TECHNIQUES TO IMPROVE GENOME ASSEMBLY QUALITY

A Dissertation
Presented to
The Academic Faculty

By

Rahul Nihalani

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in
Computational Science and Engineering

Georgia Institute of Technology

May 2019

TECHNIQUES TO IMPROVE GENOME ASSEMBLY QUALITY

Approved by:

Dr. Srinivas Aluru, Advisor
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Richard Vuduc
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Ümit Çatalyürek
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. May Wang
Department of Biomedical Engineering
Georgia Institute of Technology

Dr. King Jordan
School of Biology
Georgia Institute of Technology

Date Approved: March 25, 2019

It gets worse before it gets better

Anonymous

Dedicated to my beloved uncle
Pratap Rai
1953 - 2018

ACKNOWLEDGEMENTS

I would like to start by acknowledging my parents for their unconditional support towards me throughout this journey. I consider myself fortunate to have such loving parents and I deeply value the many sacrifices they have made to enable me to do well. I am also immensely grateful to my advisor, Dr. Srinivas Aluru for yielding a strong support and having faith in my abilities. Like many other PhD paths, mine has had its share of discouraging times, and I am thankful to him for the motivation and support he has provided during such times. I thank my labmates for providing a supportive and productive environment that has been very helpful in conducting my research. Lastly, I would like to thank my wife. I met her during the later part of my PhD. It must have been incredibly difficult for her to share my hardships, but she has remained a steady source of encouragement for me and has kept patience with me. I owe a lot of gratitude to her.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

De-novo genome assembly is an important problem in the field of genomics. Discovering and analyzing genomes of different species has numerous applications. For humans, it can lead to early detection of disease traits and timely prevention of diseases like cancer. In addition, it is useful in discovering genomes of unknown species. Even though it has received enormous attention in the last couple of decades, the problem remains unsolved to a satisfactory level, as shown in various scientific studies [1, 2, 3]. Paired-end sequencing is a technology that sequences pairs of short strands from a genome, called *reads*. The pairs of reads originate from nearby genomic locations, and are commonly used to help more accurately determine the genomic location of individual reads and resolve repeats in genome assembly. In this thesis, we describe the genome assembly problem, and the key challenges involved in solving it. We discuss related work where we describe the two most popular models to approach the problem: *de-Bruijn graphs* and *overlap graphs*, along with their pros and cons. We describe our proposed techniques to improve the quality of genome assembly. Our main contribution in this work is designing a de-Bruijn graph based assembly algorithm to effectively utilize paired reads to improve genome assembly quality. We also discuss how our algorithm tackles some of the key challenges involved in genome assembly. We adapt this algorithm to design a parallel strategy to obtain high quality assembly for large datasets such as rice within reasonable time-frame. In addition, we describe our work on probabilistically estimating overlap graphs for large short reads datasets. We discuss the results obtained for our work, and conclude with some future work.

CHAPTER 1
INTRODUCTION

Genome of an organism is an important entity that facilitates in governing its entire functionality and contains information required to maintain the organism. The genome comprises of all the genes, as well as other regions called non-coding regions. Given a set of short DNA fragments derived from the genome of an organism, called *reads*, the goal of *genome assembly* is to assemble the given set of reads and construct the original genome. Reads are short pieces of the genome, and overlap between the reads can be used to construct the genome back. Genome assembly can be thought of as solving a jigsaw puzzle from pieces.

Two categories of approaches currently exist to perform genome assembly:

- *Reference based genome assembly*: This method uses a template reference genome of an organism from a species to assemble the genome of a different organism from the same species. Genomes of organisms within a species are highly similar to each other (99.9% similarity in case of humans), and reads obtained from an organism can be mapped to the genome of an organism from the same species to conduct the assembly.

- *De-novo genome assembly*: In the absence of a template genome, reads are assembled using overlaps between different reads. De-novo assembly is particularly useful in discovering genomes of new species. This problem is considered much harder than reference based assembly. The main focus of our work is on de-novo genome assembly, and therefore in the following text, we use "genome assembly" to mean de-novo genome assembly (unless stated otherwise).

The genome assembly problem is known to be NP-hard [4], which means that no

provably efficient algorithm can be expected for solving it. In the following section, we provide the motivation for solving this problem and discuss the work that currently exists related to it.

1.1 Motivation

The importance of genome assembly in bioinformatics has been established more than three decades ago [5]. There is huge interest in discovering genomes of different organisms and study them. Applications include:

- *Comparitive genomics:* Genomes of various organisms can be compared and classified taxonomically to discover closely related species and study evolutionary changes among species. Discovery of genes that are common between humans and other species can be useful in studying properties of those genes via studying their behavior in other species.

- *Detection of diseases:* Information about diseases such as diabetes, cancer is encoded in genes in the form of markers and mutations at specific locations. High quality genome assembly can help early detection and prevention of such diseases. Personalized medicine and treatments based on specific mutations can be assigned to treat patients.

- *Ancestral studies:* Genomes of a human population can be studied to identify their ancestral tree. This can be used for studies such as migration and diversity of population in a given region.

Given such a wide interest, a number of assemblers have been developed in the last decade to solve the assembly problem: ABySS [6], Velvet [7], AllPaths [8], Newbler [9], to name a few. We will discuss these assemblers in detail in chapter 3.

In the following sections, we describe the genome assembly problem in detail and discuss some of the key challenges in solving it.

1.2 Problem description

A genome can be thought of as a set of large strings (each string representing a *chromosome*) over the alphabet $\sum = \{A, C, G, T\}$. These represent four possible nucleotides that a genome comprises of. DNA of an organism is broken into short fragments using a chemical process. These fragments are then sequenced using a sequencing machine (or *sequencer*) to produce reads. The reads then need to be assembled back to infer the genome. Ideally we would like the sequencer to read the genome from start to end as a single string, thereby eliminating the need for the arduous task of assembling the reads back. However, we are currently limited by the sequencer's capabilities that significantly fall short of sequencing the complete genome in one go. We elaborate more on this in section 1.4.

Note that any assembly algorithm will have to use overlaps between sequences and merge them in order to produce longer sequences, which ideally would be the entire genome. It can be inferred therefore that a single copy of DNA cannot give any information on the relative ordering of reads in the genome. At least two copies of DNA must be used to produce the reads to have some overlap information to deduce relative ordering of reads. The number of reads spanning any given position in the genome is known as the *coverage* at that position. Coverage of a read dataset is generally used to imply the average coverage over all positions in the genome. A coverage of at least 2 (or 2X coverage) is needed to have any hope of a good assembly. Figure 1.1 shows reads overlapping with each other, eventually forming a longer sequence.

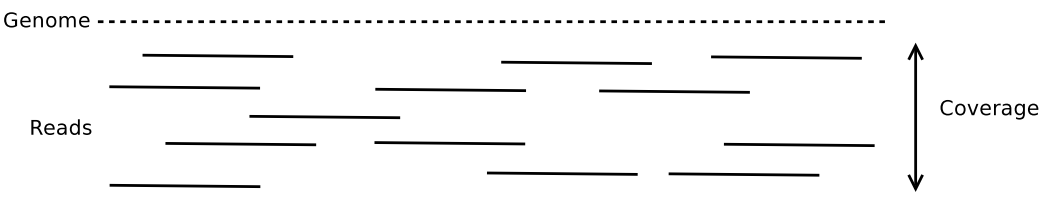

Figure 1.1: Reads mapping to different areas in genome

1.3 Key challenges

In this section, we discuss a few aspects of genome assembly that make the problem challenging.

1.3.1 Double strandedness of the genome

Genomes are *double stranded*, which means that each position along the genome consists of two nucleotides. For this reason, a position in a genome is represented by *basepair* (*bp*). The pairing of nucleotides in a basepair is deterministic. A is always paired with T and G paired with C. In addition, each strand has two directions labeled as $5'$ and $3'$, directed opposite to each other in the two strands. Figure 1.2 shows the two strands of the genome along with directions for each strand.

Consequently, the fragments into which a DNA are broken are also double stranded. Sequencer, while sequencing the fragment, can arbitrarily sequence it using either strand to produce the read. The reading is done in the direction $5'$ to $3'$. Any section of a double stranded genome contains two strands, and one strand is deterministic given the other. Such strands are called *reverse complements* (*complements* in short) of each other. The double stranded characteristics add to the complexity of the assembly problem since it is unknown which strand a given read originates from.

Figure 1.2: Double stranded genome: sequencing takes place from 5' direction to 3'

1.3.2 Multiple copies of genome

Many organisms contain more than one genome in each cell, different genomes usually inherited from different parents. Such organisms are referred to as *polyploid*. Humans, for example, are *diploid*, containing two genomes in each cell, one inherited from each parent. Bread wheat is hexaploid, containing six different genomes its cell.

When sequencing such organisms, the reads can originate from any genome present in the cell. The read dataset thereby consists of reads from multiple genomes without the knowledge of where any given read has come from. This increases the difficulty of performing assembly as in principle, we are tasked with assembling multiple genomes instead of a single genome.

1.3.3 Read lengths and errors

A low read coverage for a read dataset would be considered good if the reads were perfect. That however is not the case. Sequencers occasionally make mistakes in reading DNA fragments which lead to erroneous reads. The error percentage and characteristics depend on the sequencing machine. Sequencers manufactured by companies like Illumina mostly make *substitution* errors, where a base in the genome is substituted by a different base in the read. Figure 1.3 shows an example of substitution error. Here, the upper strand represents the genome and the lower strand represents the sequenced read. It should be noted that the genome is unknown and therefore given an isolated read, it is impossible to determine which position is erroneous (if at all). Other forms of errors are *insertion*/*deletion* errors where a base is inserted/deleted in/from the genome when producing the read.

The read length for Illumina sequencers is typically 150bp to 250bp, with less than 1% error rate. To assemble genomes with lengths of the order of billions of bp, such short reads contain overlaps that are too short in length to obtain a high quality assembly. In addition, such short reads are limited in their use for resolving repeats

(explained further in subsection 1.3.5).

Other companies like Pacific Biosciences (PacBio) and Oxford Nanopore Technologies (ONT) manufacture sequencers that produce reads with mean lengths of 5kbp-20kbp (largest read being as long as 1 million bp), thereby providing larger overlap information. Their error rate is however typically in the range 11% to 15% [10, 11]. Error correction for such reads becomes extremely important for any reliable downstream processing.

Figure 1.3: An example of a substitution error

High coverage data is needed to detect and correct errors in sequences while performing assembly. Producing data with coverage of 40-60X for long genomes, and above 200X for short genomes is typical for current short read sequencers. High coverage gives (at least in principle) a way to correct errors by taking consensus of all the reads aligning to a position, and taking the consensus of that position to be the truth (two truths in case of diploid genome). Most assemblers today have a separate phase to just perform error correction.

1.3.4 Computational scale

Genomes of different species have different lengths and characteristics. For example, a human genome is about 3 billion basepairs long. On the other hand, some plant genomes are generally tens of billions basepairs long.

Consider a read set sequenced from a human genome. Assuming each read to be 150 bp long (typical for Illumina sequencers), to obtain an average read coverage of about 40X, the read set would have to consist of 800 million reads. Any further processing of such large dataset requires efficient parallel algorithms.

