisfies the following equation

$$\begin{aligned} & D([\alpha, 1-\alpha]||[\frac{3}{4}, \frac{1}{4}]) \\ &= \ 2 \cdot D([\alpha, 1-\alpha]||[2 \cdot p - \frac{4}{3}p^2, 1 - 2 \cdot p - \frac{4}{3}p^2]) \end{aligned} \tag{B.2}$$

In order to correctly align the reads, one can use the following overlap score metric along with the greedy algorithm (Algorithm 22) to correctly align the reads. Let function $d(\cdot, \cdot)$ denote the Hamming distance between its two arguments. We define the score metric between two reads $(\vec{r}'_i, \vec{r}'_j)$ to be the maximum overlap segment length that disagrees by at most a factor $\alpha$ of the overlapping segment. More formally, denoting the score metric by $s(\cdot, \cdot)$:

$$s(\vec{r}'_i, \vec{r}'_j) = \max_{0 \le t \le L} [t \mid d(\vec{r}'_i[L-t:L], \vec{r}'_j[1:t]) \le \alpha t]$$

where we use the notation $\vec{v}[a:b]$ to denote a segment of vector $\vec{r}$ spanning position $a$ to position $b$ of $\vec{r}$ (including the end points). The following lemma presents the precise asymptotic performance guarantees of Algorithm22.

**Lemma B.2.1** ([38]). *When $L > 2/D([\alpha^*, 1-\alpha^*]||[\frac{3}{4}, \frac{1}{4}]) \cdot \ln G$ and $N > \frac{G \ln G}{L}$, the greedy algorithm (Algorithm22) returns the correct alignment with high probability as $G \to \infty$.*

### Main theorem

**Proposition B.2.2.** *If $p < \frac{3}{4}$ and $L > 2/D([\alpha^*, 1-\alpha^*]||[\frac{3}{4}, \frac{1}{4}]) \cdot \ln G$, we obtain a lower bound and an upper bound on the data requirement for exact recovery as follows. Let $f(p) = \frac{2p(9/8-p)}{(p-3/4)^2}$,*

- *If $N < \max\{1, f(p)\} \cdot \frac{G \ln G}{L}$, then $\exists \alpha > 0$ such that $\mathcal{P}(\vec{x}_G \neq \vec{x}'_G) \ge \alpha > 0$*
- *If $N > \max\{1, \frac{1}{1-exp(-\frac{1}{f(p)})}\} \cdot \frac{G \ln G}{L}$, then Algorithm 22 achieves $\mathcal{P}(\vec{x}_G \neq \vec{x}'_G) = 0$*

That is the minimum data requirement on $N^*$ lies in the range of $\max\{1, f(p)\} \cdot \frac{G \ln G}{L} < N^* < \max\{1, \frac{1}{1-exp(-\frac{1}{f(p)})}\} \cdot \frac{G \ln G}{L}$.

This expression is consistent with our intuition on coupon collector problem that the number of reads required should be multiple of $\frac{G \ln G}{L}$. To have more intuition of the coefficient of growth of $\frac{G \ln G}{L}$, we have plotted the lower and upper bounds in Fig.B.1. Indeed, when $p = 0$, it reduces to the coverage constraint without noise, which is consistent of the Lander-Waterman coverage condition[32]. Thus this condition can be viewed as a generalization of Lander-Waterman coverage condition in the presence of noise.

## Proof of propositionB.2.2

**Precondition** Let $(n_1, ..., n_G)$ be the number of reads covering location $(1, ..., G)$ of the underlying sequence $\vec{x}_G$ respectively. Note that $\sum_{i=1}^{G} n_i = N \cdot L$. We first show a result Prop( B.2.3) that allows us to work on the asymptotic regime. Prop (B.2.3) means that the number of reads covering any base of the underlying sequence $\vec{x}_G$ grows with $G$. This is a precondition for us to later use the normal approximation. The precise statement of this precondition is in Prop (B.2.3).

**Proposition B.2.3.** *We have,* $\forall \epsilon' > 0$, *if* $N > \frac{G \ln G}{L}(1 + \epsilon')$,

$$\lim_{G \to \infty} \mathcal{P}(\exists i \text{ such that }, n_i < \ln \ln G) = 0$$

**Algorithm 22** Greedy algorithm with consensus

1. Compute overlap score between all pairs of noisy reads by $s(\vec{r}_i', \vec{r}_j')$
2. Initialize contig list as reads
3. **for** $w = L$ *down to* $0$ **do**
   - **for** *all* $(\vec{r}_i', \vec{r}_j')$ **do**
     - **if** $(\vec{r}_i', \vec{r}_j')$ *do not belong to the same contig and* $s(\vec{r}_i', \vec{r}_j') = w$ **then**
     - **end**
     - Declare $(\vec{r}_i', \vec{r}_j')$ as consecutive read and declare them to belong to the same contig
   - **end**

   **end**
4. Align consecutive reads together to form a long string $\vec{x}_G'$ which is the recovered sequence
5. Do a majority vote from the reads at each location in the $\vec{x}_G'$ to form correctedGenome
6. Return correctedGenome

*Proof.*

$$
\begin{aligned}
&\mathcal{P}(\exists i \text{ such that },n_i < \ln\ln G) \\
&\leq\ G \cdot \mathcal{P}(n_1 < \ln\ln G) && \text{(B.3)} \\
&=\ G \cdot \sum_{i=0}^{\ln\ln G} \left( \begin{array}{c} N \\ i \end{array} \right) (\frac{L}{G})^i (1-\frac{L}{G})^{N-i} && \text{(B.4)} \\
&=\ G \cdot \sum_{i=0}^{\ln\ln G} \frac{1}{i!}[N(N-1)...(N-i+1)(\frac{L}{G})^i] \\
&\quad \cdot[(1-\frac{L}{G})^N (1-\frac{L}{G})^{-i}] \\
&\leq\ G \cdot \sum_{i=0}^{\ln\ln G} \frac{1}{i!}[\frac{NL}{G}]^i \cdot [(e^{-\frac{L}{G}})^N (1-\frac{L}{G})^{-\ln\ln G}] \\
&=\ G \cdot [(1-\frac{L}{G})^{-\ln\ln G}] \cdot \sum_{i=0}^{\ln\ln G} \frac{1}{i!}[(\frac{NL}{G})^i \cdot (e^{-\frac{NL}{G}})] \\
&=\ G \cdot [(1-\frac{L}{G})^{-\ln\ln G}] \cdot \mathcal{P}(\text{Poisson}(\frac{NL}{G}) \leq \ln\ln G) \\
&\leq\ G \cdot [(\frac{1}{e})^{-\ln\ln G}] \cdot \mathcal{P}(\text{Poisson}(\frac{NL}{G}) \leq \ln\ln G) \\
&=\ [G\ln G] \cdot \mathcal{P}(\text{Poisson}(\frac{NL}{G}) \leq \ln\ln G) \\
&\leq\ [G\ln G] \cdot \frac{e^{-\frac{NL}{G}} \cdot (e\frac{NL}{G})^{\ln\ln G}}{(\ln\ln G)^{\ln\ln G}}] && \text{(B.5)} \\
&\leq\ [G\ln G] \cdot [e^{\frac{-NL}{G}} \cdot \ln G \cdot (\frac{NL}{G})^{\ln\ln G}] \\
&=\ [G\ln^2 G] \cdot [e^{\frac{-NL}{G}} \cdot (\frac{NL}{G})^{\ln\ln G}] \\
&\leq\ exp(\ln G - \frac{NL}{G} + \ln(\frac{NL}{G}) \cdot \ln\ln G + 2 \cdot \ln\ln G) \\
& && \text{(B.6)}
\end{aligned}
$$

where(B.3) follows from union bound, (B.4) follows because $n_1 \sim Bin(N, \frac{L}{G})$, and(B.5) follows from the Chernoff bound. Letting $G \to \infty$ in(B.6), we get the desired result. □

**Necessary condition** We now establish the necessary condition on data requirement to have exact recovery of the target genome. That is, if $N < \max(1, f(p))$, then $\mathcal{P}(error) > 0$ as $G \to \infty$.

Let us start by defining some notation. Let

$P_1 = \mathcal{P}(\exists i \text{ such that } n_i = 0)$
$P_2 = \mathcal{P}(\exists i \text{ such that } \vec{x}_G(i) \neq \vec{x}'_G(i) \mid \forall i, n_i > \ln\ln G)$
$P_3 = \mathcal{P}(\forall i, n_i > \ln\ln G)$
In terms of these quantities, the probability of error $\mathcal{P}(\text{error}) \geq P_1 + P_2 \cdot P_3$
If $N < \frac{G\ln G}{L}$, $P_1 \to 1$, thus $\mathcal{P}(\text{error}) \to 1$, which is the Lander-Waterman condition on coverage.

If $N > \frac{G\ln G}{L}$ is satisfied but $N < f(p) \cdot \frac{G\ln G}{L}$, we want to show that $\exists \alpha > 0$ such that $P_2 \geq \alpha > 0$ as $G \to \infty$. This, in turn, suggests $\mathcal{P}(\text{error}) \geq \alpha > 0$ (Remark: the precondition ensures that $P_3 \to 1$).

Now, let us proceed to establish the lower bound on $P_2$ by considering an upper bound on $1 - P_2$. Here we condition on the event that $\{\forall i, n_i > \ln\ln G\}$. For any vector $\vec{v}$, we shall denote its $i^{\text{th}}$ element as $\vec{v}(i)$.

$$\begin{aligned} & 1 - P_2 \\ &= \mathbf{E}_{n_1^G}[\mathcal{P}(\vec{x}'_G(i) = \vec{x}_G(i) \forall i) \mid n_1^G] \\ &\leq \mathbf{E}_{n_1^G}[\Pi_{i=1}^G \mathcal{P}(\vec{x_G}^{MAP}(i) = \vec{x_G}(i)) \mid n_1^G] \\ &\leq \mathbf{E}_{n_1^G}[\Pi_{i=1}^G \mathcal{P}(\text{At location } i, \#\text{reads having A} \\ &\quad > \#\text{reads having C} \mid \vec{x}_G(i) = A) \mid n_1^G] \end{aligned} \tag{B.7}$$

Now let us define random variables

$$Y_j^{(i)} = \begin{cases} 1 & \text{with probability } 1 - p \\ 0 & \text{with probability } \frac{2p}{3} \\ -1 & \text{with probability } \frac{p}{3} \end{cases}$$

We have $\mathbf{E}[Y_j^{(i)}] = 1 - \frac{4p}{3}$, $\mathbf{Var}[Y_j^{(i)}] = 2p \cdot (1 - \frac{8p}{9})$. We proceed the bounding of(B.7), with notation $\Phi(x)$ being the cumulative distribution of standard normal distribution. $\forall \epsilon' > 0$, we have,

$$\begin{aligned} & 1 - P_2 \\ &\leq \mathbf{E}_{n_1^G}[\Pi_{i=1}^G \mathcal{P}(\sum_{j=1}^{n_i} Y_j^{(i)} > 0) \mid n_1^G] \end{aligned}$$

$$
\begin{aligned}
&= \mathbf{E}_{n_1^G}\{\Pi_{i=1}^{G}[1-\mathcal{P}(\sum_{j=1}^{n_i} Y_j^{(i)} \leq 0)] \mid n_1^G\} \\
&\leq \mathbf{E}_{n_1^G}\{\Pi_{i=1}^{G}[1-\frac{1}{2}\Phi(\frac{-n_i(1-\frac{4p}{3})}{\sqrt{n_i 2p\cdot(1-\frac{8p}{9})}})] \mid n_1^G\} \\
&\leq \mathbf{E}_{n_1^G}\{\Pi_{i=1}^{G}[1-exp(n_i\cdot\frac{(1-\epsilon')}{f(p)})] \mid n_1^G\} \qquad \text{(B.8)} \\
&\leq \mathbf{E}_{n_1^G}\{\Pi_{i=1}^{G} exp[-exp(n_i\cdot\frac{(1-\epsilon')}{f(p)})] \mid n_1^G\} \\
&= \mathbf{E}_{n_1^G}\{exp[-\sum_{i=1}^{G} exp(n_i\cdot\frac{(1-\epsilon')}{f(p)})] \mid n_1^G\} \\
&\leq \mathbf{E}_{n_1^G}\{exp[-\frac{G}{(\Pi_{i=1}^{G} exp(n_i\cdot\frac{(1-\epsilon')}{f(p)}))^{\frac{1}{G}}}] \mid n_1^G\} \qquad \text{(B.9)} \\
&= \mathbf{E}_{n_1^G}\{exp[-\frac{G}{(exp(\frac{NL}{G}\cdot\frac{(1-\epsilon')}{f(p)}))}] \mid n_1^G\} \\
&= \mathbf{E}_{n_1^G}\{exp[-G\cdot(exp(-\frac{NL}{G}\cdot\frac{(1-\epsilon')}{f(p)}))] \mid n_1^G\} \\
&= \mathbf{E}_{n_1^G}\{exp[-exp(\ln G-\frac{NL}{G}\cdot\frac{(1-\epsilon')}{f(p)})] \mid n_1^G\}
\end{aligned}
$$

Here,(B.8)) follows from a normal approximation, and(B.9) follows since $A.M. \geq G.M.$. Thus, if $N < \frac{G\ln G}{L}\cdot f(p)$, then $1-P_2 \to 0$ and thus as $G\to\infty$ , $\mathcal{P}(\text{error}) > 0$.