As noted earlier, identifying overlaps between reads would form the core of any assembly algorithm. Intuitively, it would help to identify overlaps between all pairs of reads. However, this operation is expensive for the size of datasets under consideration. Specifically, for assembling n reads, any algorithm with $\Omega(n^2)$ or higher computational cost is prohibitive. This poses a key challenge in designing a good genome assembler where efficient algorithms are needed to identify overlaps between select pairs of reads (since all pair comparison is computationally infeasible).

1.3.5 Long repeats

Many genomes consist of a large number of stretches that are nearly identical to sequences present elsewhere in the genome. Such stretches are called *repeats*, shown in figure 1.4. Repeats make the assembly problem harder, since given a read from a repeat region, it is ambiguous as to where this read should belong to in the genome. In figure 1.4, the reads marked in blue and red, though originating from different regions, would be indistinguishable.

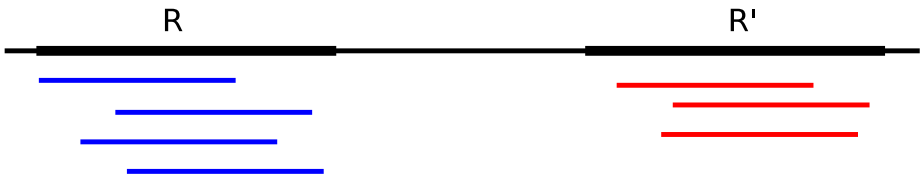

Figure 1.4: Long stretches of repeats in the genome

We will discuss in section 1.4.1 on how *paired reads* are helpful in resolving repeats to a certain extent.

1.3.6 Non-uniform coverage

As mentioned in section 1.2, high coverage data is needed to have any hope of a good assembly. The coverage reported for the sequencing datasets is typically the average coverage over the entire genome. Coverage at specific positions can have huge

variations depending on the genomic region, as shown in figure 1.5. Some regions in the genome are *over-sampled*, contributing large number of reads, while other regions are *under-sampled*, contributing fewer number of reads. The maximum coverage at a position in a genome can be as high as five times the minimum coverage. One reason for such huge variation is that some genomic regions have biological challenges in terms of sequencing them.

Given a dataset with read length l, many assemblers use frequency of k length ($k < l$) substrings of reads (k-mers) observed within the dataset, along with the average coverage, to estimate number of times that k-mer appears in the genome. For uniform coverage, this estimate would be simply the k-mer frequency divided by the coverage. However such large variations in the coverage make this estimation challenging, adding to the difficulty of the problem.

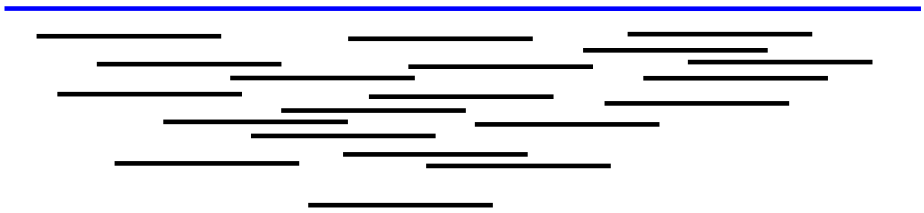

Figure 1.5: Different genomic regions have different coverages

1.4 Sequencing technology

In this section, we take a close look at the sequencing process for producing reads using a DNA sample from an organism. As a first step, the DNA sample is broken down into small DNA fragments. A library, consisting of numerous such DNA fragments is prepared. A sequencer then starts sequencing the fragments to produce substrings of the genome, known as reads. Illumina sequencing technology has a limit on the length of the fragment that it can read continuously, before the error rate goes very high. As a result, irrespective of the fragment length, the sequencer can read the fragment only up to a certain threshold of length. Sequencers produced by companies like Illumina

are capable of performing paired end reading of fragments, which we will describe later.

Until 2005, sequencing was mostly performed using a technology called Sanger sequencing. Sanger reads are typically around 500-1000 bp long. One experiment on a Sanger would typically yield a million reads.

Post 2005, a number of new sequencing technologies emerged with very different read characteristics. They are together referred to as *Next Generation Technology (NGS)* sequencers. The 454 Roche was the first NGS sequencer. Among the most popular NGS sequencers today are Illumina, PacBio and ONT. Throughput of NGS sequencers is orders of magnitude higher than Sanger. One experiment on Illumina, for example, produces about 2 to 6 billion reads in an experiment lasting for about 8 days. This feature enables researchers to produce high coverage data, enhancing the assembly process. This advantage however, comes at a price. The read length for Illumina sequencers is much smaller than Sanger. Illumina typically produces 150 bp reads, at around 1% error rate. Though read lengths for PacBio and ONT are quite long ($>$ 10kb), their error rate is high as well (11-15%). Due to these varying characteristics, different sequencing platforms are more suitable for some application.

1.4.1 Paired end reading

As mentioned earlier, a genome is double stranded. Sequencers from some companies (eg. Illumina) have a way of reading genomic fragments that can provide distance constraints between two reads. The sequencer reads all fragments from a collection, called a *library*. The reading is done in paired manner, where the sequencer reads from both strands simultaneously. As described in subsection 1.3.1, the two strands are read in opposite directions. Figure 1.6 illustrates the reading mechanism. A and B are sequenced starting from opposite ends. Approximate fragment length, known as *insert size*, is known within a range. Depending on the method used to create the

library, this range can be in order of a few hundred bps to a few thousand bps. This produces two reads that are d distance apart in the genome. A significant contribution of this work is to use this information to improve genome assembly quality, and is described in chapter 4.

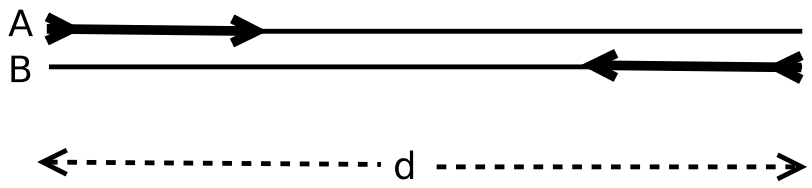

Figure 1.6: Paired end reading

CHAPTER 2
MODELING THE PROBLEM

In this chapter, we describe how the genome assembly problem is modeled as a graph theoretic problem. Input to the assembly problem is the set of reads sequenced with a given coverage. Under ideal circumstances, the output of the assembly should be the set of chromosomes, representing the genome from which the reads were sequenced. However, due to the complex nature of the problem, most assemblers are only able to produce a set of large substrings of the chromosomes, called *contigs*. These contigs are later extended and ordered using paired read information, a stage known as *scaffolding*.

All current assemblers try to solve the problem using one of the two models:

- Overlap Layout Consensus (OLC approach)
- De-Bruijn graph (DB approach)

We discuss each approach in detail.

2.1 Overlap Layout Consensus approach

In this model, a directed graph is constructed using the set of reads as follows:

a. A vertex v_i is constructed for every read r_i in the read set.

b. An edge $(v_i \rightarrow v_j)$ is constructed if r_i and r_j have a good suffix-prefix overlap. Specifically, if $r_i = x.y$ and $r_j = y'.z$ (dot (.) represents the concatanation operator) and x, y, y', z are strings over χ, then $(v_i \rightarrow v_j)$ if:

   - y and y' are sufficiently similar **and**
   - $|y| \geq t$, $|y'| \geq t$, where t is the overlap threshold.

Note that the criteria for introducing an edge is described assuming r_i and r_j originate from the same strand in the genome. Similar check needs to be done assuming the two reads originate from different strands (this is done by creating the *complementary* strand for r_j: $r_j^{'}$ and checking for overlap between r_i and $r_j^{'}$).

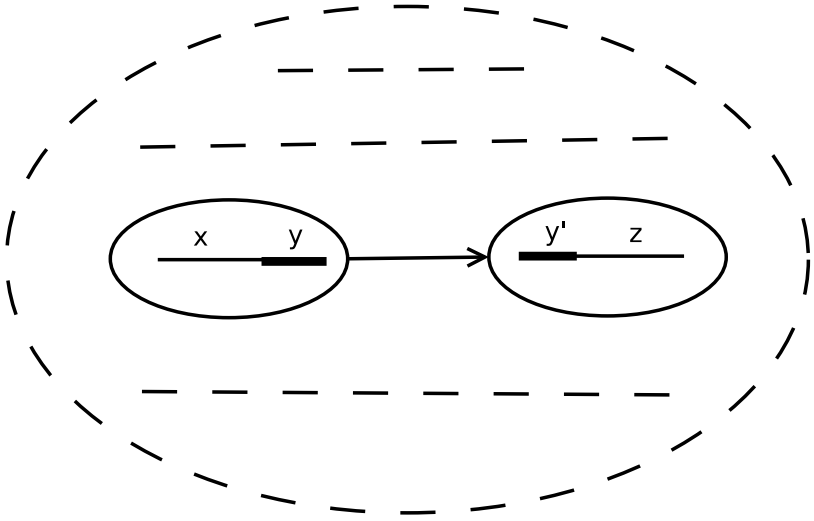

Figure 2.1: OLC graph with edge between two vertices having suffix-prefix overlap

Such a graph captures the overlap between different reads and provides an opportunity to iteratively merge reads and produce long contigs. Specifically, if $r_i = x.y$ and $r_j = y'.z$, then the edge $v_i \rightarrow v_j$ gives an opportunity to extend $r_i = x.y$ to form a longer substring of the genome $x.y''.z$ (y'' is deduced from y and y'). Extending this further with more edges would then result in larger contigs.

With this formulation, the genome representing the reads can be seen as a Hamiltonian cycle in the graph.

Similar to the OLC approach is the string graph approach [12] which also looks at overlaps between reads. For each read r_i, two vertices $r_i.b$ and $r_i.e$ are created (b and e referring to beginning and end of overlap). If reads r_i and r_j have a good suffix-prefix overlap, then an edge is created from $r_i.e$ to $r_j.b$ and from $r_j.e$ to $r_i.b$.

This way, r_i and r_j can both be extended to form longer contigs.

Before the arrival of NGS sequencing machines, OLC was the basis of most assemblers. Examples of most popular OLC based assemblers are Celera [13], Arachne [14], SSAKE [15], Edena [16] and CAP3 [17]. All these assemblers were designed for Sanger reads. However, NGS machines increased the sequencing throughput to billions of reads in a single experiment, making construction of OLC graphs for such large datasets computationally challenging. Considering all-pairs of overlaps to construct the graph would incur $\Omega(n^2)$ cost. In addition, the size of the overlap graph grows with the size of read dataset. Finally, short reads make overlaps between individual pairs of reads less reliable. All these factors rule the OLC approach difficult to adopt. Even though some clever heuristics were designed to greedily merge reads to form longer contigs [14, 13], the assembly techniques for NGS sequencing drifted to a more compact framework, called *de-Bruijn graphs*, discussed in next section. In chapter 7, we will discuss how we address some of the challenges posed by the OLC approach.

2.2 De-Bruijn graph approach

Similar to OLC, the DB approach also involves construction of a graph based on the read set. The method of construction however, is different. First, all the reads in the read set are split into shorter substrings called *k-mers*. A k-mer is a k length substring of a read. Thus a read of length l would yield $l - k + 1$ *k-mers*. We then construct the DB graph as follows:

a. A vertex v_i is constructed for every distinct k-mer k_i obtained from the read set.

b. An edge $v_i \rightarrow v_j$ is added if the k-mer k_i and k_j are adjacent in some read in the readset.

The concept of extending a k-mer by edge traversal is very similar to that in OLC approach. An edge allows us to extend the current sequence by 1 base. Unlike in

OLC however, it is allowed to traverse a vertex multiple times in a DB graph (since a k-mer can appear in multiple places in the genome). Thus constructing a genome in this graph is equivalent to a tour with each vertex appearing as many times as the coverage of the region where the corresponding k-mer belongs.

The DB graph is more compact than the OLC graph discussed in the previous section. While the number of vertices in the OLC graph is equal to the number of reads, the number of vertices in a DB graph is bounded by the number of distinct possible k-mers, which is 4^k. Moreover, if error correction is perfect, that is, all the errors are removed from the read set, the number of vertices is further bounded by $L - k + 1 = O(L)$, where L is the length of the genome.

Most current short read assemblers follow the de-Bruijn graph paradigm, first introduced by Idury et al. [18], and further developed by Pevzner et al. [19]. At first glance, such an approach of breaking reads into k-mers might appear counterintuitive, since we are potentially throwing away precious overlap information between a series of k-mers. This approach was however justified in [19] as it transposes a hard problem into a more tractable one. Furthermore, our proposed approach of introducing *distance constraints* in de-Bruijn graph (explained in chapter 4) further mitigates the negative effects of this. In the next chapter, we briefly discuss some of these assemblers.

CHAPTER 3
REVIEW OF RELATED LITERATURE

We discuss the current spectrum of assembly work in this chapter and lay out the motivation behind our work.

Velvet [7] was the first assembler for NGS reads to use the de-Bruijn graph (DB graph) framework for assembly. It also addressed in detail the issue of error detection and correction in the de-Bruijn graph setting. After Velvet, many assemblers like ABySS [6], ALLPATHS [8, 20], SOAPdenovo [21, 22], Meraculous [23], SWAP [24] followed the de-Bruijn graph framework to generate assemblies. Of these, SWAP assembler does not consider paired read information while performing assembly. With minute differences, ABYSS, Meraculous and SOAPdenovo use the following pipeline for assembly: they construct the de-Bruijn graph from reads, follow several subroutines for graph cleaning, produce unambiguous chains as initial set of contigs and then map the reads back to these contigs to extend and join them based on paired read information.

IDBA [25, 26] extend the DB graph approach by iteratively building the DB graph for different values of k-mer size. The DB graphs for different k values are then merged in hope of producing an assembly that has the best of small k-mer and large k-mer sizes. Few assemblers like SGA [27] and SAGE [28] use String graphs to produce assemblies. SGA uses them to achieve memory efficiency and parallelizability in genome assembly. SAGE on the other hand performs transitive reductions on edges in the string graph. They couple this with maximum likelihood genome assembly approach [29] to produce contigs.

MaSuRCA assembler [30] proposed a hybrid approach to get the best of two worlds, computational feasibility of de-Bruijn graphs and inclusiveness of the OLC approach.

This is done by converting the original set of reads into "super-reads" which are unambiguous extensions of original reads using the overlap information, and then applying an OLC like approach on these super-reads which are far fewer in number than original reads. However it is unclear how insert size variations are taken into account in this approach. DISCOVAR [31] assembler was designed specifically for identifying varients in polyploid genomes (organisms in which cells contain multiple copies of the genome with slight variations, often coming from different parents).

Given that the assembly problem is compute intensive, efforts have been devoted to parallelization of existing methods to achieve assembly with smaller runtimes. ABySS 2.0 [32] redesigned the original ABySS implementation by replacing the MPI parallelization framework with bloom filters, which is a probabilistic data structure. This was done in order to reduce the memory footprint of the tool. HipMer [33] parallelized several stages of the Meraculous assembler using UPC's one sided communication capabilities. While obtaining faster assembly is an important research area in its own right, our focus in this project is to enhance the quality of genome assembly.

As described in section 1.3.3, there are two classes of read types with different characteristics. One class of reads have short lengths (*short reads*: 100-300 bp) and are highly accurate (less than 1% error rate), produced by sequencers from companies like Illumina. The second class of reads have a much larger length (*long reads*: 5 kbp - 1000 kbp) and have much higher error rate (11-15% errors), produced by sequencers from companies like PacBio and ONT. Due to such large varying characteristics, different assembly algorithms are needed to address the challenges posed by different read characteristics. Most assembly softwares are designed to assemble one specific kind of read dataset. All the assembly software discussed so far in this chapter are designed to work on Illumina style short reads. Separate softwares like Canu [34], Hinge [35], just to name a few, have been designed for long reads. As our work is completely focussed on short reads, we will not go into algorithmic details of long read assemblers.

Recent assembler evaluations like the Assemblathon [1, 2] and GAGE [3], have shown that no single assembly software performs universally better than others.

As described earlier, paired end sequencing is a technology by which the two end portions from complementary strands of a larger DNA fragment of approximate known size are sequenced. This technology can be a key factor in improving assembly quality. Paired reads are particularly helpful in resolving repeats that are longer than the read length. When one read of a pair falls in a repeat region of the genome but the other one does not, the read that is from the non-repetitive region can anchor the other correctly. A common idea used in many assemblers is to compact all the linear chains of nodes that have an in-degree and out-degree of exactly one, into one sequence per chain. These sequences are produced as the first set of contigs. The reads are then mapped back onto the graph or the contigs and the paired read information is used to merge these contigs, whenever possible.

The above approach leads to the initial assemblies to be based solely on read overlaps, relegating the valuable paired end information to a secondary status. This approach causes crucial loss of information when performing graph traversals to produced contigs, and can lead to the paired read information not being used to its fullest potential, resulting in inferior assembly. Directly incorporating distance information from paired end reads in the assembly should lead to improved assemblies. Several assembly approaches have advocated and pursued this approach. The ALLPATHS assembler [8, 20] initially tracks all possible paths out of a node until a threshold length is reached, followed by selection of one of these paths based on evidence from paired end distance constraints. The process is repeated at the end node of the current path until an ambiguity arises. Jackson et al. [36] overcome the combinatorial complexity by developing a backward looking approach where a path is extended by choosing the edge (from among the possible edges) that is connected by several distance constraints to the path behind.

Medvedev et al. introduced paired de-Bruijn graphs [37] (PDBG) that can represent exact distances between pairs of k-mers. As the paired end distances are approximate, they adapted this approach to handle approximate repeats using *A-Bruijn* graphs in their SPAdes assembler [38]. Though *A-Bruijn* graphs are better suited for approximate repeats, their complex structure makes it difficult to analyze them. SPAdes implements Pathset graph data structure [39] to utilize paired read distance constraints to resolve longer repeats. It maps the reads onto the assembly graph and creates a histogram of distances between two given chains of nodes and adjusts it to one or few distance value(s) based on the peak(s) of histogram. While this method is expected to work well when a clear peak is identifiable, it is not clear what can be done when this is not the case. Although these computational models have been proposed to utilize paired read information into assembly, none of the methods have been shown to work on real datasets that are large and complex.

We present a novel approach for utilizing paired read distance constraints for genome assembly. We transform the distances between paired reads into distance constraints between nodes of the de-Bruijn graph. We then generate paths from the graph using these constraints as guide to resolve ambiguities. Each path produced represents a contig. When faced with multiple node choices for extending the current path, each node is scored based on the embedded distance constraints followed by the selection of the highest scoring node. A correct traversal algorithm needs to ensure that each node is visited the number of times it appears in the genome. We describe a model to deduce when a node has been visited sufficient number of times and hence should not be visited in any further traversal steps. The graph traversal terminates when all the nodes have been visited sufficient number of times. This approach is described in detail in chapter 4

In order to assemble large genomes in reasonable time, we adapt the above approach and design a parallel algorithm that produces multiple paths in the DB graph (and

hence multiple contigs) simultaneously. This algorithm is described in detail in chapter 5.

CHAPTER 4
UTILIZING PAIRED READS TO IMPROVED ASSEMBLY QUALITY

In this chapter, we describe the first part of our contribution to improving genome assembly. We describe our work in the context of the related work discussed in chapter 3.

4.1 Distance Constraint based de-Bruijn Graph Assembly

Our algorithm utilizes many of the typical steps used in de Bruijn graph based assemblers. However, it differs in a few distinguishing characteristics. The first is direct encoding of paired-end information in the form of distance constraints between pairs of nodes in the de Bruijn graph. We also supplement this information by adding additional distance constraints that connect the beginning of a read to the corresponding end of the read. As the read is fragmented into its constituent k-mers, these additional distance constraints help navigate the de Bruijn graph in a way that preserves the sanctity of reads. The second is a novel algorithm that generates long contigs by finding paths in the graph supported by evidence from distance constraints. In contrast to many assemblers that merely produce compacted chains in the graph as initial contigs, our algorithm continues graph traversal at junction nodes by working to resolve ambiguity using distance constraints. The algorithm also avoids the combinatorial complexity of assemblers such as ALLPATHS [8, 20] that track all possible outgoing paths from a given node for a specified distance to determine which path is supported by paired-end information. Thirdly, we do not assume any value for the insert size estimate it from the read dataset itself. This is described in section 4.1.6. In what follows, we present comprehensive details of our algorithmic strategy.

4.1.1 Bidirected de-Bruijn Graph Formulation

We use a variation of the de-Bruijn graph framework described earlier, called the bidirected de Bruijn graph (BDBG) framework, used by Jackson et al. [36]. The BDBG framework naturally captures the double stranded nature of DNA sequences, allowing us to avoid the problem of switching strands while traversing the graph. In this framework, nodes in the de Bruijn graph represent k-molecules, consisting of a k length DNA sequence α and its complementary strand α'. The lexicographically smaller of the two sequences is called the *positive strand* of the molecule, which is also designated as the *representative strand*. The other strand is called the *negative strand* of the molecule. Figure 4.1 shows a k-molecule as part of a double stranded sequence, with positive and negative strands marked as blue and red respectively.

A *positive extension* of a k-molecule is achieved by adding a base at the end of the positive strand, and the complementary base at the beginning of the negative strand. If the positive extension is followed by removing a base from the beginning of the positive strand, and the complementary base from the end of negative strand, the operation is called a *positive shift*. The operations *negative extension* and *negative shift* are defined in a similar manner towards the other side of the molecule. Figure 4.1 shows the directions for positive and negative extensions/shifts for the k-molecule. Intuitively, a shift of a molecule gives a potentially neighboring molecule in the reference genome.