**Sufficient condition** For Algorithm.22, we have,

$$
\begin{aligned}
&\mathcal{P}(\text{error}) \\
&\quad\leq \mathcal{P}(\text{misaligned}) + \mathcal{P}(\text{mis-consensus} \mid \text{correctly aligned})
\end{aligned}
$$

From the results of[38], we know that, $\mathcal{P}(\text{misaligned}) \to 0$ with the parameters given above. Thus it remains to bound the second term. We obtain the following bound, $\forall\epsilon > 0$, when $G$ is sufficiently large, with $\mathcal{E}$ as the precondition

$$
\begin{aligned}
&\mathcal{P}(\text{misconsensus} \mid \text{correctly aligned}) \\
&\leq \mathbf{E}[\sum_{i=1}^{G} \mathcal{P}(\text{misconsensus at location } i \mid \mathcal{E})] + \epsilon \\
&\leq 3\mathbf{E}[\sum_{i=1}^{G} \mathcal{P}(\vec{x}'_G(i) = C \mid \vec{x}_G(i) = A) \mid \mathcal{E}] + \epsilon \\
&\leq 3\mathbf{E}[\sum_{i=1}^{G} \Phi(\frac{-n_i(1-\frac{4}{3}p)}{\sqrt{2pn_i(1-\frac{8}{9}p)}}) \mid \mathcal{E}] + \epsilon \qquad \text{(B.10)} \\
&\leq 3G \cdot \mathbf{E}_{n_1}\{\frac{1}{2}exp[-n_i \cdot \frac{1}{f(p)}]\} + \epsilon \qquad \text{(B.11)} \\
&= \frac{3}{2}G \cdot [1 - \frac{L}{G}(1 - e^{-\frac{1}{f(p)}})]^N + \epsilon \\
&\leq \frac{3}{2}exp[\ln G - \frac{LN}{G}(1 - e^{-\frac{1}{f(p)}})] + \epsilon
\end{aligned}
$$

Here,(B.10) follows from a normal approximation,(B.11) results from bounding the $\Phi$ function. Thus, if the conditions are satisfied, $\mathcal{P}(\text{error}) \to 0$ as $G \to \infty$.

# Appendix C

# Appendix of FinisherSC

# C.1 Appendix: Detailed experimental results on bacterial genomes

In this section, we provide the detailed Quast report for the results described in Table 4.2. Moreover, we compare in Fig C.1 the memory consumption and running time of FinisherSC with those of PBJelly. The computing experiments for this section were performed on the genepool cluster at JGI. Below are commands used to run PBJelly.

```
Jelly.py setup Protocol.xml -x "-minGap=1"
Jelly.py mapping Protocol.xml
Jelly.py support Protocol.xml -x " -debug"
Jelly.py extraction Protocol.xml
Jelly.py assembly Protocol.xml -x "-nproc=16"
Jelly.py output Protocol.xml
```

The BLASR configuration in Protocol.xml is

```
-minMatch 8 -minPctIdentity 70 -bestn 8
-nCandidates 30 -maxScore -500 -nproc 16
-noSplitSubreads
```

## Details of the scalability experiments

We run the scalability experiments on a server computer, which is equipped with 64 cores of CPU at clock rate of 2.4-3.3GHz and 512GB of RAM. We also note that, for even larger contig or read data with genomes of higher repeat content, one may be interested in the following options. They are [-f True] for fast alignment and [-l True] for breaking down large contig file. As a reference, we also attach the Quast analysis results on all the intermediate output for the scalability test in Table C.6, C.7 and C.8. We note that the misassembly count in the Quast analysis for these genomes should only be used as a reference because there is a lack of high quality reference and reference genomes may be from different strains.

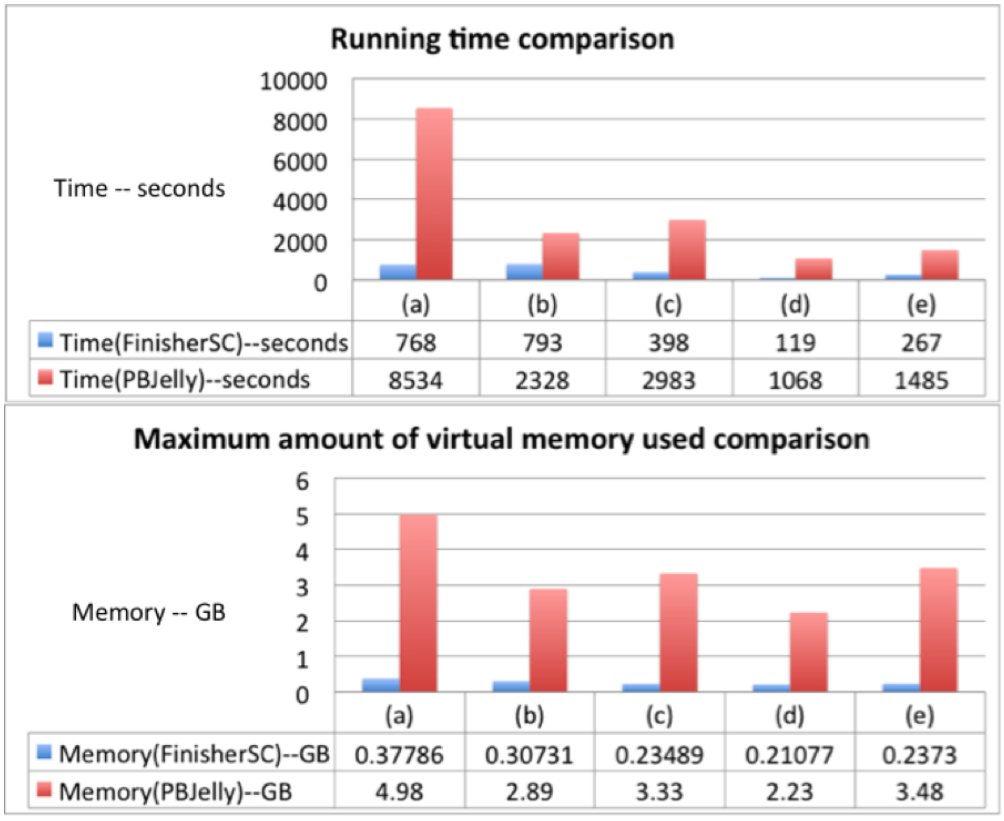



Figure C.1: Running time and memory consumption comparison of FinisherSC and PBJelly . (a) to (e) are the corresponding data sets in Table 4.2.

Table C.1: (a) in Table 4.2. All statistics are based on contigs of size ≥ 500 bp, unless otherwise noted (e.g., "# contigs (≥ 0 bp)" and "Total length (≥ 0 bp)" include all contigs).

| Assembly | HGAP | FinisherSC | PBJelly |
|---|---|---|---|
| # contigs (≥ 0 bp) | 45 | **4** | 44 |
| # contigs (≥ 1000 bp) | 45 | **4** | 44 |
| Total length (≥ 0 bp) | 5340498 | 5212355 | **5**383836 |
| Total length (≥ 1000 bp) | 5340498 | 5212355 | **5**383836 |
| # contigs | 45 | **4** | 44 |
| Largest contig | 4097401 | **5**168551 | 4099674 |
| Total length | 5340498 | 5212355 | **5**383836 |
| Reference length | 5167383 | 5167383 | 5167383 |
| GC (%) | 42.16 | 42.06 | 42.19 |
| Reference GC (%) | 42.05 | 42.05 | 42.05 |
| N50 | 4097401 | **5**168551 | 4099674 |
| NG50 | 4097401 | **5**168551 | 4099674 |
| N75 | 4097401 | **5**168551 | 4099674 |
| NG75 | 4097401 | **5**168551 | 4099674 |
| L50 | 1 | 1 | 1 |
| LG50 | 1 | 1 | 1 |
| L75 | 1 | 1 | 1 |
| LG75 | 1 | 1 | 1 |
| # misassemblies | **1** | **1** | 3 |
| # misassembled contigs | **1** | **1** | 2 |
| Misassembled contigs length | **9**679 | **9**679 | 4117533 |
| # local misassemblies | **2** | 3 | 4 |
| # unaligned contigs | 39 + 0 part | **1** + 0 part | 39 + 0 part |
| Unaligned length | 135514 | **1**7453 | 163702 |
| Genome fraction (%) | 100.000 | 100.000 | 100.000 |
| Duplication ratio | 1.007 | **1**.005 | 1.010 |
| # N's per 100 kbp | 4.25 | **0**.48 | 4.46 |
| # mismatches per 100 kbp | 9.56 | **1**.30 | 10.14 |
| # indels per 100 kbp | 92.62 | **5**4.90 | 94.75 |
| Largest alignment | 4097400 | **5**168480 | 4098915 |
| NA50 | 4097400 | **5**168480 | 4098915 |
| NGA50 | 4097400 | **5**168480 | 4098915 |
| NA75 | 4097400 | **5**168480 | 4098915 |
| NGA75 | 4097400 | **5**168480 | 4098915 |
| LA50 | 1 | 1 | 1 |
| LGA50 | 1 | 1 | 1 |
| LA75 | 1 | 1 | 1 |
| LGA75 | 1 | 1 | 1 |

Table C.2: (b) in Table 4.2. All statistics are based on contigs of size ≥ 500 bp, unless otherwise noted (e.g., "# contigs (≥ 0 bp)" and "Total length (≥ 0 bp)" include all contigs).

| Assembly | HGAP | FinisherSC | PBJelly |
|---|---|---|---|
| # contigs (≥ 0 bp) | 163 | **4**1 | 115 |
| # contigs (≥ 1000 bp) | 163 | **4**1 | 115 |
| Total length (≥ 0 bp) | 5536634 | 5139491 | **5**821106 |
| Total length (≥ 1000 bp) | 5536634 | 5139491 | **5**821106 |
| # contigs | 163 | **4**1 | 115 |
| Largest contig | 254277 | **6**37485 | 495596 |
| Total length | 5536634 | 5139491 | **5**821106 |
| Reference length | 5167383 | 5167383 | 5167383 |
| GC (%) | 41.98 | 41.96 | 42.01 |
| Reference GC (%) | 42.05 | 42.05 | 42.05 |
| N50 | 89239 | **2**15810 | 145441 |
| NG50 | 94672 | **2**15810 | 161517 |
| N75 | 44568 | **1**17879 | 98297 |
| NG75 | 53723 | **1**17879 | 116800 |
| L50 | 20 | **9** | 14 |
| LG50 | 18 | **9** | 12 |
| L75 | 42 | **1**7 | 26 |
| LG75 | 36 | **1**7 | 21 |
| # misassemblies | **0** | **0** | 12 |
| # misassembled contigs | **0** | **0** | 10 |
| Misassembled contigs length | **0** | **0** | 439591 |
| # local misassemblies | **0** | 3 | 3 |
| # unaligned contigs | 46 + 1 part | **1** + 0 part | 43 + 22 part |
| Unaligned length | 200727 | **1**1862 | 302804 |
| Genome fraction (%) | 98.727 | 98.957 | **9**9.964 |
| Duplication ratio | 1.046 | **1**.003 | 1.068 |
| # N's per 100 kbp | 15.64 | **6**.17 | 9.50 |
| # mismatches per 100 kbp | **6**3.00 | 67.78 | 68.92 |
| # indels per 100 kbp | **5**77.98 | 589.72 | 597.99 |
| Largest alignment | 254274 | **6**37485 | 495589 |
| NA50 | 89239 | **2**15810 | 145441 |
| NGA50 | 94672 | **2**15810 | 161490 |
| NA75 | 44567 | **1**17879 | 98293 |
| NGA75 | 50860 | **1**17879 | 115834 |
| LA50 | 20 | **9** | 14 |
| LGA50 | 18 | **9** | 12 |
| LA75 | 42 | **1**7 | 26 |
| LGA75 | 36 | **1**7 | 21 |

Table C.3: (c) in Table 4.2. All statistics are based on contigs of size ≥ 500 bp, unless otherwise noted (e.g., "# contigs (≥ 0 bp)" and "Total length (≥ 0 bp)" include all contigs).