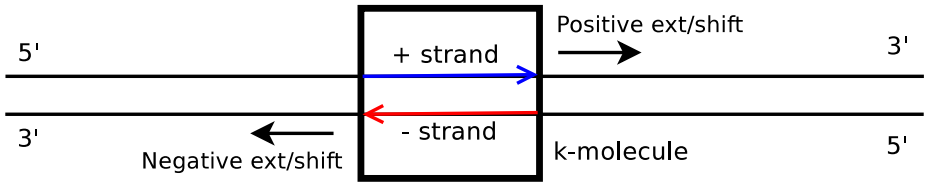

Figure 4.1: k-molecule from a strand

A BDBG is constructed by extracting all the k-mers from reads and constructing k-molecules corresponding to each k-mer. The nodes of the graph are all the unique k-molecules so obtained. The $frequency$ of a node is the number of times either the

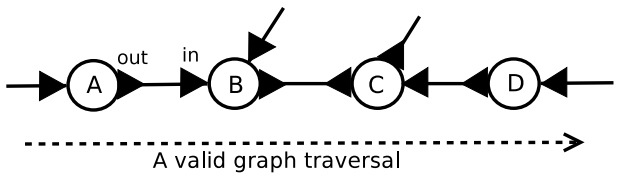

Figure 4.2: Respecting the opposite directionalities on each node while traversing

positive or the negative strand of the corresponding k-molecule is observed in the readset. The edges in a bidirected graph are of the form $(< u, d_u >, < v, d_v >)$ where $d_u \in \{out, in\}$ is the direction of edge at node u. The edges in the BDBG represent *(k+1)-molecules* obtained by *(k+1)-mers* present in the readset. The directions for edges are determined as follows: if v is obtained by a positive shift of u, then $d_u = out$, else $d_u = in$. The *multiplicity* of an edge $(< u, d_u >, < v, d_v >)$ is the number of times the extension of u leading to v is observed in the readset. A valid traversal through the graph must respect the directions at each node as follows: if a path enters a node through *out* direction, it must exit the node through *in* direction, and vice versa. Figure 4.2 shows a valid traversal through the graph. The direction *out* (*in*) of an edge at a node is denoted by an arrow pointing out of (into) the node, illustrated in the figure. Note that there is no constraint on the two directions of any edge that is traversed.

4.1.2 Generating Distance Constraints

As discussed before, paired end sequencing protocols sequence pairs of reads from opposite ends and complementary strands of a DNA fragment, typically called an *insert*. The approximate length of the insert is either available along with the readset, or can be deduced from the readset. The method of deducing insert size from readset is described in the subsection 4.1.6. The insert size imposes a *distance constraint (DC)* between the genomic positions of reads from the pair. To embed this information into the BDBG, we transform the DCs between reads into DCs between k-molecules as shown in Figure 4.3. We introduce a DC between k-molecules corresponding to the

farthest k-mers in every read pair. This DC follows from the fact that the mean insert size d is known, and hence the two k-molecules are at a distance of approximately d from each other in some part of the genome. We refer to such a DC as an *inter read DC*.

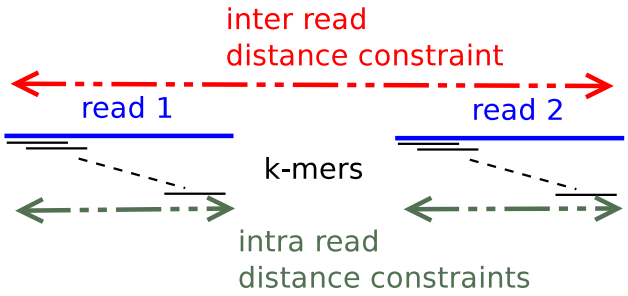

Figure 4.3: Inter and intra read distance constraints between k-molecules

In addition, we introduce a DC between k-molecules corresponding to the first and last k-mer of every read. Since the read length l is known, this DC implies that the two k-molecules are at an exact distance of l from each other in some part of the genome (except for errors caused due to chimeric fragments). We refer to such a DC as *intra read DC*.

4.1.3 Embedding Constraints into the de Bruijn Graph

The DCs between k-molecules are embedded into the de Bruijn graph as DCs between corresponding nodes in the graph. For any DC, these nodes are referred to as *DCnodes*. A DC of length s between nodes u and v requires that some path traversed should visit nodes u and v (not necessarily in that order) with s (exact in case of intra read or approximate in case of inter read) k-molecules in between them.

In addition to the two DCnodes and the distance, a DC has a few additional components. As stated, the known insert size for paired end sequencing is imprecise, and hence a good traversal algorithm must permit a corresponding *error* in the DCs (the error depends on the variance in insert size observed using the method described in subsection 4.1.6). Moreover, a DC should respect the *directionality* in

which it was observed. For instance, suppose a read consists of m k-mers. Let they be ordered by starting position as $\{k_1, k_2, \ldots, k_m\}$, and let the corresponding k-molecules be $\{K_1, K_2, \ldots, K_m\}$. Then the intra read DC between K_1 and K_m will have two directionality components, one for each DCnode. The directionality at K_1 is positive (negative) if K_2 is obtained using a positive (negative) shift of K_1. Similarly, the directionality at K_m is positive (negative) if K_{m-1} is obtained by a positive (negative) shift of K_m. Similar principle applies for the directionality of inter read distance constraints.

4.1.4 Displacing Distance Constraints

A majority of the nodes in the de-Bruijn graph have a degree of two with opposite directions on the two edges. We call such nodes as *unitig nodes*, and a chain of such nodes as a *unitig chain*. Such nodes have no ambiguity while traversing through them. Thus compacting a chain of such nodes is beneficial in reducing the graph size and subsequent traversal time. However, some of these nodes might be DCnodes involved in DCs embedded earlier, and we would like to preserve them while compacting the graph.

To achieve this, we displace such constraints to the end of the respective chains as shown in figure 4.4. A DC with at least one DCnode in the middle of a unitig chain is shortened by moving the ends of the constraint to the ends of the unitig chains in which the constraint ends belong, and adjusting the distance accordingly. This is performed for all the constraints in the graph. With this displacement, we can compress the chains of unitig nodes and still maintain appropriate distance constraint relationships between nodes.

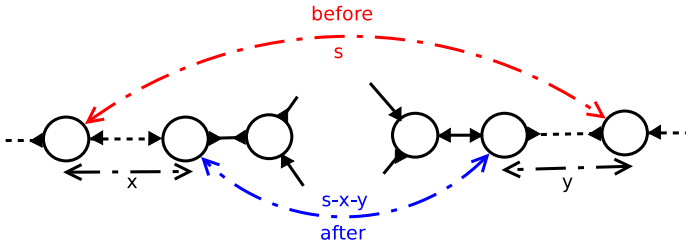

Figure 4.4: Displacing distance constraints to end of a unitig chain

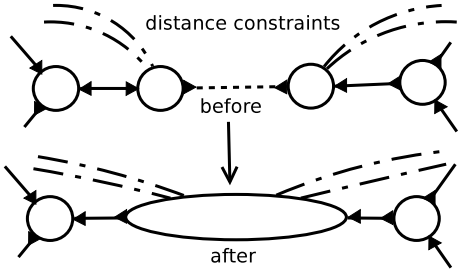

Figure 4.5: Compacting a unitig chain into a single node

4.1.5 Compacting Unitig Nodes

After the DCs have been displaced, we compact every unitig chain into a single node as shown in Figure 4.5.

Such nodes will no longer be k-molecules, since their strands would be formed by combining the k-molecules of the chain, where consecutive k-molecules have a $k-1$ length overlap. Their strands will have a length $h \geq k$, and we refer to such nodes as *h-molecules*, and their strands as h-mers. For ease of description, we define the *length* of a node as the length of its strands.

4.1.6 Estimating insert size

To be able to use the paired read information, it is necessary to have an estimate on the insert size. Most sequencing instruments do not provide such information. Therefore, we estimate the size from the data itself. In order to do so, we first select the subset of read pairs Ψ from the dataset such that both the reads in a pair map to the same chain in the de-Bruijn graph. Since each pair $p = \langle r_1, r_2 \rangle$ in Ψ maps to a

single sequence, the insert size corresponding to p can be calculated. Using the insert sizes corresponding to all pairs in Ψ, we calculate their mean and variance. Using these values and the insert sizes for read pairs in Ψ, we try to fit various statistical distribution models (e.g. Normal, Gamma, Binomial) on to the insert size data and select the distribution model that gives the minimum error. For most datasets, the set Ψ is large enough to obtain a statistically meaningful distribution on the insert size values. We therefore use the calculated estimate as the insert size for performing assembly. SAGE assembler has a similar method for estimating insert size on string graphs.

4.2 Traversing de-Bruijn Graph to Generate Contigs

We present a graph traversal algorithm that operates on the above constructed graph, and utilizes embedded and displaced DCs to produce a set of paths that are as long as possible. Each path represents a contig of the genome, and is constructed incrementally. The paths themselves are constructed successively, one after another. In what follows, we first present our path extension algorithm that extends a partial path constructed so far by finding the next node to include in the path. We refer to the last node in the path as *current node*. The next node to visit is chosen from among the neighbors of the current node that are connected by an edge which can be traversed (i.e., having the opposite directionality to how path constructed so far entered the current node).

We use the embedded DCs to help resolve the ambiguity when presented with multiple options for choosing the next node. This is achieved by using a supporting queue data structure Q. The queue conceptually represents the contig corresponding to the current path, with one base pair at each position. The de Bruijn graph nodes are pushed into various queue positions based on where they align on the current contig. Algorithm 1 describes the pseudocode for the graph traversal algorithm, which is elaborated in the following subsections.

Algorithm 1 Graph traversal algorithm.

```
1: while true do
2:    select_start_node()
3:    if no starting node then
4:       break;
5:    end if
6:    while true do
7:       select_node_direction()
8:       update_candidate_nodes()
9:       update_distance_constraints()
10:      select_next_node()
11:      if no node selected then
12:         break;
13:      end if
14:   end while
15: end while=0
```

4.2.1 Selecting the Start Node

To initiate a path, we need to select the beginning node. We select the longest available node to start the traversal and push it at the beginning of Q. Long nodes have a good likelihood of having a unique occurrence in the genome, which helps in determining the number of occurrence of other nodes in the path. Figure 4.6 shows the supporting queue data structure Q, with the starting node S pushed at the beginning of queue. Our algorithm also contains a method to estimate the number of times each node in the graph should be visited during the entire traversal, detail of which are presented in subsection 4.2.8. When all the nodes have been visited sufficient number of times, no starting node is selected and the traversal is terminated.

4.2.2 Selecting Node Direction

To extend the current path, we need to select a direction for the current node in the path in which traversal must continue. If the path consists of only the starting node, the direction is chosen arbitrarily. Otherwise, the traversal must be consistent with the edge direction used to enter the current node (discussed in subsection 4.1.1). If

the path entered the current node using the *in* direction, then it must exit using the *out* direction and vice versa.

4.2.3 Updating Candidate Nodes

The current path is extended by choosing one node from the neighbors of the current node in the selected direction. To achieve this, we push all the neighboring nodes as *candidate nodes* in Q. The nodes are pushed at a position so that they align with the current path, specifically with the current node. In other words, if the current node has length h, then the candidate nodes are pushed at a distance $h - (k - 1)$ from the current node. Figure 4.6 shows candidate nodes A and B (neighboring nodes of S) pushed into the queue, depicted as *triangle shaped nodes*.

4.2.4 Updating Distance Constraints

If the current node has any DCs to be satisfied, they are pushed into the queue as well. These constraints are helpful in deciding the next node to traverse from the list of candidate nodes. If the current node, with length h, has a DC of distance d to a node A, then node A is pushed as an *evidence node* at a distance $d + h$ from the current node. This is because the DCs are measured from ends of the two nodes, whereas position of a node in Q represents the beginning of that node. An evidence node A at a position p implies that some node in the path already traversed expects node A to align to the current path at position p. Evidence nodes are depicted using *square shaped nodes* (Figure 4.6).

4.2.5 Selecting the Next Node

This part is responsible for selecting one node from the list of candidate nodes for traversing and extending the current path. A good selection strategy is the key to producing quality contigs, making this the most crucial part of the traversal algorithm.

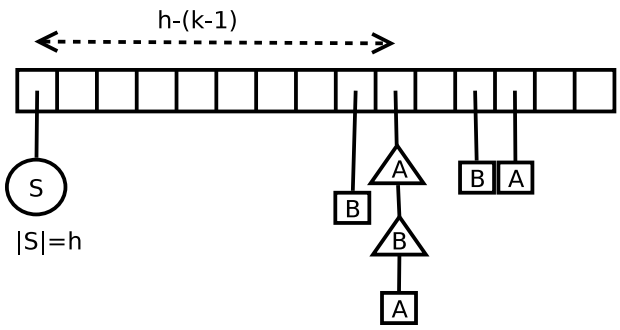

Figure 4.6: A snapshot of the queue data structure

For this section, we will assume that there are at least two candidate nodes to choose from. The candidate node selected to extend the current path is referred to as *winner*. Algorithm 2 describes the module for selecting the winner, and we elaborate on it as follows.

Algorithm 2 Select next node.

1: **for** all candidate nodes $\{C_i\}$ **do**
2: $\quad \overrightarrow{EV_i} = gather\_evidence(C_i)$
3: $\quad score_i = \mathcal{F}(\overrightarrow{EV_i})$
4: **end for**
5: Pick candidate with max score =0

Gathering Evidence

For each candidate node, we need to determine the extent of *support* it has in the form of evidence nodes. The length of the current node determines the exact position where we seek to begin the next node, for which we plan to choose from among the candidate nodes (indicated by triangle nodes at that position). Since the inter read DCs are not precise, we need to account for laxity in them. To accommodate this, we look for support for each candidate node within a range around the position where it should lie, as shown in Figure 4.7. The size of this range depends on the variance in insert size calculated using the method described in subsection 4.1.6. Every evidence node in support of a given candidate node increases its likelihood of being chosen as the winner. We collect the evidence for a candidate node as a vector of pairs (d, n),

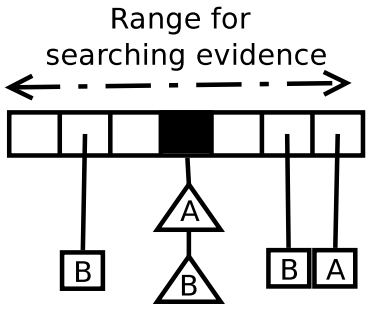

Figure 4.7: Evidence searching range around candidates' position

where d represents the distance from the candidate node position at which there are n evidence nodes supporting this candidate node. We refer to this vector as *evidence vector*.

Calculating Candidate Score

Once the evidence vector $\overrightarrow{EV_i} =< (d_1, n_1), (d_2, n_2), \ldots (d_m, n_m) >$ is computed for all the candidate nodes c_i, we would like to compare them to choose a winner. To do this, we score each candidate using a *scoring function* $\mathcal{F}$. The scoring function takes as input the evidence vector of a given candidate and gives as output the candidate score. We discuss the design of scoring function in a later subsection.

Given the complex nature of the problem at hand, finding a scoring function that leads to the correct choice every time is challenging. A number of reasonably good scoring functions can be designed, and each one is expected to fail on a few occasions. To avoid our results getting affected by arbitrarily choosing one reasonable scoring function out of many, we define a parameter called *minimum score difference*, denoted as δ. The parameter is used to avoid the bias of a specific scoring function as follows: Let A and B be the two candidates having the best score S_A and the second best score S_B respectively. We require that for A to be chosen as the winner, $S_A - S_B \geq \delta$. The rationale for this is that in this case, we assume that the two scores have enough difference that any good scoring function would have chosen A as the winner, though with possibly different scores for various candidates. On the other hand, if $S_A - S_B < \delta$,

we consider the scores to be "too close", and thus A might have been chosen as the winner just due to a particular scoring function. In this case, we explore all the paths emerging from the current node, as described in a later subsection.

Scoring function

Designing a reasonably accurate scoring function is perhaps the most important part of this assembly pipeline. An accurate scoring function is crucial for a high quality output. An intuitive observation for the scoring function is that it must reward high values of $n's$ (number of items of evidence). This follows from the observation that higher quantity of evidence for a node indicates more nodes in the constructed path signaling pairing information with the node under consideration. Secondly, the scoring function must be punitive towards high values of $d's$ (distance of the evidence from the node under consideration). This follows from the observation that the farther an evidence is from the node under consideration, the less probable it is to be an evidence for that node in reality, suggesting the candidate node may be the wrong choice. With these considerations, we use a scoring function of the following format for our algorithm

$$\mathcal{F}(\overrightarrow{EV}) = \sum_{i=1}^{|\overrightarrow{EV}|} f\left(n_i\right) \times p\left(d_i\right) \tag{4.1}$$

Here $f\left(n_i\right)$ represents the weight given to n_i number of distance constraints. We currently use $f\left(n_i\right) = \alpha\left(n_i\right)^{\beta}$ where α and β are tunable parameters. $p\left(d_i\right)$ is a *scaledown* that we apply in accordance with the insert size distribution we observe as described in section 4.1.6. If the distribution curve observed is denoted as G (origin at the mean value), then we use $p\left(d_i\right) = \frac{G(d_i)}{G(0)}$, as shown in figure 4.8. The rationale for this is that the distribution curve provides the probability distribution of deviation of insert size from mean, and therefore we use this probability to scaledown the effect of an evidence, depending on the deviation at which it is observed.

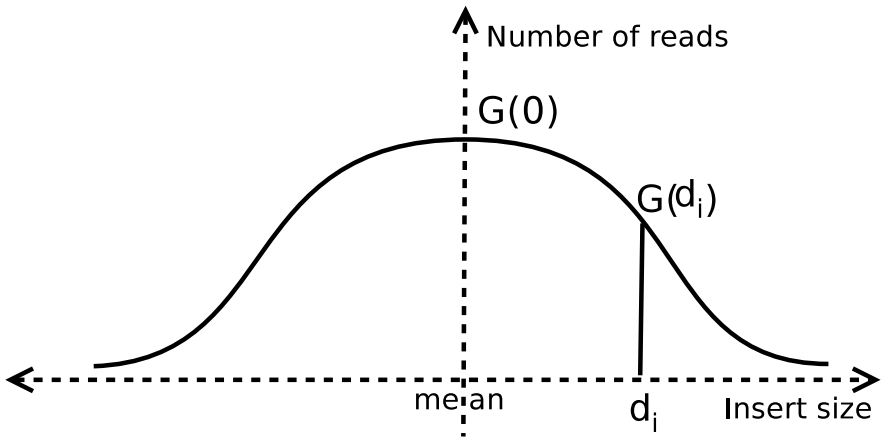

Figure 4.8: Scaledown used for scoring function

One caveat with this scoring mechanism is that it is biased in favor of longer candidate nodes. This is because a long node comprises of a chain of a large number of k-molecules. Hence even if a shorter candidate node is the correct choice, it might lose to a long candidate node just because a long node has more potential targets for DCs, thus increasing the number of evidence nodes supporting it. To deal with this, we normalize this score to the length of the candidate before comparison.

4.2.6 All paths exploration

Sometimes the decision process described above might not yield any node to extend the path already constructed. This can happen under various circumstances:

a. There is no neighboring node to extend the current path.

b. All neighboring nodes are *saturated* (The term saturated is defined in section 4.2.8. To put it simplistically, they have been visited "enough" number of times during previous traversals and are therefore unavailable to be used for path extension)

c. Multiple candidates have strong evidence scores and none of them exceeds its peers by the threshold δ, discussed in a previous section.

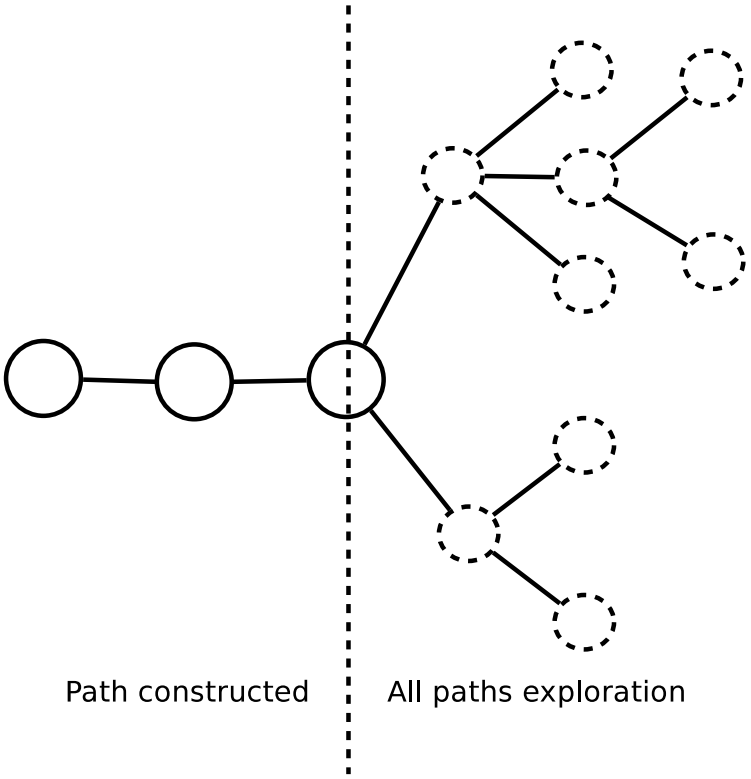

Figure 4.9: All paths exploration

While nothing much can be done in cases a and b, we try to extend the path for case c as follows: All forward paths emerging from candidate nodes are explored as shown in figure 4.9. The exploration continues until a score difference of δ is obtained between the best and second best paths till a certain depth. If no conclusion is reached till all the paths have been explored until a maximum path length l_{max}, we terminate the current contig at the current node.

4.2.7 Bubble detection

Detecting bubbles in the graph while traversing it is an important task. Bubbles in a graph are defined as two paths that are identical except a few nodes differing in the middle of the paths. For example, two paths $\langle v_1, v_2, v_3, v_4, v_5 \rangle$ and $\langle v_1, v_2, v_3', v_4, v_5 \rangle$

would form a bubble $v_1 - v_2 - \left\langle \begin{matrix} v_3 \\ \\ v_3' \end{matrix} \right\rangle - v_4 - v_5$ in the de-Bruijn graph.

Several situations in the genome assembly problem can translate to forming bubbles in the de-Bruijn graph

- *Approximate repeat regions*: Repeat regions in the genome that are identical apart from a few bases would form a bubble in the de-Bruijn graph representing the differing part. Identifying such bubbles enables traversal over them, thereby elongating the length of the contig being produced.

- *Read errors*: Reads containing errors would differ from the genome as well as other correct reads that span the erroneous position. This would result in a bubble where the two branches would represent the correct reads (usually high frequency) and the erroneous reads (usually low frequency). Identifying such bubbles help removal of erroneous branch and increasing the quality of the contig being produced.

- *Biological mutations*: Organisms that are diploid (e.g. humans) contain two copies of the genome, one inherited from each of their parent. Both copies participate in the sequencing process (i.e. reads are produced from both the copies simultaneously). The two copies however, arising from two different organisms (mother and father in case of humans) are not identical and have minor differences. These differences are referred to as Single Nucleotide Polymorphism (SNP) These differences appear as *mutations* in the reads. Specifically, the reads sequencing a SNP position will be in disagreement at that position since the reads are originating from two sources that have a different base present at the SNP position.