| Assembly | HGAP | FinisherSC | PBJelly |
|---|---|---|---|
| # contigs (≥ 0 bp) | 21 | **7** | 14 |
| # contigs (≥ 1000 bp) | 21 | **7** | 14 |
| Total length (≥ 0 bp) | 4689701 | 4660679 | **4**718818 |
| Total length (≥ 1000 bp) | 4689701 | 4660679 | **4**718818 |
| # contigs | 21 | **7** | 14 |
| Largest contig | 1241016 | **2**044060 | 1958341 |
| Total length | 4689701 | 4660679 | **4**718818 |
| Reference length | 4639221 | 4639221 | 4639221 |
| GC (%) | 50.87 | 50.85 | 50.85 |
| Reference GC (%) | 50.79 | 50.79 | 50.79 |
| N50 | 392114 | **1**525398 | 1200847 |
| NG50 | 392114 | **1**525398 | 1200847 |
| N75 | 252384 | **1**525398 | 275618 |
| NG75 | 252384 | **1**525398 | 321636 |
| L50 | 3 | **2** | **2** |
| LG50 | 3 | **2** | **2** |
| L75 | 7 | **2** | 4 |
| LG75 | 7 | **2** | 3 |
| # misassemblies | **8** | **8** | 12 |
| # misassembled contigs | 4 | **3** | 5 |
| Misassembled contigs length | **2**530799 | 3584781 | 3672462 |
| # local misassemblies | **3** | **3** | 4 |
| # unaligned contigs | 0 + 1 part | **0** + 0 part | 0 + 3 part |
| Unaligned length | 205 | **0** | 2605 |
| Genome fraction (%) | 99.583 | 99.656 | **9**9.689 |
| Duplication ratio | 1.015 | **1**.008 | 1.021 |
| # N's per 100 kbp | 0.00 | 0.00 | 0.00 |
| # mismatches per 100 kbp | **3**.66 | 4.61 | 4.11 |
| # indels per 100 kbp | **8**.36 | 13.82 | 10.75 |
| Largest alignment | 683967 | **1**094192 | 949307 |
| NA50 | 339478 | **8**60437 | 685586 |
| NGA50 | 339478 | **8**60437 | 685586 |
| NA75 | 229039 | **3**78942 | 255377 |
| NGA75 | 229039 | **3**78942 | 255377 |
| LA50 | 5 | **3** | **3** |
| LGA50 | 5 | **3** | **3** |
| LA75 | 9 | **5** | 7 |
| LGA75 | 9 | **5** | 7 |

Table C.4: (d) in Table 4.2. All statistics are based on contigs of size ≥ 500 bp, unless otherwise noted (e.g., "# contigs (≥ 0 bp)" and "Total length (≥ 0 bp)" include all contigs).

| Assembly | HGAP | FinisherSC | PBJelly |
|---|---|---|---|
| # contigs (≥ 0 bp) | 3 | **1** | 2 |
| # contigs (≥ 1000 bp) | 3 | **1** | 2 |
| Total length (≥ 0 bp) | 3102769 | 3099349 | **3**106774 |
| Total length (≥ 1000 bp) | 3102769 | 3099349 | **3**106774 |
| # contigs | 3 | **1** | 2 |
| Largest contig | 1390744 | **3**099349 | 1715191 |
| Total length | 3102769 | 3099349 | **3**106774 |
| Reference length | 3097457 | 3097457 | 3097457 |
| GC (%) | 63.38 | 63.39 | 63.38 |
| Reference GC (%) | 63.38 | 63.38 | 63.38 |
| N50 | 1053479 | **3**099349 | 1715191 |
| NG50 | 1053479 | **3**099349 | 1715191 |
| N75 | 1053479 | **3**099349 | 1391583 |
| NG75 | 1053479 | **3**099349 | 1391583 |
| L50 | 2 | **1** | **1** |
| LG50 | 2 | **1** | **1** |
| L75 | 2 | **1** | 2 |
| LG75 | 2 | **1** | 2 |
| # misassemblies | 0 | 0 | 0 |
| # misassembled contigs | 0 | 0 | 0 |
| Misassembled contigs length | 0 | 0 | 0 |
| # local misassemblies | 2 | 2 | 2 |
| # unaligned contigs | 0 + 0 part | 0 + 0 part | 0 + 0 part |
| Unaligned length | 0 | 0 | 0 |
| Genome fraction (%) | 99.966 | **9**9.986 | **9**9.986 |
| Duplication ratio | 1.002 | **1**.001 | 1.003 |
| # N's per 100 kbp | 0.00 | 0.00 | 0.00 |
| # mismatches per 100 kbp | **0**.03 | 0.26 | 0.10 |
| # indels per 100 kbp | **1**.10 | 3.87 | 1.32 |
| Largest alignment | 1390744 | **3**099004 | 1713236 |
| NA50 | 1053134 | **3**099004 | 1713236 |
| NGA50 | 1053134 | **3**099004 | 1713236 |
| NA75 | 1053134 | **3**099004 | 1391558 |
| NGA75 | 1053134 | **3**099004 | 1391558 |
| LA50 | 2 | **1** | **1** |
| LGA50 | 2 | **1** | **1** |
| LA75 | 2 | **1** | 2 |
| LGA75 | 2 | **1** | 2 |

Table C.5: (e) in Table 4.2. All statistics are based on contigs of size ≥ 500 bp, unless otherwise noted (e.g., "# contigs (≥ 0 bp)" and "Total length (≥ 0 bp)" include all contigs).

| Assembly | HGAP | FinisherSC | PBJelly |
|---|---|---|---|
| # contigs (≥ 0 bp) | 18 | **5** | 8 |
| # contigs (≥ 1000 bp) | 18 | **5** | 8 |
| Total length (≥ 0 bp) | 5184825 | 5167414 | **5**210862 |
| Total length (≥ 1000 bp) | 5184825 | 5167414 | **5**210862 |
| # contigs | 18 | **5** | 8 |
| Largest contig | 2103385 | 2913716 | **3**343452 |
| Total length | 5184825 | 5167414 | **5**210862 |
| Reference length | 5167383 | 5167383 | 5167383 |
| GC (%) | 42.05 | 42.05 | 42.07 |
| Reference GC (%) | 42.05 | 42.05 | 42.05 |
| N50 | 1403814 | 2913716 | **3**343452 |
| NG50 | 1403814 | 2913716 | **3**343452 |
| N75 | 790287 | **2**225895 | 1814491 |
| NG75 | 790287 | **2**225895 | 1814491 |
| L50 | 2 | **1** | **1** |
| LG50 | 2 | **1** | **1** |
| L75 | 3 | **2** | **2** |
| LG75 | 3 | **2** | **2** |
| # misassemblies | **1** | **1** | 2 |
| # misassembled contigs | **1** | **1** | 2 |
| Misassembled contigs length | **1**403814 | 2913716 | 1820739 |
| # local misassemblies | **0** | **0** | 1 |
| # unaligned contigs | **0** + 0 part | **0** + 0 part | 0 + 3 part |
| Unaligned length | **0** | **0** | 13698 |
| Genome fraction (%) | 99.900 | 99.934 | **9**9.954 |
| Duplication ratio | 1.005 | **1**.001 | 1.007 |
| # N's per 100 kbp | 0.00 | 0.00 | 0.00 |
| # mismatches per 100 kbp | 3.41 | 3.56 | **2**.28 |
| # indels per 100 kbp | **2**.91 | 5.31 | 5.21 |
| Largest alignment | 2103385 | 2225895 | **3**343452 |
| NA50 | 1259090 | 1656831 | **3**343452 |
| NGA50 | 1259090 | 1656831 | **3**343452 |
| NA75 | 790287 | **1**656831 | 1270970 |
| NGA75 | 790287 | **1**656831 | 1270970 |
| LA50 | 2 | 2 | **1** |
| LGA50 | 2 | 2 | **1** |
| LA75 | 3 | **2** | **2** |
| LGA75 | 3 | **2** | **2** |

Table C.6: Quast analysis report of Caenorhabditis elegans

| Assembly | contigs | noEmbed | improved | improved2 | improved3 |
|---|---|---|---|---|---|
| # contigs (≥ 0 bp) | 245 | 237 | 207 | 200 | **1**96 |
| # contigs (≥ 1000 bp) | 245 | 237 | 207 | 200 | **1**96 |
| Total length (≥ 0 bp) | **1**04169699 | 103975584 | 103804734 | 103826835 | 103822380 |
| Total length (≥ 1000 bp) | **1**04169699 | 103975584 | 103804734 | 103826835 | 103822380 |
| # contigs | 245 | 237 | 207 | 200 | **1**96 |
| Largest contig | 3165643 | 3165643 | **4**217234 | **4**217234 | **4**217234 |
| Total length | **1**04169699 | 103975584 | 103804734 | 103826835 | 103822380 |
| Reference length | 100286401 | 100286401 | 100286401 | 100286401 | 100286401 |
| GC (%) | 35.67 | 35.66 | 35.66 | 35.66 | 35.66 |
| Reference GC (%) | 35.44 | 35.44 | 35.44 | 35.44 | 35.44 |
| N50 | 1613475 | 1613475 | 1773224 | **1**832604 | **1**832604 |
| NG50 | 1619480 | 1619480 | 1832604 | **1**910174 | **1**910174 |
| N75 | 834223 | 834223 | 947068 | **9**59597 | **9**59597 |
| NG75 | 881109 | 881109 | 959597 | **1**031901 | **1**031901 |
| L50 | 24 | 24 | 22 | **2**1 | **2**1 |
| LG50 | 23 | 23 | 21 | **2**0 | **2**0 |
| L75 | 48 | 48 | 43 | **4**1 | **4**1 |
| LG75 | 44 | 44 | 40 | **3**8 | **3**8 |
| # misassemblies | 1358 | **1**334 | 1411 | 1407 | 1403 |
| # misassembled contigs | 127 | 120 | 108 | 104 | **1**02 |
| Misassembled contigs length | 96744440 | **9**6571278 | 98809318 | 99454743 | 99479049 |
| # local misassemblies | 784 | **7**83 | 809 | 805 | 810 |
| # unaligned contigs | 70 + 20 part | 70 + 20 part | 70 + 11 part | 68 + 11 part | **6**8 + 10 part |
| Unaligned length | 1862487 | 1862487 | 1743676 | 1745585 | **1**743404 |
| Genome fraction (%) | 99.525 | 99.525 | 99.531 | **9**9.535 | **9**9.535 |
| Duplication ratio | 1.027 | 1.025 | **1**.024 | **1**.024 | **1**.024 |
| # N's per 100 kbp | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| # mismatches per 100 kbp | 13.27 | 13.27 | 13.23 | 13.20 | **1**3.13 |
| # indels per 100 kbp | 20.89 | 20.89 | **2**0.63 | 20.80 | 20.86 |
| Largest alignment | 1362562 | 1362562 | 1362562 | 1362562 | 1362562 |
| NA50 | 407833 | 407833 | **4**07888 | 402635 | 402635 |
| NGA50 | **4**21272 | **4**21272 | **4**21272 | 412370 | 412370 |
| NA75 | 213728 | **2**17528 | 213728 | 211961 | 211961 |
| NGA75 | **2**32043 | **2**32043 | 229862 | 226413 | 226413 |
| LA50 | 85 | 85 | 85 | 85 | 85 |
| LGA50 | 80 | 80 | 80 | 80 | 80 |
| LA75 | 169 | **1**68 | **1**68 | 169 | 169 |
| LGA75 | **1**56 | **1**56 | **1**56 | 157 | 157 |

Table C.7: Quast analysis report of Drosophila

| Assembly | contigs | noEmbed | improved | improved2 | improved3 |
|---|---|---|---|---|---|
| # contigs (≥ 0 bp) | 128 | 128 | 110 | 96 | **9**3 |
| # contigs (≥ 1000 bp) | 128 | 128 | 110 | 96 | **9**3 |
| Total length (≥ 0 bp) | **1**38490501 | **1**38490501 | 138082066 | 138131721 | 138096303 |
| Total length (≥ 1000 bp) | **1**38490501 | **1**38490501 | 138082066 | 138131721 | 138096303 |
| # contigs | 128 | 128 | 110 | 96 | **9**3 |
| Largest contig | 24648237 | 24648237 | **2**7967410 | **2**7967410 | **2**7967410 |
| Total length | **1**38490501 | **1**38490501 | 138082066 | 138131721 | 138096303 |
| Reference length | 168717020 | 168717020 | 168717020 | 168717020 | 168717020 |
| GC (%) | 41.86 | 41.86 | 41.87 | 41.87 | 41.87 |
| Reference GC (%) | 41.74 | 41.74 | 41.74 | 41.74 | 41.74 |
| N50 | 15305620 | 15305620 | **2**1710673 | **2**1710673 | **2**1710673 |
| NG50 | 6168915 | 6168915 | **1**5305620 | **1**5305620 | **1**5305620 |
| N75 | 920983 | 920983 | 1012145 | **1**448033 | **1**448033 |
| NG75 | 291391 | 291391 | 308285 | 389196 | **4**02690 |
| L50 | 4 | 4 | **3** | **3** | **3** |
| LG50 | 6 | 6 | **4** | **4** | **4** |
| L75 | 15 | 15 | 11 | **1**0 | **1**0 |
| LG75 | 57 | 57 | 50 | 42 | **4**1 |
| # misassemblies | 8272 | 8272 | **8**057 | 8071 | 8064 |
| # misassembled contigs | 98 | 98 | 84 | 76 | **7**3 |
| Misassembled contigs length | **1**27872011 | **1**27872011 | 130423069 | 130352009 | 130316591 |
| # local misassemblies | 290 | 290 | **2**82 | 283 | 284 |
| # unaligned contigs | 1 + 0 part | 1 + 0 part | 1 + 0 part | 1 + 0 part | 1 + 0 part |
| Unaligned length | 61383 | 61383 | 61383 | 61383 | 61383 |
| Genome fraction (%) | **7**6.376 | **7**6.376 | 76.348 | 76.367 | 76.364 |
| Duplication ratio | 1.082 | 1.082 | **1**.079 | **1**.079 | **1**.079 |
| # N's per 100 kbp | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| # mismatches per 100 kbp | 21.85 | 21.85 | **2**1.34 | 21.81 | 21.84 |
| # indels per 100 kbp | 20.88 | 20.88 | **2**0.81 | 20.94 | 20.99 |
| Largest alignment | 6494798 | 6494798 | 6494798 | 6494798 | 6494798 |
| NA50 | 1316520 | 1316520 | 1316520 | **1**377072 | **1**377072 |
| NGA50 | 820949 | 820949 | 821097 | **8**29069 | **8**29069 |
| NA75 | 308622 | 308622 | 341648 | **3**98564 | **3**98564 |
| NGA75 | **6**441 | **6**441 | 6389 | 6414 | 6384 |
| LA50 | 26 | 26 | 26 | 26 | 26 |
| LGA50 | 41 | 41 | 41 | **4**0 | **4**0 |
| LA75 | 78 | 78 | 76 | **7**2 | **7**2 |
| LGA75 | 532 | 532 | 533 | **5**20 | 522 |