  In terms of de-Bruijn graph, the biological mutations are similar to read errors

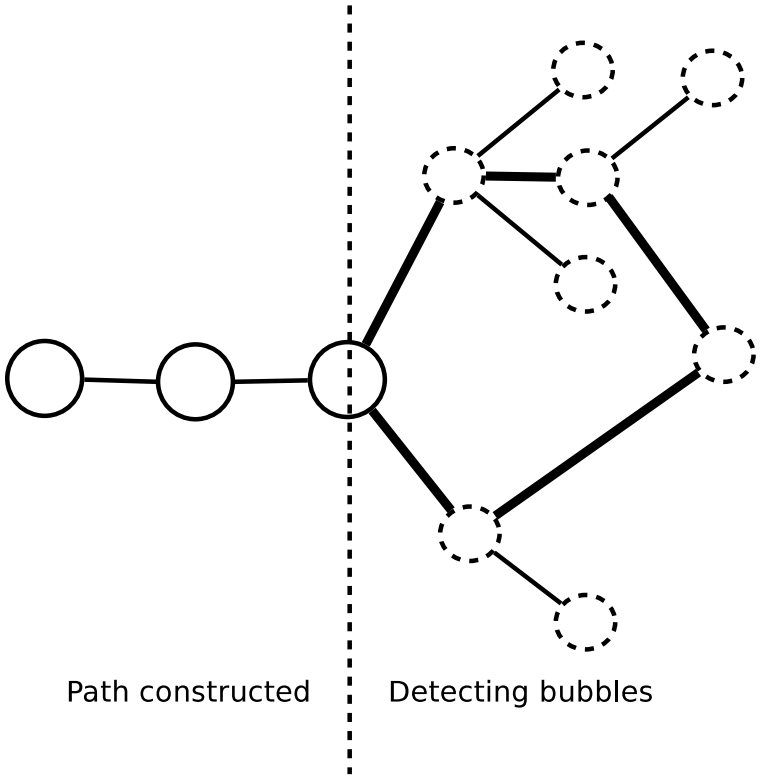

Figure 4.10: Detecting bubbles while exploring all paths

and form similar de-Bruijn graph structure as read errors, except that the two branches in case of mutations will have similar frequency (since odds of sequencing from two copies are similar) whereas in case if read errors will have highly differing frequency (since error rates for the reads we use is low). On identifying bubbles that correspond to mutations, we can traverse over one branch and report the other as a mutant along with the assembly.

During traversal, we detect bubbles as part of the all paths exploration routine (discussed earlier) as shown in figure 4.10. When exploring all paths, if we encounter two identical nodes in the exploration tree, we check if their sequence lengths are similar (that is, their lengths differ by less than a maximum threshold). If so, we declare the structure as a bubble. We then categorize it into the appropriate kind of bubble (approximate repeat, errors or mutations) and address it accordingly.

4.2.8 Estimating Node Visit count

A key challenge in traversing the de Bruijn graph is deducing the number of times each node should be visited during graph traversal. An ideal traversal should visit a node v as many times as its sequence appears in the genome (referred to as η_v). When v has been visited η_v number of times, v is said to be *saturated*. This deduction is particularly important to be able to terminate the traversal after all the nodes in the graph have been saturated. Moreover, this is also useful in terminating the current path when all the candidate nodes for extension have been saturated.

For uniform read coverage cov, η_v can be obtained as $\eta_v \approx \frac{f_v}{cov}$, where f_v is the frequency of occurrence of the molecule in v in the readset. Since the read coverage in practice is not perfectly uniform, determining η_v is far from trivial. We describe a method to estimate η_v for a k-molecule v. It is straightforward to extend this to a h-molecule for $h > k$.

Let $\{p_1, p_2, \ldots p_\eta\}$ be the genomic positions where the strands of v occur in the genome. Let $\{c_1, c_2, \ldots c_\eta\}$ be the coverages at respective positions. Then we have the following equation.

$$f_v \approx \sum_{i=1}^{\eta_v} c_i \tag{4.2}$$

Assume we are at node v while traversing the graph, and let this visit to v correspond to position p_i for some i. If there is a mechanism to deduce the coverage at current position p_i (that is c_i), then we can add c_i to a variable (called $usage_v$, initiated to 0), and do this every time v is visited. When the condition $usage_v \approx f_v$ is satisfied, we know that v has become saturated and that v should not be visited anymore in our traversal. Therefore the problem of estimating η_v reduces to estimating the coverage at the genomic part corresponding to current node of traversal, which we describe in the following subsection.

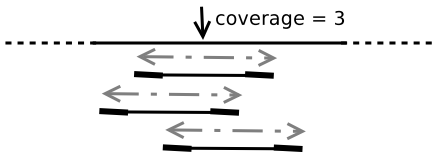

Figure 4.11: Coverage at a given position is the number of intra read DCs crossing it

Coverage estimation

In this section, we propose a method to estimate read coverage at a genomic position p (referred to as cov_p) corresponding to a node v. Since the purpose of this estimation is to eventually deduce η_v, a close estimate is expected to suffice. In addition to coverage at a specific position, we also discuss coverage in a short genomic region X (referred to as cov_X), which is expected to be roughly constant throughout the region, assuming that the region is too short for any significant coverage variation.

Coverage at any given genomic position is the number of times the position is sequenced in the reads. Since we have an intra read DC between two ends of every read, the coverage at any given position, cov_p is nearly equal to the number of intra read DCs that have the two ends on opposite sides of the position, as shown in figure 4.11. Such DCs are said to *cross* over p and the number of such DCs is referred to as $genome\_intra\_cross_p$. Therefore,

$$cov_p \approx genome\_intra\_cross_p \tag{4.3}$$

During the de-Bruijn graph traversal, $genome\_intra\_cross_p$ for a given position p can be estimated as follows: After traversing the graph to create a path containing v, we can count the number of intra read DCs that have the two DCnodes on opposite sides of v (DCs *crossing* over v). For a node v in path P, we refer to this number as $path\_intra\_cross_{v,P}$ or simply $path\_intra\_cross_v$ when the path in the discussion is implied. Thus $path\_intra\_cross_v$ will be the coverage at the current genomic point of traversal corresponding to v, and thus will be equal to cov_p.

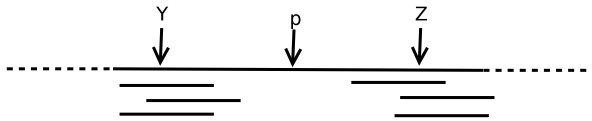

Figure 4.12: Coverage at a given position is related to coverage in nearby regions

The method described will work if genomic position p is not part of a long repeat. However, if it is part of a long repeat, most of the DCnodes of DCs crossing over v are expected to come from the repeat region. In this case, the above method will give the sum of coverages for all such repeat regions. Thus $path\_intra\_cross_v$ will not provide a good estimate of $genome\_intra\_cross_p$. It is unclear how to separate intra read DCs from different repeat regions to get a good estimate of $genome\_intra\_cross_p$.

To handle this, we use *inter* read DCs instead of *intra* read DCs. An advantage of inter read DCs is that their distance is generally longer than that of intra read DCs, and therefore their DCnodes are less likely to be part of a repeat region. Hence the inter read DCs are more reliable than intra read DCs for coverage estimation. We define $genome\_inter\_cross$ and $path\_inter\_cross$ for inter read DCs similar to $genome\_intra\_cross$ and $path\_intra\_cross$ respectively for intra read DCs. We use inter read DCs to estimate the coverage as follows.

Though the read coverage for the entire genome is non-uniform, sudden coverage changes between nearby positions are rare. Therefore, to deduce cov_p at a position p, we use the coverage of genomic regions in proximity of p. Figure 4.12 shows pairs of reads with each pair having the two reads mapped in genomic regions Y and Z. Since the position p is in proximity to regions Y and Z, cov_p is expected to increase (decrease) as cov_Y and cov_Z increase (decrease). Moreover, increase (decrease) of coverage at cov_Y and cov_Z also results in increase (decrease) of $genome\_inter\_cross_p$. Therefore, we use the following model for estimating coverage

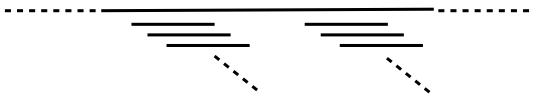

Figure 4.13: Synthetic example to calculate proportionality constant in coverage calculation model

$$cov_p \propto genome\_inter\_cross_p$$

$$\implies cov_p = \beta \times genome\_inter\_cross_p$$

To calculate β, we use the scenario shown in figure 4.13. Each read pair has two reads of length l each, and are obtained from an insert of size d. Therefore the unknown gap between the two reads has length $d - 2l$. There is one read pair mapped at every position in the genome. In this scenario, for any position q, the coverage at that position is:

$$cov_q = 2l$$

And the number of inter read DCs crossing over q is:

$$genome\_inter\_cross_q = d$$

Therefore, we have

$$\beta = \frac{cov_q}{genome\_inter\_cross_q}$$

$$\implies \beta = \frac{2l}{d}$$

We calculate $genome\_inter\_cross_p$ using $path\_inter\_cross_v$ in the same way we calculated $genome\_intra\_cross_p$ using $path\_intra\_cross_v$. For the current path, we count the number of inter read DCs crossing over v. Since a repeat region is unlikely to

span both DCnodes of an inter read distance constraint, $path\_inter\_cross_v$ is expected to accurately estimate $genome\_inter\_cross_p$ in most cases. Therefore,

$$cov_p = \beta \times path\_inter\_cross_v$$
$$\implies cov_p = \frac{2l}{d} \times path\_inter\_cross_v$$

Therefore, every time our traversal visits a node v, we calculate the coverage using the method described and add it to the $usage_v$ variable. When the $usage_v$ variable becomes approximately equal to f_v, we deduce that the node v is saturated and thus, prevent our traversal algorithm from visiting it anymore.

While the model described above is expected to work reasonably well, given the complex nature of the problem the model can fail to accurately predict the coverage under certain situations. Predicting a value that is higher than the correct value (over-prediction) in the coverage in a genomic region could result in traversing the corresponding path less than ideal number of times (under-traversal). Predicting a lower coverage value than the correct (under-prediction) on the other hand could lead to more than ideal number of traversals through that path (over-traversal). To accommodate for that, we err on the side of under-predicting the coverage, thereby potentially over-traversing certain paths in the de-Bruijn graph. The reason for this is that an over-traversal of de-Bruijn graph paths would result in overlapping contigs that can be merged or removed from the assembly at a later stage (explained further in section 4.3). An under-traversal on the other hand results in a loss of information (in this case, potential loss of a contig representing a genomic region) that has no hopes of being recovered.

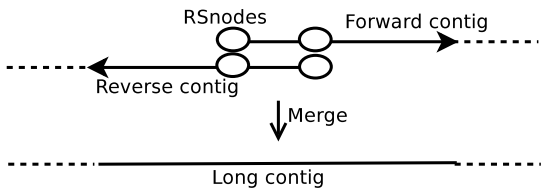

Figure 4.14: Merging forward and reverse contigs

4.2.9 Reverse traversal

Once a contig $\mathcal{C}$ is produced, we produce its *reverse contig* $\hat{\mathcal{C}}$ in the manner we describe below. For ease of discussion, we refer to $\mathcal{C}$ as *forward contig*.