Table C.8: Quast analysis report of Saccharomyces cerevisiae

| Assembly | contigs | noEmbed | improved | improved2 | improved3 |
|---|---|---|---|---|---|
| # contigs (≥ 0 bp) | 30 | 27 | 23 | 22 | **2**0 |
| # contigs (≥ 1000 bp) | 30 | 27 | 23 | 22 | **2**0 |
| Total length (≥ 0 bp) | **1**2370681 | 12295446 | 12221471 | 12225040 | 12220764 |
| Total length (≥ 1000 bp) | **1**2370681 | 12295446 | 12221471 | 12225040 | 12220764 |
| # contigs | 30 | 27 | 23 | 22 | **2**0 |
| Largest contig | 1538192 | 1538192 | 1538192 | 1538192 | 1538192 |
| Total length | **1**2370681 | 12295446 | 12221471 | 12225040 | 12220764 |
| Reference length | 12157105 | 12157105 | 12157105 | 12157105 | 12157105 |
| GC (%) | 38.21 | 38.18 | 38.17 | 38.17 | 38.17 |
| Reference GC (%) | 38.15 | 38.15 | 38.15 | 38.15 | 38.15 |
| N50 | 777787 | 777787 | 777787 | 777787 | 777787 |
| NG50 | 777787 | 777787 | 777787 | 777787 | 777787 |
| N75 | 544615 | 544615 | 544615 | 544615 | **5**83193 |
| NG75 | 544615 | 544615 | 544615 | 544615 | **5**83193 |
| L50 | 6 | 6 | 6 | 6 | 6 |
| LG50 | 6 | 6 | 6 | 6 | 6 |
| L75 | 11 | 11 | 11 | 11 | 11 |
| LG75 | 11 | 11 | 11 | 11 | 11 |
| # misassemblies | 112 | 109 | **1**06 | **1**06 | 107 |
| # misassembled contigs | 27 | 24 | 20 | 20 | **1**8 |
| Misassembled contigs length | 11049515 | 10974280 | 10900305 | 10900305 | **1**0896029 |
| # local misassemblies | 21 | 21 | 21 | 21 | 21 |
| # unaligned contigs | 0 + 0 part | 0 + 0 part | 0 + 0 part | 0 + 0 part | 0 + 0 part |
| Unaligned length | 0 | 0 | 0 | 0 | 0 |
| Genome fraction (%) | 98.117 | 98.117 | 98.215 | **9**8.243 | **9**8.243 |
| Duplication ratio | 1.038 | 1.032 | **1**.024 | **1**.024 | **1**.024 |
| # N's per 100 kbp | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| # mismatches per 100 kbp | **7**6.68 | **7**6.68 | 78.86 | 79.14 | 78.05 |
| # indels per 100 kbp | 11.71 | 11.71 | **1**1.67 | 13.58 | 13.50 |
| Largest alignment | 1027016 | 1027016 | 1027016 | **1**053518 | **1**053518 |
| NA50 | **3**77112 | **3**77112 | 377016 | 377016 | 377016 |
| NGA50 | **3**77112 | **3**77112 | 377016 | 377016 | 377016 |
| NA75 | 181420 | **1**98393 | **1**98393 | **1**98393 | **1**98393 |
| NGA75 | 198393 | 198393 | 198393 | 198393 | 198393 |
| LA50 | 11 | 11 | 11 | 11 | 11 |
| LGA50 | 11 | 11 | 11 | 11 | 11 |
| LA75 | 23 | **2**2 | **2**2 | **2**2 | **2**2 |
| LGA75 | 22 | 22 | 22 | 22 | 22 |

# Appendix D

# Appendix of BIGMAC

## D.1 Appendix: Outline of the appendix

This appendix includes the following sections.

1. Implementation details of the break point finding algorithm
2. Data analysis of the Breaker and Merger
3. Feasbility of Breaker to recover consistent contigs
4. More information on the EM algorithm and the MSA
5. Commands used to run various tools
6. Detailed Quast reports

## D.2 Appendix: Implementation details of the break point finding algorithm

In forming a De Bruijn graph, we use the following method. First, we fill in the hidden end points by inspecting any inconsistent number of end points between repeat copies. In our example of $x_1 = a[b(c]d)e, x_2 = f[bc]g, x_3 = h(cd)i$, we have [()] as the long($\geq 2L$) repeat end points. We fill in the hidden end points $x_1 = a[b(c]d)e, x_2 = f[b|c]g, x_3 = h(c|d)i$ because between [] there should be a ), and between (), there should be a ]After filling in the hidden end points, we label and cluster the end points. At first, two end points have the same label if they correspond to the same side of the same repeat. Then, we cluster end points that are close to each other to have the same label. With the relabelled end points along each contig, we form a graph. Note that the end points correspond to edges of the graph. In the previous example, let the label of end point of [(]) be $1, 2, 3, 4$ respectively, we have the edge sequences of $x_1, x_2, x_3$ being $(1, 2, 3, 4), (1, 2, 3), (2, 3, 4)$. And we will append beginning and ending edge to the sequences, so the actual edge sequences of $x_1, x_2, x_3$ are $(b_1, 1, 2, 3, 4, e_1), (b_2, 1, 2, 3, e_2), (b_3, 2, 3, 4, e_3)$. Next, we need to find the nodes. This can be done by scannig for successive end points in the edge sequences. Any two successive end points define a node. And if they do not correspond to a closed end point followed by an open end point, it is considered as a repeat node. For example, $(1, 2)$ is a repeat node node, and $(b_1, 1)$ is a non-repeat node. Now we note that from the repeat nodes, we can gather together the edges to form the graph. For example, the incoming edges of node $(1, 2)$ are the two end points corresponding to 1 and outgoing edges of the node $(1, 2)$ are the two end points corresponding to 2. In order to handle double stranded nature of the genome, when scanning the edge sequences, we search both forward and backward to identify the nodes. The approximate nature of matching is handled when we cluster end points close to each other.

## D.3 Appendix: Data analysis of the Breaker and Merger

We perform independent data analysis of the performance of Breaker and Merger of BIGMAC. We note that we both use QUAST and an independent evaluation(which is implemented by us) from QUAST. Users can use our evaluation scripts to evaluate the performance of their own improvement as well. We note that the dataset 1,2,3 are those studied in the experiment section and the dataset 0 is the synthetic dataset.

### Quast reports

The Breaker only and BIGMAC end-to-end results are tabulated as follows. We note that Breaker can decrease the number of contigs because it remove redundant contigs after breaking at potentially mis-assembled points. The are located at the QUAST report section.

### Data anaysis on Breaker

We measure mis-assemblies fixing capability of Breaker. Specifically, we study the performance of ChimericContigFixing(Palindrome) and the combination of LocatePotentialMisassemblies and ConfirmBreakPoints (Repeat&Coverage). We map the contigs back to the ground truth to see if the segments mapped to different locations. We note that our method is more stringent that QUAST. Even in the cases of repeat, we only map the segment to the best matched location. Thus, occasionally, a FP may not be a real false positive. The script can be run as python -m srcRefactor.evalmfixer foldername mummerpath The precision and recall on the subcompoents are as follows.

Table D.1: Breaker Evaluation

| Dataset | Break point detector | Precision | Recall | Number of TP | Number of FP |
|---|---|---|---|---|---|
| 0 | Palindrome | 1 | 0 | 0 | 0 |
| 0 | Repeat&Coverage | 1 | 1 | 2 | 0 |
| 1 | Palindrome | 1 | 0 | 0 | 0 |
| 1 | Repeat&Coverage | 0.102041 | 0.483871 | 15 | 132 |
| 2 | Palindrome | 0.605556 | 0.246606 | 109 | 71 |
| 2 | Repeat&Coverage | 0.021898 | 0.032967 | 9 | 402 |
| 3 | Palindrome | 0.818182 | 0.157895 | 9 | 2 |
| 3 | Repeat&Coverage | 0.142857 | 0.113636 | 5 | 30 |

### Data anaysis on Merger

To evaluation, we collect data from graphsurgery merges(when condensing edges), BRepeat merges(when repeat node is not a separte node) and XRepeat merges(when repeat node is a separate node). We map back to reference to identify correct successors. Then we report the percentage left. The scripts can be run as python -m srcRefactor.evalasplitter foldername mummerpath The precision and recall on the subcompoents are as follows. Note that we are more stringent than QUAST, because if two are not immediate successors then we report as FP here. Also, we use best match on the reference, meaning that repeat can be mapped to more than one location, thus a FP may not really be a FP. So, the number reported only serves as an approximation here. We note that we have duplicated tje contigs to handle reverse complements, so all numbers are approximately double of the actual number, with some offset due to slight variation due to tie-breaking in the alignment tool.

Table D.2: Merger Evaluation

| Dataset | Merger subroutine | precision | recall | TP_num | FP_num |
|---|---|---|---|---|---|
| 0 | GraphSurgery | 1 | 0 | 0 | 0 |
| 0 | BResolve | 1 | 1 | 4 | 0 |
| 0 | XResolve | 1 | 0 | 0 | 0 |
| 1 | GraphSurgery | 0.829268 | 0.164251 | 68 | 14 |
| 1 | BResolve | 0.745455 | 0.099034 | 41 | 14 |
| 1 | XResolve | 0.823529 | 0.033816 | 14 | 3 |
| 2 | GraphSurgery | 0.741379 | 0.076512 | 43 | 15 |
| 2 | BResolve | 0.384615 | 0.008897 | 5 | 8 |
| 2 | XResolve | 0.250000 | 0.001779 | 1 | 3 |
| 3 | GraphSurgery | 0.235294 | 0.090909 | 4 | 13 |
| 3 | BResolve | 0.333333 | 0.045455 | 2 | 4 |
| 3 | XResolve | 1.000000 | 0.022727 | 1 | 0 |

## D.4 Appendix: Feasbility of Breaker to recover consistent contigs

In this section, we study why Breaker can recover contigs by modelling the mis-assemblies formed by an upstream assembler

We define the ground truth to be $S_0 = \{s_1, s_2, ..., s_n\}$ which is a set of strings with alphabets taken from $\Sigma = \{A, C, G, T\}$. Now we specify their repeat structures as follows. Let $x, y$ be length $L$ substrings of $s_i, s_j$ respectively , where $i \neq j$ and $L > 2$. If $\forall 1 < k < L, x[k] = y[k]$ and $x[1] \neq y[1], x[L] \neq y[L]$, then we call $(x, y)$ be a maximal exact repeat of length $L - 2$. Although this notion of maximal exact repeat can be generalized to the same

string, for simplicity of discussion, we assume they are extracted from different strings. We fix $K_0$ to be a large constant which is related to the length of the reads and assume that there are only $r$ maximal exact repeats of length $> K_0$.

Next, we model the upstream assembler's mis-assembly formation process by the following sequence of operations of strings. Let $\{T_j\}_{1\leq j\leq m}$ be a sequence of operations that act on strings $S_0$ and form $\{S^{(j)}\}_{1\leq j\leq m}$ successively. That is, $S^{(0)} = S_0$ and $1 \leq j \leq m, S^{(j)} = T_j(S^{(j-1)})$. Now, we specify the action of $T_j$. It picks two arbitrary strings with a maximal repeat of length $> K_0$. Then, it breaks at the start of the repeat and joins the corresponding string at the breakpoint. Symbolically, let $T$ operate on two strings $s = axb, t = cxd$, where the common segment is $x$ and the breakpoint is the position immediately before $x$. The resultant strings are $s' = axd, t' = cxb$. We further assume that each string under the operations does not have repeat within itself of length $> K_0$.

Under this setting, we prove the following theorem.

**Theorem D.4.1.** *Given $S^{(m)}$ generated from $S_0 = \{s_i\}_{1\leq i\leq n}$ after successive operations by $\{T_j\}_{1\leq j\leq m}$ , we can recover a set of strings $W$ of cardinality at most $n + 4r$ such that $W$ is consistent with $S_0$ (i.e. for each string $w \in W$, $w$ is a substring of some string $s \in S_0$).*

*Proof.* The way to construct the set $W$ is as follows. We first identify all maximal exact repeats across the strings in $S^{(m)}$. We then break the strings at every endpoints of each of these maximal exact repeats. Now, it remains to show that 1) there are at most $n + 4r$ strings in $W$ and 2) they are consistent with the ground truth.

To show them, we use the following bookkeeping method. Let us assign a unique label to each position at each string in the ground truth $S_0$. Let the set of all the labels be $B$ and the mapping from $B$ to string index and offset be $f_0$. At the beginning, we define $\Phi_0$ as the labels that are the endpoints of any maximal exact repeat of length $> K_0$. That is, $\Phi_0 = \{a \in B \mid a$ corresponds to an endpoint of some maximal exact repeat of of length $> K_0$ in $S^{(0)}\}$. When we apply $T_j$ on the strings, let $x$ be the repeat. We move both the segment and the associated labels to the other string starting at the left endpoint of $x$. The exceptions are the labels within the repeat $x$ which are associated with some right endpoints of another repeat $x'$ that has left endpoint before $x$. We keep those labels at the original positions. Since the set of labels remains invariant, and they correspond to a bijection, $f_j$, from $B$ to string position at each stage after $T_j$, we can define $\Phi_j = \{a \in B \mid a$ corresponds to an endpoint of some maximal exact repeat of of length $> K_0$ in $S^{(j)}\}$

We consider the simple case when initially no two pairs of repeat copies overlap at exactly one point(otherwise, we just need to generalize our book keeping scheme by introducing multiple labels at those points). In that case, it turns out that $\Phi_j$ is invariant(i.e. $\Phi_j = \Phi_0$ for all $j$), which we will prove in a separate Lemma. With this Lemma, then we can show the theorem follows.

We first show that $W$ is consistent with $S_0$. We note that for each $T_j$, if we mark the label of the junction as $b_j$ and break them, then the resulting set of string will be consistent throughout. But since $b_j \in \Phi_j$ and $\bigcup_j \{b_j\} \subset \bigcup_j \Phi_j = \Phi_m = \Phi_0$, it suffices to

break at every position corresponding to $\Phi_m$ in $S^{(m)}$ to obtain consistent strings. Moreover, $|\Phi_m| = |\Phi_0| \leq 4r$. So, if we break at every position corresponding to $\Phi_m$ in $S^{(m)}$, we have at most $n + 4r$ resultant strings. This gives, $|W| \leq n + 4r$.

□

**Lemma D.4.2.** *If* $0 \leq j \leq m$, *we have* $\Phi_j = \Phi_0$.

*Proof.* We consider $j = 1$ and inductively, the lemma follows. Without loss of generality, we assume $s_1, s_2$ are the strings that $T_1$ acts on and the associated repeat is $x$.

If $T_1$ can cause an element $b \in B$ to enter or leave $\Phi_1$, it could only belong to a maximal repeat that includes a copy of $x$. Otherwise the labels and the moving segment, which include that potential repeat segment, are moved together. Thus, there cannot be any creation/destruction of maximal exact repeats. We will show that, even for those repeats that include a copy of $x$, their endpoints are still invariant. Without loss of generality, we take the suspicious repeat to end at the right endpoint of $s_1$. There are two cases that can cause changes in $\Phi_1$ upon $T_1$. These include getting a bigger maximal repeat or getting a big repeat separated into smaller pieces with a third string. Since we assume that we cannot have a repeat of length $> K_0$ on the same string in the sequence of operations, the third string cannot be $s_1$ or $s_2$. They correspond to a $T_1$ that goes either from left to right or right to left in Fig D.1. We enumerate the pairwise maximal repeats as shown in Fig D.1. It turns out that in both cases, the set of associated repeat endpoints is invariant. This concludes the proof that $\Phi_1 = \Phi_0$ □

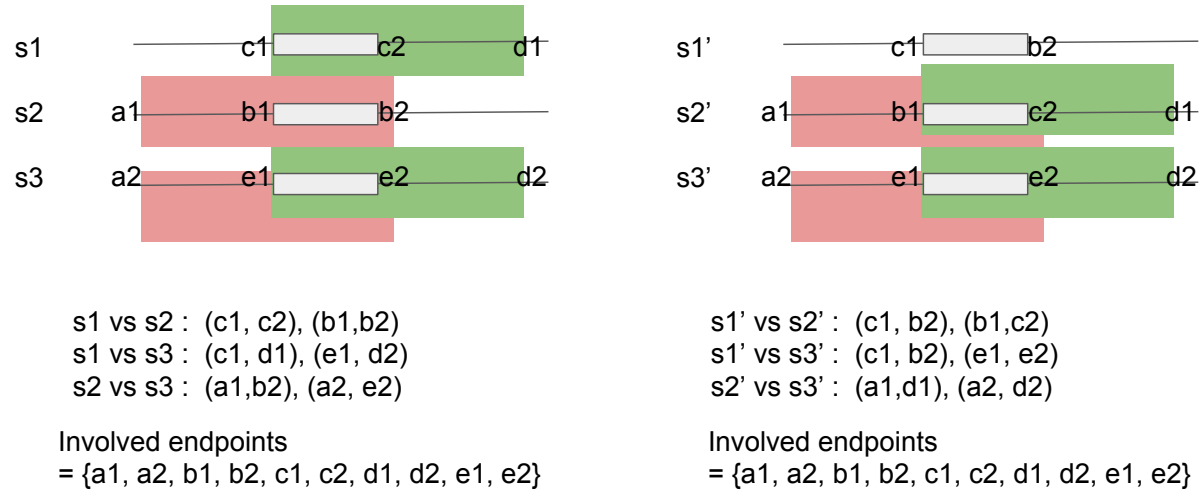



Figure D.1: Illustration of conservation of endpoints

## D.5 Appendix: More information on the EM algorithm and the MSA

In this section, we discuss about the details of the EM algorithm used and related materials.

### Derivation of the EM algorithm

$$
\begin{aligned}
&\log P_\theta(X, Z) \\
&= \log \Pi_{1\le i\le n} P_\theta(R_i, Z_i) \\
&= \sum_{1\le i\le n} \log P_\theta(R_i, Z_i) \\
&= \sum_{1\le i\le n} \log \Pi_{1\le j\le k} (\lambda_j P(R_i \mid Z_i = j))^{1_{Z_i=j}} \\
&= \sum_{1\le i\le n} \sum_{1\le j\le k} 1_{Z_i=j} [\log \lambda_j - \log \ell_j + \log(q^{d(R_i,I_j)}(1-2q)^{L-d(R_i,I_j)})] \\
&= \sum_{1\le i\le n} \sum_{1\le j\le k} 1_{Z_i=j} [\log \lambda_j - \log \ell_j + d(R_i, I_j) \log \frac{q}{1-2q} + L \log(1-2q)]
\end{aligned}
$$

Thus, after taking expectation, we get $E_{q(z|x,\theta^t)}[\ell(x, Z, \theta^{t+1})]$ as desired.

### Feasibility of MSA in our setting

Note that when we only have substitution noise and if all the $R_i$ originates from the same genomic location, the problem of $min_x \sum d(x, R_i)$ can be readily solved by a majority vote. We expect similar results regarding indel noise. However, we need to pre-process with an alignment phase before the majority vote. We thus introduce Algorithm majority-consensus-star-alignment.

1. Compute alignment of $R_1$ and $R_j$ where $j \ge 2$
2. for j = 2 to n, use the alignment of $R_1$ and $R_j$ to form introduce gaps to previous alignment with the principle of "once a gap always a gap"
3. Take column-wise majority to form $x^*$
4. return $x^*$

We note that in the alignment, we use the scoring scheme of (1, -1 ,-1 ,-10) for match, insertion, deletion, substitution. It is because pure substitution noise is rare in current long read technology. We also note that when there is a run of alphabet, we will push the gap towards the end of the alignment. For example CCAAATT is aligned to CCAA_TT.

**Theorem D.5.1.** *Let $\{R_i\}_{1\leq i\leq n}$ be a set of string with alphabets in $\{A, C, G, T\}$ of length $\{\ell(R_i)\}_{1\leq i\leq n}$ where $\ell(R_i) > n > 5$. If $\forall i \neq j, d(R_i, R_j) = 2$ and $\exists x^*$ such that $\forall i, d(x^*, R_i) = 1$ then the majority-consensus-star-alignment can find the optimizer of $\min_x \sum d(x, R_i)$.*

*Proof.* We can break it down into the following three steps. A high level intuition is that we are randomly placing an error on $R_i$ generated from the same source, so, a simple majority vote should just work after doing an initial alignment.

1. Note that $x^*$ is the optimizer. If we define $R_{n+1} = R_1$, we have, $\forall x, \sum_{1\leq i\leq n} d(x, R_i) = \frac{1}{2}\sum_{1\leq i\leq n}[d(x, R_i)+d(x, R_{i+1})] \geq \frac{1}{2}\sum_{1\leq i\leq n} d(R_i, R_{i+1}) = n$ But since $\sum_{1\leq i\leq n} d(x^*, R_i) = \sum_{1\leq i\leq n} 1 = n$ , we know that $x^*$ is the optimizer.

2. Second, we assume we input the ground truth $x^*$ as a read, we will find that the algorithm give $x^*$ as the output.

   The reason is as follows. Let $e_i$ be the edit introduced by $R_i$ when aligned to $x*$. Note that $e_i \neq e_j$ if $i \neq j$ otherwise, $d(R_i, R_j) = 0$. So, it means that $e_i$ cannot win the majority vote at the end because $n \geq 6$ and $|\{A, C, G, T, -\}| = 5$, so entry at $x*$ will be voted instead.

3. Finally, we find that the alignment with $x^*$ is the same as that without it as input.

   The reason is as follows. We have the notation of $M(A, B)$ as the alignment of A and B when $x^*$ is the first input, and $M_S(A, B)$ as the alignment of A and B when $x^*$ is the input. We claim that a small lemma, which says that $\forall j, M(R_1, R_j) = M_S(R_1, R_j)$. Note that it suffices because no gaps are introduced without conflicting some $R_i$. Then with the lemma, we have alignment of every reads be identical with and without $x^*$, and by step 1 and 2, we know that the algorithm will output the right optimizer. Now we proceed to show the lemma. First note $e_i$ corresponds to edit on $x*$ for $R_i$. Recall that, $e_i$ has to be distinct due to $d(R_i, R_j) = 2$. Now consider , without loss of generality, $e_1, e_2$ and their corresponding location when $x^*$ is the input. We define runs of alphabets that $e_i$ lands on under $M_S$ as $r_i$. Now, we exhaust the cases on $r_i$.

   a) There exists at least one other run between $r_1$ and $r_2$. For example, AAAA-CCCTTT vs AAA-CCCTT- Since putting $e_1, e_2$ on $M_S$ gives two edits between $r_1, r_2$, we cannot shift the alphabets at the middle to give the same edit distance. This means that the same alignment shows up under $M$ so as to conserve the same edit distance. Moreover, as the - is always put to the end of run, we will have that consistent under both $M_S$ and $M$ too.

   b) $r_1, r_2$ are neighboring runs. For example, CCC-TTT vs CCCCTT-. Shifting of run at $r_1, r_2$ will cause substitution error, so it is not used under $M_S$. Thus, $r_1, r_2$ will have the same alignment too under $M$ to conserve the edit distance of 2.

   c) $r_1, r_2$ are on the same run. For example, CCCC– vs CCCCC while $x^*$gives CCCC- Since the - is always put at the end of the run, we have the alignment conserved under $M$ and $M_S$.

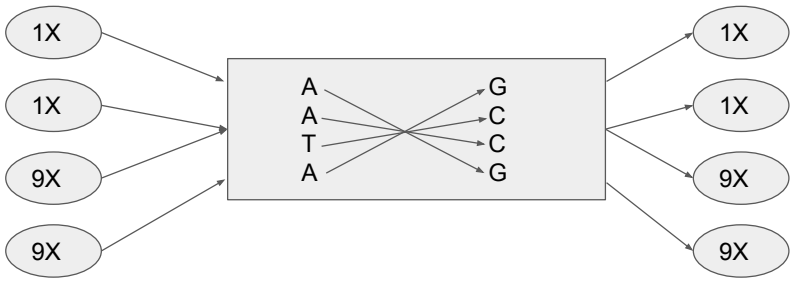
□

We note that, in our implementation of BIGMAC, we use ClustalW2[33] to do the core of multiple sequence alignment. We first use MUMmer to get a rough anchors of the reads and then we chop up the reads into smaller Kmers. Then, we group the related Kmers together use ClustalW2 to do the multiple sequence alignment.