We select a few beginning nodes of the forward contig, such that the sub-contig formed by these nodes is longer than the expected length of the longest repeat. These nodes are referred to as *reverse seed nodes (RSnodes)*. The RSnodes are then again set on to Q, but with their order and directions opposite to the one in which they were set in forward contig. We also push the distance constraints for these nodes in Q, and then begin the traversal.

The purpose of doing this is to extend the forward contig in reverse direction, and then merge the forward and reverse contigs to produce one long contig, as illustrated in figure 4.14.

4.2.10 Terminating graph traversal

We terminate the current path when one of the following situations occur:

a. When all the candidate nodes for further extension of current path are saturated.

b. When the traversal fails to achieve the necessary minimum score difference between the best and the second best path scores, when extended up to a maximum length of a threshold l_{max}.

After producing a forward contig and its reverse contig, we merge the two. We start a new contig with the longest node that is not saturated, and continue producing

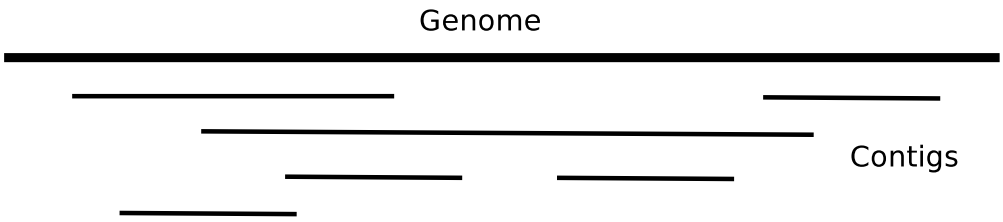

Figure 4.15: Overlapping contigs produced by our assembly

contigs until all the nodes in the graph become saturated, at which point we terminate the traversal.

4.3 Merging overlapping contigs

After the serial traversal algorithm described in section 4.2 (or parallel traversal algorithm described in chapter 5) execute, the contigs produced can have long stretches of overlapping regions, as shown in figure 4.15. This is due to the fact that we underestimate our coverage during traversal leading to over-traversing of paths in the de-Bruijn graph, as described in section 4.2.8.

These overlaps could be in form of suffix-prefix overlaps between contigs or smaller contigs contained in larger ones. In order to produce long and non-redundant contigs, we would like to identify pairs of contigs that share large overlapping regions. Using this, we can remove the contigs contained in others and merge contigs with suffix-prefix overlap into one.

Mashmap [40] is a tool that can be used to identify local alignment between long DNA fragments, approximately and quickly. Given an overlapping length threshold l and an identity cutoff t, is uses Jaccard similarity to identify all overlapping regions between two sequences of length greater than l and identity greater than t. After obtaining the contigs, we use Mashmap to identify all pairs of contigs sharing a good overlap by running an all-pair Mashmap alignment (all-pair comparison in this case is feasible as the number of contigs is small and mashmap is a fast and approximate comparison tool).

CHAPTER 5
PARALLEL GRAPH TRAVERSAL

In the previous chapter, we described a serial graph traversal algorithm to produce long paths in the de-Bruijn graph representing contigs. The algorithm extends paths iteratively, taking paired read distance constraints into consideration for adding a node to currently produced path. Termination of any path was followed by beginning a new path construction. The serial algorithm works well for assembling small genomes, like those of bacteria and virus where the genome sizes are of the order of hundred thousand basepairs to few million basepairs. The de-Bruijn graph for such datasets have a few hundred thousand nodes and the runtime for the serial algorithm is of the order of hours. However, for large genomes where the genome sizes are of the order of billions of basepairs respectively, the de-Bruijn graph can contain hundreds of millions of nodes. The serial traversal on such graphs would take days or weeks of runtime.

In this chapter, we describe a parallel graph traversal algorithm to produce contigs from such large graphs. The algorithm is shared memory parallel and is adapted from the serial algorithm described in previous chapter. The key challenge in designing an efficient parallel algorithm is that the algorithm discussed in the previous chapter is inherently sequential. Specifically, at any given decision point, the choice of next node is dependent on nodes that precede it. Hence independent tasks that can be executed in parallel are difficult to identify. In the following sections, we discuss how we address this challenge.

5.1 Partitioning the graph

The basic idea of our parallel algorithm is to initiate multiple serial traversals in the graph simultaneously. Each traversal can produce a contig independent of other

traversals, thereby producing multiple contigs simultaneously. A major challenge in executing this idea is to address conflicts where two or more traversals want to visit the same node simultaneously. To prevent such conflicts, as a pre step to run our parallel algorithm using p threads (denoted by $\{t_0, t_1, \ldots, t_{p-1}\}$), we partition the nodes of the de-Bruijn graph into p parts while optimizing two criteria:

- Minimizing the number of edges crossing over partitions (i.e. minimizing the *edge-cut*). The rationale for this criteria will become clear at a later stage.
- Balancing the number of nodes assigned to each partition.

There is a plethora of graph partitioners available that partition graph while satisfying the above two criteria reasonably well. For our work, we use KaHIP partitioner [41]. The set $\{P_0, P_1, \ldots, P_{p-1}\}$ represents the partitions thus obtained in the de-Bruijn graph.

5.2 Parallel graph traversal

The parallel graph traversal algorithm is designed on the principle of assigning disjoint sets of nodes to threads such that each thread operates only on the nodes assigned to it. Therefore, we assign the nodes in partition P_i to thread t_i. This leads to a clean division of work between threads and avoids two threads operating on the same node simultaneously.

To begin, each thread t_i starts a serial graph traversal in the subgraph corresponding to partition P_i as described in the previous chapter. Therefore p simultaneous traversals are started in the graph. Balancing the number of nodes in different partitions leads to balanced work loads amongst threads. KaHIP partitioner is consistently able to achieve a load balance ratio (ratio of maximum partition size to average partition size) of under 1.05 for all datasets in our experiments. Figure 5.1 shows four simultaneous traversals starting in a graph with four partitions. Since all the traversals start in

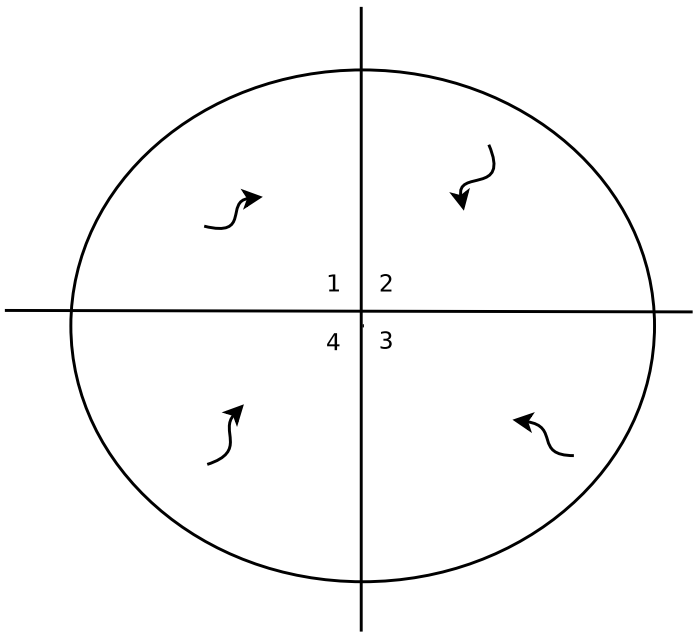

Figure 5.1: Four simultaneous traversals in the partitioned graph

separate partitions, they can proceed without conflicts as long as they do not cross partition boundaries.

When a traversal is about to cross a partition boundary, it needs special handling as described in the following subsection.

5.2.1 Contig Extension Jobs

To adhere to our design principle, a thread t_i can keep extending a contig $\mathcal{C}$ only as long as the traversal remains within the partition P_i. When a traversal crosses into a different partition, say P_j, we prohibit t_i to extend $\mathcal{C}$. This is done in order to prevent threads from interfering with each other. However, it would be incorrect to stop this traversal here since this would lead to shortening of contigs when in principle they could be extended further.

To handle this, threads create *contig extension jobs* for other threads when a contig needs to be extended beyond a partition boundary. In the above example, t_i will create a job for t_j. This job contains all the information that t_i would have needed to

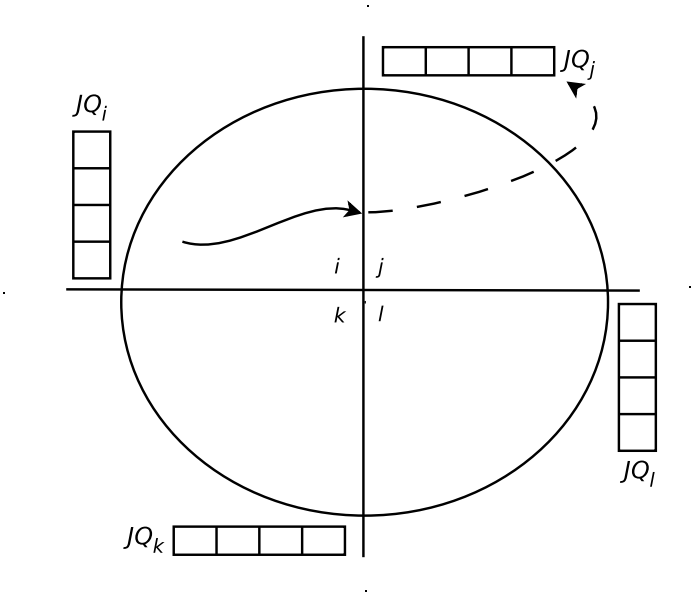

Figure 5.2: t_i pushing a job in JQ_j, later to be pulled by t_j to extend the contig

extend $\mathcal{C}$ had we not prohibited it from crossing partition boundary of P_i. Instead, now this information will be used by t_j to extend this contig.

Each thread t_r maintains a *job-queue* (abbreviated as JQ_r), a queue in which other threads can push jobs and t_r will serially pull jobs in *first in first out* order to extend contigs (shown in figure 5.2). Note that this pushing of jobs can be done in a cascading manner. Thus in the above example, thread t_i can push a job into JQ_j even if the contig $\mathcal{C}$ did not start in P_i and was instead extended by t_i using a job from JQ_i, pushed by another thread. With this design, simultanious traversals can take place in the graph without traversals interfering with each other.

Note that transferring a traversal from one partition to another is an expensive process, and although unavoidable, we would like to minimize it as much as possible. This is the reason our graph partitioning strategy had minimizing the *edge cut* (percentage of edges crossing partition boundaries) as one of the criteria. KaHIP partitioner is consistently able to achieve edge cut of under 0.5% for all datasets we used in our experiments.

Algorithm 3 Parallel Graph traversal algorithm.

```
1: while true do
2:    if no start node in any partition then
3:       break;
4:    end if
5:    if no job in my queue then
6:       select_start_node()
7:    else
8:       pull_job()
9:    end if
10:   while true do
11:      select_node_direction()
12:      update_candidate_nodes()
13:      update_distance_constraints()
14:      select_next_node()
15:      if no node selected then
16:         break;
17:      end if
18:      if diff par node selected then
19:         push job to diff queue
20:         break;
21:      end if
22:   end while
23: end while=0
```

5.2.2 Termination of parallel algorithm

Analogous to the serial algorithm termination, the parallel traversal algorithm terminates when all nodes in all partitions have been saturated and all ongoing traversals have finished.