## An interesting repeat

There is an interesting case which can justify why we need the EM algorithm for some tough cases. Consider the situation in Fig D.2. The correct matching is the one that follows row by row. However, there exists matching at the interior such that the polymorphic sites are still consistent (as shown in the figure). Moreover, if we only consider abundance information alone, this repeat cannot be resolved as well (in the sense that we cannot find the correct matching). However, if we consider both the abundances and the polymorphism together during the decision making, we can identify the correct linkage. That is why we introduce the parameter formulation to incorporate both of these quantities.

Figure D.2: An example regarding why it requires abundances and edit distance should be considered together



## D.6 Appendix: Commands for datasets

Commands for using BIGMAC on synthetic data and real data are all based on the following commands.

```
$ python –m srcRefactor.misassemblyFixerLib.mFixer destF mPath
$ python –m srcRefactor.repeatPhaserLib.aSplitter destF mPath
```

FinisherSC, SSPACE_LongRead and PBJelly are run at their default settings.
In particular, the commands used to run them are as follows.

```
FinisherSC :
$ python finisherSC.py dest mPath

PBJelly :
$ Jelly.py setup Protocol.xml
$ Jelly.py mapping Protocol.xml
$ Jelly.py support Protocol.xml
$ Jelly.py extraction Protocol.xml
$ Jelly.py assembly Protocol.xml
$ Jelly.py output Protocol.xml

SSPACE_LongRead :
$  perl SSPACE–LongRead.pl –t 20 –c LC.fasta –p LR.fasta –b e2e/
```

The protocol.xml has the following setting for BLASR, <blasr>-minMatch 8 -minPctIdentity 70 -bestn 1 -nCandidates 20 -maxScore -500 -nproc 20 -noSplitSubreads</blasr>

Moreover, we note that you can reproduce results regarding BIGMAC by running python reproduce.py to download data, dependencies and run the tools. The results is saved in allinone.txt

## D.7 Appendix: Detailed Quast reports

The Quast reports for various comparison for synthetic data and dataset 1,2,3 are in the following tables.

Table D.3: Synthetic data (Comparison with Breaker only and HGAP results). All statistics are based on contigs of size ≥ 500 bp, unless otherwise noted (e.g., "# contigs (≥ 0 bp)" and "Total length (≥ 0 bp)" include all contigs).

| Assembly | Original | Breaker only | BIGMAC end-to-end |
|---|---|---|---|
| # contigs (≥ 0 bp) | **2** | 4 | **2** |
| # contigs (≥ 1000 bp) | **2** | 4 | **2** |
| Total length (≥ 0 bp) | **1**0000000 | **1**0000000 | 9999992 |
| Total length (≥ 1000 bp) | **1**0000000 | **1**0000000 | 9999992 |
| # contigs | **2** | 4 | **2** |
| Largest contig | **5**000000 | 2512000 | 4999998 |
| Total length | **1**0000000 | **1**0000000 | 9999992 |
| Reference length | 10000000 | 10000000 | 10000000 |
| GC (%) | 50.01 | 50.01 | 50.01 |
| Reference GC (%) | 50.01 | 50.01 | 50.01 |
| N50 | **5**000000 | 2512000 | 4999998 |
| NG50 | **5**000000 | 2512000 | 4999994 |
| N75 | **5**000000 | 2488000 | 4999994 |
| NG75 | **5**000000 | 2488000 | 4999994 |
| L50 | **1** | 2 | **1** |
| LG50 | **1** | 2 | 2 |
| L75 | **2** | 3 | **2** |
| LG75 | **2** | 3 | **2** |
| # misassemblies | 2 | **0** | **0** |
| # misassembled contigs | 2 | **0** | **0** |
| Misassembled contigs length | 10000000 | **0** | **0** |
| # local misassemblies | 0 | 0 | 0 |
| # unaligned contigs | 0 + 0 part | 0 + 0 part | 0 + 0 part |
| Unaligned length | 0 | 0 | 0 |
| Genome fraction (%) | 100.000 | 100.000 | 100.000 |
| Duplication ratio | 1.000 | 1.000 | 1.000 |
| # N's per 100 kbp | 0.00 | 0.00 | 0.00 |
| # mismatches per 100 kbp | **0**.00 | **0**.00 | 0.05 |
| # indels per 100 kbp | **0**.00 | **0**.00 | 2.38 |
| Largest alignment | 2512000 | 2512000 | **4**999998 |
| NA50 | 2512000 | 2512000 | **4**999998 |
| NGA50 | 2512000 | 2512000 | **4**999994 |
| NA75 | 2488000 | 2488000 | **4**999994 |
| NGA75 | 2488000 | 2488000 | **4**999994 |
| LA50 | 2 | 2 | **1** |
| LGA50 | 2 | 2 | 2 |
| LA75 | 3 | 3 | **2** |
| LGA75 | 3 | 3 | **2** |

Table D.4: Dataset 1 (Comparison with Breaker only and HGAP results): All statistics are based on contigs of size ≥ 500 bp, unless otherwise noted (e.g., "# contigs (≥ 0 bp)" and "Total length (≥ 0 bp)" include all contigs).

| Assembly | Original | Breaker only | BIGMAC end-to-end |
|---|---|---|---|
| # contigs (≥ 0 bp) | **1**30 | 199 | 131 |
| # contigs (≥ 1000 bp) | 130 | 197 | **1**29 |
| Total length (≥ 0 bp) | **3**0499818 | 29452892 | 29273543 |
| Total length (≥ 1000 bp) | **3**0499818 | 29452752 | 29273403 |
| # contigs | 130 | 197 | **1**29 |
| Largest contig | **8**887616 | 8615553 | 8615553 |
| Total length | **3**0499818 | 29452752 | 29273403 |
| Reference length | 30128987 | 30128987 | 30128987 |
| GC (%) | 56.54 | 57.45 | 57.68 |
| Reference GC (%) | 56.98 | 56.98 | 56.98 |
| N50 | 818655 | 758280 | **4**352719 |
| NG50 | 1595590 | 567256 | **4**352719 |
| N75 | **2**74801 | 157172 | **2**74801 |
| NG75 | **2**77114 | 132279 | 256020 |
| L50 | 4 | 4 | **3** |
| LG50 | **3** | 5 | **3** |
| L75 | 23 | 28 | **1**4 |
| LG75 | 22 | 32 | **1**6 |
| # misassemblies | 18 | **4** | 7 |
| # misassembled contigs | 15 | **4** | 7 |
| Misassembled contigs length | 16357196 | **5**36534 | 1785642 |
| # local misassemblies | **6** | **6** | 9 |
| # unaligned contigs | 0 + 0 part | 0 + 0 part | 0 + 0 part |
| Unaligned length | 0 | 0 | 0 |
| Genome fraction (%) | **9**8.189 | 96.217 | 96.325 |
| Duplication ratio | 1.033 | 1.016 | **1**.010 |
| # N's per 100 kbp | 0.00 | 0.00 | 0.00 |
| # mismatches per 100 kbp | 33.76 | **2**2.38 | 44.80 |
| # indels per 100 kbp | 7.13 | **5**.40 | 63.44 |
| Largest alignment | **8**631596 | 8615553 | 8615553 |
| NA50 | 758280 | 758280 | **4**351628 |
| NGA50 | 758280 | 567256 | **4**351628 |
| NA75 | 227835 | 148337 | **2**62515 |
| NGA75 | **2**54545 | 132279 | 181075 |
| LA50 | 5 | 4 | **3** |
| LGA50 | 5 | 5 | **3** |
| LA75 | 26 | 29 | **1**4 |
| LGA75 | 25 | 32 | **1**7 |

Table D.5: Dataset 2 (Comparison with Breaker only and HGAP results). All statistics are based on contigs of size ≥ 500 bp, unless otherwise noted (e.g., "# contigs (≥ 0 bp)" and "Total length (≥ 0 bp)" include all contigs).

| Assembly | Original | Breaker only | BIGMAC end-to-end |
|---|---|---|---|
| # contigs (≥ 0 bp) | 477 | 382 | **3**51 |
| # contigs (≥ 1000 bp) | 477 | 371 | **3**41 |
| Total length (≥ 0 bp) | **3**2897488 | 29572416 | 29605579 |
| Total length (≥ 1000 bp) | **3**2897488 | 29569477 | 29603092 |
| # contigs | 477 | 374 | **3**44 |
| Largest contig | 4673711 | 4673711 | 4673711 |
| Total length | **3**2897488 | 29571716 | 29605331 |
| Reference length | 66662626 | 66662626 | 66662626 |
| GC (%) | 47.38 | 48.81 | 48.80 |
| Reference GC (%) | 46.01 | 46.01 | 46.01 |
| N50 | **3**97611 | 354308 | **3**97611 |
| N75 | 38471 | 59190 | **7**5666 |
| L50 | **9** | 13 | 12 |
| L75 | 101 | 70 | **5**7 |
| # misassemblies | 187 | **2**5 | 28 |
| # misassembled contigs | 176 | **2**1 | 22 |
| Misassembled contigs length | 18192123 | **8**079336 | 8582043 |
| # local misassemblies | 22 | **1**8 | 19 |
| # unaligned contigs | 39 + 7 part | 30 + 7 part | **2**9 + 8 part |
| Unaligned length | 1646412 | **9**82915 | 993710 |
| Genome fraction (%) | 41.946 | 41.941 | **4**1.995 |
| Duplication ratio | 1.118 | 1.023 | **1**.022 |
| # N's per 100 kbp | 0.00 | 0.00 | 0.00 |
| # mismatches per 100 kbp | **1**.58 | 1.97 | 8.68 |
| # indels per 100 kbp | **8**.63 | 9.02 | 37.15 |
| Largest alignment | 4547258 | 4547258 | 4547258 |
| NA50 | **3**69454 | 333580 | **3**69454 |
| NA75 | 32926 | 47209 | **5**6711 |
| LA50 | **1**2 | 15 | 14 |
| LA75 | 123 | 81 | **6**8 |

Table D.6: Dataset 3 (Comparison with Breaker only and HGAP results). All statistics are based on contigs of size ≥ 500 bp, unless otherwise noted (e.g., "# contigs (≥ 0 bp)" and "Total length (≥ 0 bp)" include all contigs).

| Assembly | Original | Breaker only | BIGMAC end-to-end |
|---|---|---|---|
| # contigs (≥ 0 bp) | 185 | 154 | **1**45 |
| # contigs (≥ 1000 bp) | 185 | 149 | **1**40 |
| Total length (≥ 0 bp) | **1**7393660 | 13844743 | 13912664 |
| Total length (≥ 1000 bp) | **1**7393660 | 13843875 | 13911796 |
| # contigs | 185 | 149 | **1**40 |
| Largest contig | 3968563 | 3968563 | 3968563 |
| Total length | **1**7393660 | 13843875 | 13911796 |
| Reference length | 7883268 | 7883268 | 7883268 |
| GC (%) | 61.18 | 60.96 | 60.98 |
| Reference GC (%) | 61.71 | 61.71 | 61.71 |
| N50 | 257044 | **3**59704 | **3**59704 |
| NG50 | 3968563 | 3968563 | 3968563 |
| N75 | 82370 | 82649 | **9**9878 |
| NG75 | **3**924590 | 474671 | 517104 |
| L50 | **5** | 7 | 7 |
| LG50 | 1 | 1 | 1 |
| L75 | 38 | 29 | **2**7 |
| LG75 | **2** | 5 | 5 |
| # misassemblies | 26 | **1**1 | 14 |
| # misassembled contigs | 20 | **5** | **5** |
| Misassembled contigs length | 5470082 | **4**234268 | 4328506 |
| # local misassemblies | 2 | 2 | 2 |
| # unaligned contigs | 118 + 0 part | 121 + 1 part | **1**15 + 2 part |
| Unaligned length | 5585886 | **5**543409 | 5553281 |
| Genome fraction (%) | **9**9.983 | 99.982 | 99.982 |
| Duplication ratio | 1.498 | **1**.053 | 1.060 |
| # N's per 100 kbp | 0.00 | 0.00 | 0.00 |
| # mismatches per 100 kbp | **0**.18 | 0.24 | 2.30 |
| # indels per 100 kbp | **8**.70 | 22.88 | 23.60 |
| Largest alignment | **3**924590 | 1719755 | 1719755 |
| NA50 | 137772 | **2**84436 | **2**84436 |
| NGA50 | **1**719755 | 569978 | 576251 |
| NGA75 | **1**452284 | 474671 | 517104 |
| LA50 | 11 | **1**0 | **1**0 |
| LGA50 | **2** | 4 | 4 |
| LGA75 | **3** | 7 | 7 |

Table D.7: Dataset 1 (Comparison with other tools) : All statistics are based on contigs of size ≥ 500 bp, unless otherwise noted (e.g., "# contigs (≥ 0 bp)" and "Total length (≥ 0 bp)" include all contigs).

| Assembly | original | BIGMAC | finisherSC_e2e | jelly_e2e | SSPACE_e2e |
|---|---|---|---|---|---|
| # contigs (≥ 0 bp) | 130 | 131 | **5**3 | 100 | 86 |
| # contigs (≥ 1000 bp) | 130 | 129 | **5**3 | 100 | 86 |
| Total length (≥ 0 bp) | 30499818 | 29273543 | 29883342 | **3**0619263 | 30589751 |
| Total length (≥ 1000 bp) | 30499818 | 29273403 | 29883342 | **3**0619263 | 30589751 |
| # contigs | 130 | 129 | **5**3 | 100 | 86 |
| Largest contig | 8887616 | 8615553 | 8887616 | **8**889022 | 8887616 |
| Total length | 30499818 | 29273403 | 29883342 | **3**0619263 | 30589751 |
| Reference length | 30128987 | 30128987 | 30128987 | 30128987 | 30128987 |
| GC (%) | 56.54 | 57.68 | 57.14 | 56.54 | 56.54 |
| Reference GC (%) | 56.98 | 56.98 | 56.98 | 56.98 | 56.98 |
| N50 | 818655 | 4352719 | 2531294 | 4642330 | **4**657611 |
| NG50 | 1595590 | 4352719 | 2531294 | 4642330 | **4**657611 |
| N75 | 274801 | 274801 | 415024 | 418480 | **4**93683 |
| NG75 | 277114 | 256020 | 399053 | **8**18655 | **8**18655 |
| L50 | 4 | **3** | **3** | **3** | **3** |
| LG50 | 3 | 3 | 3 | 3 | 3 |
| L75 | 23 | 14 | 12 | **6** | **6** |
| LG75 | 22 | 16 | 13 | **5** | **5** |
| # misassemblies | 18 | **7** | 32 | 19 | 32 |
| # misassembled contigs | 15 | **7** | 23 | 16 | 20 |
| Misassembled contigs length | 16357196 | **1**785642 | 20096169 | 21804531 | 17545253 |
| # local misassemblies | **6** | 9 | 11 | 9 | 36 |
| # unaligned contigs | **0** + 0 part | **0** + 0 part | **0** + 0 part | 0 + 11 part | **0** + 0 part |
| Unaligned length | **0** | **0** | **0** | 33217 | **0** |
| Genome fraction (%) | 98.189 | 96.325 | 98.330 | **9**8.423 | 98.189 |
| Duplication ratio | 1.033 | **1**.010 | 1.030 | 1.034 | 1.037 |
| # N's per 100 kbp | **0**.00 | **0**.00 | **0**.00 | **0**.00 | 294.00 |
| # mismatches per 100 kbp | **3**3.76 | 44.80 | 73.10 | 34.06 | 33.96 |
| # indels per 100 kbp | 7.13 | 63.44 | 23.53 | 9.39 | **6**.69 |
| Largest alignment | 8631596 | 8615553 | 8631596 | **8**631646 | 8631596 |
| NA50 | 758280 | **4**351628 | 2530093 | 3871007 | 3854031 |
| NGA50 | 758280 | **4**351628 | 1537643 | 3871007 | 3854031 |
| NA75 | 227835 | 262515 | 304665 | **3**61412 | 361362 |
| NGA75 | 254545 | 181075 | 304665 | **4**14429 | **4**14429 |
| LA50 | 5 | **3** | **3** | **3** | **3** |
| LGA50 | 5 | **3** | 4 | **3** | **3** |
| LA75 | 26 | 14 | 16 | **8** | **8** |
| LGA75 | 25 | 17 | 16 | **7** | **7** |

Table D.8: Dataset 2 (Comparison with other tools) : All statistics are based on contigs of size ≥ 500 bp, unless otherwise noted (e.g., "# contigs (≥ 0 bp)" and "Total length (≥ 0 bp)" include all contigs).

| Assembly | original | BIGMAC | finisherSC_e2e | jelly_e2e | SSPACE_e2e |
|---|---|---|---|---|---|
| # contigs (≥ 0 bp) | 477 | 351 | 447 | 403 | **3**07 |
| # contigs (≥ 1000 bp) | 477 | 341 | 447 | 403 | **3**07 |
| Total length (≥ 0 bp) | 32897488 | 29605579 | 32870423 | **3**4484366 | 33520228 |
| Total length (≥ 1000 bp) | 32897488 | 29603092 | 32870423 | **3**4484366 | 33520228 |
| # contigs | 477 | 344 | 447 | 403 | **3**07 |
| Largest contig | 4673711 | 4673711 | 4673711 | 4673711 | 4673711 |
| Total length | 32897488 | 29605331 | 32870423 | **3**4484366 | 33520228 |
| Reference length | 66662626 | 66662626 | 66662626 | 66662626 | 66662626 |
| GC (%) | 47.38 | 48.80 | 47.40 | 46.90 | 47.38 |
| Reference GC (%) | 46.01 | 46.01 | 46.01 | 46.01 | 46.01 |
| N50 | 397611 | 397611 | 654163 | **1**585584 | 1568442 |
| NG50 | - | - | - | 17013 | 14909 |
| N75 | 38471 | 75666 | 43018 | 61775 | **9**5133 |
| L50 | 9 | 12 | 8 | **6** | 7 |
| LG50 | - | - | - | 329 | 294 |
| L75 | 101 | 57 | 89 | 65 | **4**5 |
| # misassemblies | 187 | **2**8 | 192 | 271 | 255 |
| # misassembled contigs | 176 | **2**2 | 168 | 246 | 165 |
| Misassembled contigs length | 18192123 | **8**582043 | 18393113 | 24250973 | 23415983 |
| # local misassemblies | 22 | **1**9 | 22 | 37 | 101 |
| # unaligned contigs | 39 + 7 part | 29 + 8 part | 34 + 7 part | 38 + 23 part | **17** + 5 part |
| Unaligned length | 1646412 | **9**93710 | 1594170 | 1760782 | 1479235 |
| Genome fraction (%) | 41.946 | 41.995 | 41.999 | **4**3.521 | 41.946 |
| Duplication ratio | 1.118 | **1**.022 | 1.117 | 1.128 | 1.146 |
| # N's per 100 kbp | **0**.00 | **0**.00 | **0**.00 | **0**.00 | 1857.80 |
| # mismatches per 100 kbp | **1**.58 | 8.68 | 4.39 | 15.40 | **1**.58 |
| # indels per 100 kbp | 8.63 | 37.15 | 16.06 | 71.69 | **8**.49 |
| Largest alignment | 4547258 | 4547258 | 4547258 | 4547258 | 4547258 |
| NA50 | 369454 | 369454 | 401563 | **7**42006 | 737193 |
| NA75 | 32926 | **5**6711 | 33995 | 46245 | 42004 |
| LA50 | 12 | 14 | 11 | **9** | **9** |
| LA75 | 123 | **6**8 | 113 | 90 | 82 |

Table D.9: Dataset 3 (Comparison with other tools) : All statistics are based on contigs of size ≥ 500 bp, unless otherwise noted (e.g., "# contigs (≥ 0 bp)" and "Total length (≥ 0 bp)" include all contigs).

| Assembly | original | BIGMAC | finisherSC_e2e | jelly_e2e | SSPACE_e2e |
|---|---|---|---|---|---|
| # contigs (≥ 0 bp) | 185 | 145 | 162 | 133 | **9**7 |
| # contigs (≥ 1000 bp) | 185 | 140 | 162 | 133 | **9**7 |
| Total length (≥ 0 bp) | 17393660 | 13912664 | 17391031 | **1**8003698 | 17738519 |
| Total length (≥ 1000 bp) | 17393660 | 13911796 | 17391031 | **1**8003698 | 17738519 |
| # contigs | 185 | 140 | 162 | 133 | **9**7 |
| Largest contig | 3968563 | 3968563 | 3968563 | 3971059 | **4**319145 |
| Total length | 17393660 | 13911796 | 17391031 | **1**8003698 | 17738519 |
| Reference length | 7883268 | 7883268 | 7883268 | 7883268 | 7883268 |
| GC (%) | 61.18 | 60.98 | 61.19 | 61.19 | 61.18 |
| Reference GC (%) | 61.71 | 61.71 | 61.71 | 61.71 | 61.71 |
| N50 | 257044 | 359704 | 996532 | 1103847 | **1**266912 |
| NG50 | 3968563 | 3968563 | 3968563 | 3971059 | **4**319145 |
| N75 | 82370 | 99878 | 97964 | 128718 | **2**90104 |
| NG75 | 3924590 | 517104 | 3924590 | 3927083 | **3**985906 |
| L50 | 5 | 7 | **3** | **3** | **3** |
| LG50 | 1 | 1 | 1 | 1 | 1 |
| L75 | 38 | 27 | 27 | 19 | **1**0 |
| LG75 | **2** | 5 | **2** | **2** | **2** |
| # misassemblies | 26 | **1**4 | 25 | 27 | 43 |
| # misassembled contigs | 20 | **5** | 17 | 21 | 23 |
| Misassembled contigs length | 5470082 | **4**328506 | 5465644 | 9434182 | 10736561 |
| # local misassemblies | **2** | **2** | **2** | **2** | 5 |
| # unaligned contigs | 118 + 0 part | 115 + 2 part | 99 + 0 part | 66 + 14 part | **5**0 + 0 part |
| Unaligned length | 5585886 | **5**553281 | 5602837 | 6149028 | 5791170 |
| Genome fraction (%) | **9**9.983 | 99.982 | **9**9.983 | **9**9.983 | **9**9.983 |
| Duplication ratio | 1.498 | **1**.060 | 1.496 | 1.504 | 1.516 |
| # N's per 100 kbp | **0**.00 | **0**.00 | **0**.00 | **0**.00 | 1944.13 |
| # mismatches per 100 kbp | 0.18 | 2.30 | **0**.16 | 0.60 | 0.18 |
| # indels per 100 kbp | 8.70 | 23.60 | **5**.14 | 6.39 | 8.70 |
| Largest alignment | 3924590 | 1719755 | 3924590 | **3**925633 | 3924590 |
| NA50 | 137772 | **2**84436 | 152488 | 107893 | 126445 |
| NGA50 | **1**719755 | 576251 | **1**719755 | **1**719755 | **1**719755 |
| NGA75 | 1452284 | 517104 | 1452284 | **1**453076 | 1452284 |
| LA50 | 11 | **1**0 | **1**0 | 13 | 12 |
| LGA50 | **2** | 4 | **2** | **2** | **2** |
| LGA75 | **3** | 7 | **3** | **3** | **3** |

# Bibliography


[1] Anton Bankevich, Sergey Nurk, Dmitry Antipov, Alexey A Gurevich, Mikhail Dvorkin, Alexander S Kulikov, Valery M Lesin, Sergey I Nikolenko, Son Pham, Andrey D Prjibelski, et al. "SPAdes: a new genome assembly algorithm and its applications to single-cell sequencing". In: *Journal of Computational Biology* 19.5 (2012), pp. 455–477.

[2] Zhirong Bao and Sean R Eddy. "Automated de novo identification of repeat sequence families in sequenced genomes". In: *Genome Research* 12.8 (2002), pp. 1269–1276.

[3] Konstantin Berlin, Sergey Koren, Chen-Shan Chin, James P Drake, Jane M Landolin, and Adam M Phillippy. "Assembling large genomes with single-molecule sequencing and locality-sensitive hashing". In: *Nature biotechnology* 33.6 (2015), pp. 623–630.

[4] Avrim Blum, Tao Jiang, Ming Li, John Tromp, and Mihalis Yannakakis. "Linear approximation of shortest superstrings". In: *Proceedings of the twenty-third annual ACM symposium on Theory of computing*. ACM. 1991, pp. 328–336.

[5] Marten Boetzer and Walter Pirovano. "SSPACE-LongRead: scaffolding bacterial draft genomes using long read sequence information". In: *BMC bioinformatics* 15.1 (2014), p. 1.

[6] Keith R Bradnam, Joseph N Fass, Anton Alexandrov, Paul Baranay, Michael Bechner, Inanç Birol, Sébastien Boisvert, Jarrod A Chapman, Guillaume Chapuis, Rayan Chikhi, et al. "Assemblathon 2: evaluating de novo methods of genome assembly in three vertebrate species". In: *GigaScience* 2.1 (2013), pp. 1–31.

[7] Guy Bresler, Ma'ayan Bresler, and David Tse. "Optimal Assembly for High Throughput Shotgun Sequencing". In: *BMC Bioinformatics* (2013).

[8] J. Butler, I. MacCallum, M. Kleber, I.A. Shlyakhter, M.K. Belmonte, E.S. Lander, C. Nusbaum, and D.B. Jaffe. "ALLPATHS: De novo assembly of whole-genome shotgun microreads". In: *Genome research* 18.5 (2008), pp. 810–820.

[9] Mark J Chaisson and Glenn Tesler. "Mapping single molecule sequencing reads using basic local alignment with successive refinement (BLASR): application and theory". In: *BMC bioinformatics* 13.1 (2012), p. 238.

[10] Mark JP Chaisson, John Huddleston, Megan Y Dennis, Peter H Sudmant, Maika Malig, Fereydoun Hormozdiari, Francesca Antonacci, Urvashi Surti, Richard Sandstrom, Matthew Boitano, et al. "Resolving the complexity of the human genome using single-molecule sequencing". In: *Nature* 517.7536 (2015), pp. 608–611.

[11] Kevin Chen and Lior Pachter. "Bioinformatics for whole-genome shotgun sequencing of microbial communities". In: *PLoS Comput Biol* 1.2 (2005), pp. 106–112.

[12] Chen-Shan Chin, David H Alexander, Patrick Marks, Aaron A Klammer, James Drake, Cheryl Heiner, Alicia Clum, Alex Copeland, John Huddleston, Evan E Eichler, et al. "Nonhybrid, finished microbial genome assemblies from long-read SMRT sequencing data". In: *Nature methods* 10.6 (2013), pp. 563–569.

[13] P.E.C. Compeau, P.A. Pevzner, and G. Tesler. "How to apply de Bruijn graphs to genome assembly". In: *Nature biotechnology* 29.11 (2011), pp. 987–991.

[14] John Eid, Adrian Fehr, Jeremy Gray, Khai Luong, John Lyle, Geoff Otto, Paul Peluso, David Rank, Primo Baybayan, Brad Bettman, et al. "Real-time DNA sequencing from single polymerase molecules". In: *Science* 323.5910 (2009), pp. 133–138.

[15] Isaac Elias. "Settling the intractability of multiple alignment". In: *Journal of Computational Biology* 13.7 (2006), pp. 1323–1339.

[16] Adam C English, Stephen Richards, Yi Han, Min Wang, Vanesa Vee, Jiaxin Qu, Xiang Qin, Donna M Muzny, Jeffrey G Reid, Kim C Worley, et al. "Mind the gap: upgrading genomes with Pacific Biosciences RS long-read sequencing technology". In: *PloS one* 7.11 (2012), e47768.

[17] Sante Gnerre, Iain MacCallum, Dariusz Przybylski, Filipe J Ribeiro, Joshua N Burton, Bruce J Walker, Ted Sharpe, Giles Hall, Terrance P Shea, Sean Sykes, et al. "High-quality draft assemblies of mammalian genomes from massively parallel sequence data". In: *Proceedings of the National Academy of Sciences* 108.4 (2011), pp. 1513–1518.

[18] David Gordon, Chris Abajian, and Phil Green. "Consed: a graphical tool for sequence finishing". In: *Genome research* 8.3 (1998), pp. 195–202.

[19] Alexey Gurevich, Vladislav Saveliev, Nikolay Vyahhi, and Glenn Tesler. "QUAST: quality assessment tool for genome assemblies". In: *Bioinformatics* 29.8 (2013), pp. 1072–1075.

[20] Richard J. Hall, Chen-Shan Chin, Sudeep Mehrotra, Nikoleta Juretic, Jessica Wasserscheid, and Ken Dewar. *An interactive workflow for the analysis of contigs from the metagenomic shotgun assembly of SMRT Sequencing data*. `$http://files.pacb.com/pdf/RHall\_ASM2014\_InteractiveWorkflow.pdf$`. 2014.

[21] Miten Jain, Ian T Fiddes, Karen H Miga, Hugh E Olsen, Benedict Paten, and Mark Akeson. "Improved data analysis for the MinION nanopore sequencer". In: *Nature methods* 12.4 (2015), pp. 351–356.

[22] Haim Kaplan, Moshe Lewenstein, Nira Shafrir, and Maxim Sviridenko. "Approximation algorithms for asymmetric TSP by decomposing directed regular multigraphs". In: *Journal of the ACM (JACM)* 52.4 (2005), pp. 602–626.

[23] Asif Khalak, Ka-Kit Lam, Greg Concepcion, and David Tse. "Conditions on Finishable Read Sets for Automated De Novo Genome Sequencing". In: *Sequencing, Finishing and Analysis in the Future* (May, 2013).

[24] Sergey Koren and Adam M Phillippy. "One chromosome, one contig: complete microbial genomes from long-read sequencing and assembly". In: *Current opinion in microbiology* 23 (2015), pp. 110–120.

[25] Sergey Koren, Michael C Schatz, Brian P Walenz, Jeffrey Martin, Jason T Howard, Ganeshkumar Ganapathy, Zhong Wang, David A Rasko, W Richard McCombie, Erich D Jarvis, et al. "Hybrid error correction and de novo assembly of single-molecule sequencing reads". In: *Nature biotechnology* 30.7 (2012), pp. 693–700.

[26] Stefan Kurtz, Adam Phillippy, Arthur L Delcher, Michael Smoot, Martin Shumway, Corina Antonescu, and Steven L Salzberg. "Versatile and open software for comparing large genomes". In: *Genome biology* 5.2 (2004), R12.

[27] Ka-Kit Lam. "POSTME : POSTprocessing MEtagenomics assembly with hybrid data by highly precise scaffolding". In: (). URL: `https://github.com/kakitone/postme`.

[28] Ka-Kit Lam, Richard Hall, Alicia Clum, and Satish Rao. "BIGMAC : Breaking Inaccurate Genomes and Merging Assembled Contigs for long read metagenomic assembly". In: *bioRxiv* (2016). DOI: `10.1101/045690`. eprint: `http://www.biorxiv.org/content/early/2016/03/29/045690.full.pdf`. URL: `http://www.biorxiv.org/content/early/2016/03/29/045690`.

[29] Ka-Kit Lam, Asif Khalak, and David Tse. "Near-optimal assembly for shotgun sequencing with noisy reads." In: *BMC Bioinformatics* (2014).

[30] Ka-Kit Lam, Kurt LaButti, Asif Khalak, and David Tse. "FinisherSC: a repeat-aware tool for upgrading de novo assembly using long reads". In: *Bioinformatics* 31.19 (2015), pp. 3207–3209. DOI: `10.1093/bioinformatics/btv280`.

[31] Ka-Kit Lam and Nihar B Shah. "Towards Computation, Space, and Data Efficiency in de novo DNA Assembly: A Novel Algorithmic Framework". In: ().

[32] Eric S Lander and Michael S Waterman. "Genomic mapping by fingerprinting random clones: a mathematical analysis". In: *Genomics* 2.3 (1988), pp. 231–239.

[33] Mark A Larkin, Gordon Blackshields, NP Brown, R Chenna, Paul A McGettigan, Hamish McWilliam, Franck Valentin, Iain M Wallace, Andreas Wilm, Rodrigo Lopez, et al. "Clustal W and Clustal X version 2.0". In: *Bioinformatics* 23.21 (2007), pp. 2947–2948.

[34] Jonathan Laserson, Vladimir Jojic, and Daphne Koller. "Genovo: de novo assembly for metagenomes". In: *Journal of Computational Biology* 18.3 (2011), pp. 429–443.

[35] Elaine Mardis, John McPherson, Robert Martienssen, Richard K Wilson, and W Richard McCombie. "What is finished, and why does it matter". In: *Genome research* 12.5 (2002), pp. 669–671.

[36] Duccio Medini, Davide Serruto, Julian Parkhill, David A Relman, Claudio Donati, Richard Moxon, Stanley Falkow, and Rino Rappuoli. "Microbiology in the post-genomic era". In: *Nature Reviews Microbiology* 6.6 (2008), pp. 419–430.

[37] A Motahari, G Bresler, and D Tse. "Information Theory of DNA Sequencing". In: *Information Theory Proceedings (ISIT) 2012 IEEE International Symposium on* (2012).

[38] A. Motahari, G. Bresler, and D. Tse. "Information Theory of DNA Sequencing". In: *arXiv:1203.6233* (2012).

[39] Abolfazl Motahari, Kannan Ramchandran, David Tse, and Nan Ma. "Optimal DNA shotgun sequencing: Noisy reads are as good as noiseless reads". In: *Proceedings of the 2013 IEEE International Symposium on Information Theory, Istanbul, Turkey, July 7-12, 2013* (2013).

[40] David J Munroe and Timothy JR Harris. "Third-generation sequencing fireworks at Marco Island". In: *Nature biotechnology* 28.5 (2010), pp. 426–428.

[41] Eugene W Myers. "The fragment assembly string graph". In: *Bioinformatics* 21.suppl 2 (2005), pp. ii79–ii85.

[42] Eugene W Myers, Granger G Sutton, Art L Delcher, Ian M Dew, Dan P Fasulo, Michael J Flanigan, Saul A Kravitz, Clark M Mobarry, Knut HJ Reinert, Karin A Remington, et al. "A Whole-Genome Assembly of Drosophila". In: (2000).

[43] Gene Myers. "Efficient local alignment discovery amongst noisy long reads". In: *Algorithms in Bioinformatics*. Springer, 2014, pp. 52–67.

[44] Toshiaki Namiki, Tsuyoshi Hachiya, Hideaki Tanaka, and Yasubumi Sakakibara. "MetaVelvet: an extension of Velvet assembler to de novo metagenome assembly from short sequence reads". In: *Nucleic acids research* 40.20 (2012), e155–e155.

[45] Giuseppe Narzisi and Bud Mishra. "Comparing De Novo Genome Assembly: The Long and Short of It". In: *PLoS ONE* 6.4 (Apr. 2011), e19175. DOI: 10.1371/journal.pone.0019175. URL: http://dx.doi.org/10.1371/journal.pone.0019175.

[46] PacBio. *PacBio Devnet*. $https://github.com/PacificBiosciences/DevNet/wiki/Human\_Microbiome\_Project\_MockB\_Shotgun$.

[47] Yu Peng, Henry CM Leung, Siu-Ming Yiu, and Francis YL Chin. "IDBA–a practical iterative de Bruijn graph de novo assembler". In: *Research in Computational Molecular Biology*. Springer. 2010, pp. 426–440.

[48] Yu Peng, Henry CM Leung, Siu-Ming Yiu, and Francis YL Chin. "Meta-IDBA: a de Novo assembler for metagenomic data". In: *Bioinformatics* 27.13 (2011), pp. i94–i101.

[49] Pavel A Pevzner. "1-Tuple DNA sequencing: computer analysis". In: *Journal of Biomolecular structure and dynamics* 7.1 (1989), pp. 63–73.

[50] Pavel A Pevzner, Haixu Tang, and Michael S Waterman. "An Eulerian path approach to DNA fragment assembly". In: *Proceedings of the National Academy of Sciences* 98.17 (2001), pp. 9748–9753.

[51] Mihai Pop. "Genome assembly reborn: recent computational challenges". In: *Briefings in bioinformatics* 10.4 (2009), pp. 354–366.

[52] J Przyborowski and H Wilenski. "Homogeneity of results in testing samples from Poisson series: With an application to testing clover seed for dodder". In: *Biometrika* (1940), pp. 313–323.

[53] N. Rodriguez-Ezpeleta, M. Hackenberg, and A.M. Aransay. *Bioinformatics for high throughput sequencing*. Springer, 2011.

[54] Steven L Salzberg, Adam M Phillippy, Aleksey Zimin, Daniela Puiu, Tanja Magoc, Sergey Koren, Todd J Treangen, Michael C Schatz, Arthur L Delcher, Michael Roberts, et al. "GAGE: A critical evaluation of genome assemblies and assembly algorithms". In: *Genome research* 22.3 (2012), pp. 557–567.

[55] Peter Sanders. "Algorithm engineering–an attempt at a definition". In: *Efficient Algorithms*. Springer, 2009, pp. 321–340.

[56] DNA SEQUENCING. "A plan to capture human diversity in 1000 genomes". In: *Science* 21 (2007), p. 1842.

[57] Claude E. Shannon. "A Mathematical Theory of Communication". In: *The Bell System Technical Journal* 27 (July 1948), pp. 379–423, 623–656. URL: `http://cm.bell-labs.com/cm/ms/what/shannonday/shannon1948.pdf`.

[58] Jared T Simpson and Richard Durbin. "Efficient de novo assembly of large genomes using compressed data structures". In: *Genome Research* 22.3 (2012), pp. 549–556.

[59] *The Critical Assessment of Metagenome Interpretation (CAMI) competition*. `http://blogs.nature.com/methagora/2014/06/the-critical-assessment-of-metagenome-interpretation-cami-competition.html`.

[60] Peter J Turnbaugh, Ruth E Ley, Micah Hamady, Claire M Fraser-Liggett, Rob Knight, and Jeffrey I Gordon. "The human microbiome project". In: *Nature* 449.7164 (2007), pp. 804–810.

[61] Esko Ukkonen. "Approximate string-matching with q-grams and maximal matches". In: *Theoretical computer science* 92.1 (1992), pp. 191–211.

[62] K.A. Wetterstrand. "DNA sequencing costs: data from the NHGRI large-scale genome sequencing program". In: *Accessed November* 20 (2011), p. 2011.

[63] Daniel R Zerbino and Ewan Birney. "Velvet: algorithms for de novo short read assembly using de Bruijn graphs". In: *Genome research* 18.5 (2008), pp. 821–829.