CHAPTER 6
EVALUATION OF QUALITY AND PERFORMANCE OF OUR ASSEMBLY ALGORITHM

We implemented our graph traversal algorithm in C++. The shared memory parallelization was done using OpenMP threads. Our key contribution in this work is two fold: 1) a novel algorithm to traverse the graph with embedded distance constraints to recover a larger fraction of genome through high quality contigs than otherwise possible, and 2) a parallelization via simultaneous serial traversal in the partitioned de-Bruijn graph. The algorithm is intended to be used after constructing the de Bruijn graph, performing error correction on the graph to mitigate the unwarranted effects of read errors, and compressing chains in the graph to generate a more compact graph before subjecting it to traversal. To accomplish these tasks, we use Bruno [42], a fast de-Bruijn graph construction tool that performs error correction by filtering out low frequency k-mers prior to constructing the de-Bruijn graph. In addition, it also performs chain compaction, i.e. compressing a series of nodes in an unambiguous path into a single node.

6.1 Datasets

Our goal is to test the performance of our graph traversal algorithm against the traversal/contig generation algorithms developed by others. To conduct the testing, we used real read datasets sequenced from yeast, arabidopsis thaliana (abbreviated as A.Thaliana) and rice genomes. These datasets were downloaded from the SRA public database [43]. In addition, we tested our algorithm on a single library (Library 1) of the human chromosome 14 read dataset (abbreviated as H.Chr14) used in GAGE assembly evaluation [3]. To keep the testing comprehensive, we used a variety of

Table 6.1: Datasets used in our experiments.

	D_y	D_h	D_a	D_r
Organism	**Yeast**	**H.Chr14**	**A.Thaliana**	**Rice**
Reference length	12M	107M	135M	430M
No. Reads	$2 \times 4.3M$	$2 \times 18.3M$	$2 \times 26.5M$	$2 \times 126.1M$
Read len (bp)	126	101	300	150
Coverage	89X	34X	117X	88X
Sequencer	HiSeq 2000	-	MiSeq	HiSeq X Ten
SRA accn	SRR4244871	-	SRR5499434	SRR5011847

datasets in terms of genome length, read length and sequencer. The datasets are highlighted in table 6.1.

6.2 Evaluating quality

To assess the quality of the assembly, we compared our results with several other state of the art assemblers – ABySS 2.0, SAGE, SOAPdenovo2, IDBA, HipMer and SPAdes. As mentioned in the introduction, most assemblers use paired end information in a secondary phase to extend or scaffold initial contigs. SPAdes and AllPaths-LG are the only other currently available assemblers that directly use paired end information in graph traversal while producing contigs, and both use approaches that are distinct and different from ours. The AllPaths-LG assembler is targeted to sequencing protocols used at the Broad Institute, and requires input generated by particular sequencing protocols – for example, high coverage (>100 X), at least two paired-end libraries, the shorter library having small insert sizes that will induce overlap between the paired reads, etc. Due to these constraints, we were unable to include AllPaths-LG in our comparison.

Many assemblers (including some in our comparison, namely ABySS 2.0, SOAP-Denovo2 and HipMer) require k, the k-mer size as input. As described in chapter 3, IDBA uses multiple values of k to build iterative de-Bruijn graph, and does not need k as input. SPAdes uses multiple values of k based on k-mer frequency histogram for

different k values. In the interest of a fair comparison, we used the same k value as input for assemblers that require it. This value was selected by picking the largest value of k that SPAdes uses to perform its assembly. The SAGE assembler is not a de-Bruijn graph based assembler and instead uses string graphs. It therefore requires an overlap length threshold between two reads that is considered "good" for reads to be considered for merging. While it is unclear how to define an analogous of k in this model, we keep this overlap threshold slightly above k for SAGE. If an assembler is unable to produce the assembly within 72 hrs, we say that the assembler "timed out" and we are unable to report any results for that run.

We used the QUAST assembly evaluation tool [44] to evaluate all assemblers tested. QUAST evaluates assembled contigs by aligning each of them to the genome and providing statistics on how much of the contig aligns to the genome, along with affiliated statistics such as errors and the number of unreported nucleotides (designated by N's in the assembled contigs/scaffolds). We report the longest alignment found by QUAST for all assemblers for each dataset. We also report NGA50, which measures the maximum threshold length at which all aligned blocks extracted from raw contigs that are longer span more than 50% of the genome. In addition, we report the percentage length of contigs that is unaligned to the genome (UL), the number of mismatches (MM), the number of indel errors and the number of N characters inserted in the assembly. The mismatches, indels and $N's$ are reported per 100 kbp of contig length. Lastly we report the percentage of genome recovered (GR) by each assembly through their assembly. Table 6.2 shows a comparison of the different assembly methods on the yeast dataset. Due to miscellaneous factors, we were unable to obtain results for HipMer for this dataset. The largest k selected by SPAdes was 55 bp, and we used that for all assemblers that need k as input. For SAGE, we used an overlap threshold of 60 bp.

SPAdes was able to generate the longest alignment and highest NGA50, underscor-

Table 6.2: Comparison of various assemblers on **yeast** dataset

Assembler	*Longest align*	*NGA50*	*UL (%)*	*MM*	*Indels*	*N's*	*GR (%)*
ABySS	79.3K	23.5K	0.1	8.65	1.27	12.09	93.58
SAGE	87.6K	19.1K	0.2	11.08	2.03	220.06	95.0
SOAPdenovo2	138.1K	32.6K	0.2	4.06	11.83	130.37	93.99
SPAdes	153.2K	42.7K	0.1	6.29	2.63	16.69	93.98
IDBA	109.2K	27.4K	0.1	6.99	1.35	0	94.1
Our algorithm	81.5K	15.5K	0.1	20.53	2.06	0	93.87

Table 6.3: Comparison of various assemblers on **H.Chr14** dataset

Assembler	*Longest align*	*NGA50*	*UL (%)*	*MM*	*Indels*	*N's*	*GR (%)*
ABySS	75.6K	4.2K	0.04	87.09	14.42	28.11	75.5
SAGE	45K	3K	0.07	93.97	12.64	468.5	74.82
SOAPdenovo2	112.9K	7.3K	1.08	93.83	60.82	498.79	77.28
SPAdes	108.9K	9.7K	0.07	111.1	21.42	15.2	78.1
IDBA	67.6K	7.4K	0.07	99.1	17.65	0	79.06
Our algorithm	17.7K	2.3K	0.6	220.56	20.55	0	75.6

ing the value that can be gained by direct inclusion of paired reads distance constraints in assembly. While SOAPDenovo scores well in these measures as well, it does so at the cost of a large number of N's and high indel rate. While we had a high mismatch rate, our algorithm measured comparably with other assemblers under most metrics. The algorithm, along with IDBA were able to recover a competitive fraction of genome without the aid of N characters in the assembly. Apart from a few measures, there is little that separates different assemblers in this case. For the dataset D_h, we again used $k = 55$ and the results are shown in table 6.3. IDBA was able to recover the highest fraction of genome, demonstrating the merit of using multiple k values for assembly. However, this comes at the cost of algorithm complexity, resulting in IDBA being unable to finish on large datasets.

We perform similar evaluations for the datasets D_a and D_r. The k values chosen for these datasets are 127 bp and 77 bp respectively. Due to large sizes of these

Table 6.4: Comparison of various assemblers on **arabidopsis thaliana** dataset

Assembler	*Longest align*	*NGA50*	*UL (%)*	*MM*	*Indels*	*N's*	*GR (%)*
ABySS 2.0	589K	78K	3.5	12.52	5.27	52.43	96.45
SPAdes	554K	110K	49	20.89	10.34	1.71	96.02
HipMer	352K	48.4K	0.9	9.83	3.55	115.79	94.7
Our algorithm	110K	12.4K	1.1	11.48	2.42	0	97.17

Table 6.5: Comparison of various assemblers on **rice** dataset

Assembler	*Longest align*	*NGA50*	*UL (%)*	*MM*	*Indels*	*N's*	*GR (%)*
ABySS 2.0	163K	16K	9.3	22.07	7.23	145.19	76.73
SOAPDenovo	292K	26K	28.3	63.22	42.65	6.7K	73.34
SPAdes	168K	16.6K	38.1	30.43	6.01	53.54	77.74
HipMer	263K	24.7K	4.3	17.52	7.30	98.90	89.01
Our algorithm	144K	22K	4.0	145.5	15.09	0	89.33

datasets, we were unable to run IDBA and SAGE on these datasets. The overlap threshold chosen for SAGE in these cases were 130 and 80 respectively (though in order to possibly reduce the runtime, we experimented with lower threshold lengths as well). In addition, we were unable to run SOAPDenovo on D_a as it does not accept datasets with read length more than 256 bp. The QUAST evaluations of successful runs are highlighted in tables 6.4 and 6.5.

For the dataset D_a, although we fall short in terms of the longest alignment and NGA50, we were able to recover the largest percentage of genome with no N's in the assembly. Along with HipMer, we made close to lowest errors (low values for UL, MM, and *indels*). While SPAdes was able to achieve fairly long alignment and NGA50, it does so at the cost of large percentage of unaligned length of contigs.

For dataset D_r, apart from higher mismatches, our algorithm is competitive on other metrics. We are again able to recover the largest percentage of genome with no N's (with HipMer being close second). The other three assemblers are able to recover far less percentage of genome. In addition, SPAdes and SOAPDenovo produce a large

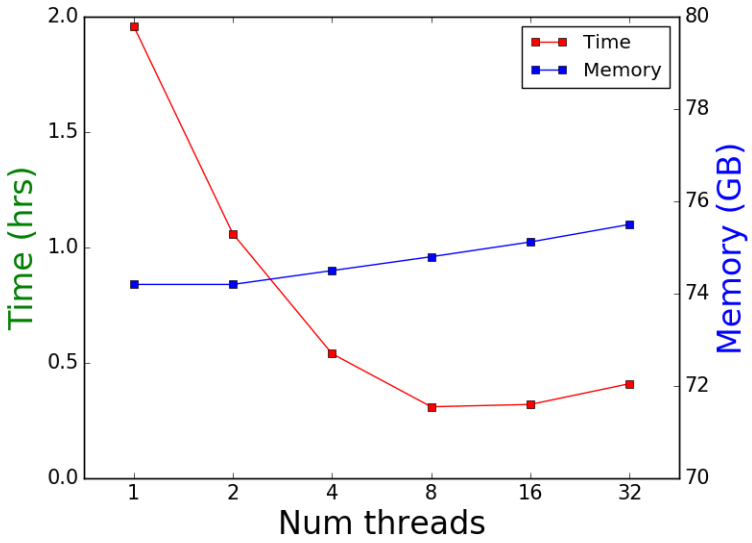

Figure 6.1: Scalability on rice

number of unaligned contigs, making it difficult for any further processing of their output.

6.3 Evaluating scalability

While computing performance enhancement is not the main objective of this work, we provide runtime evaluation of our algorithm for completeness. We do this to demonstrate that our algorithm can run in reasonable runtime. We conducted all our experiments on a shared memory compute node with 1 TB main memory, 4 sockets, 18 cores per socket, each core being an Intel(R) Xeon(R) CPU E7-8870 v3 @ 2.10GHz. All except two of the assemblers in our comparison list (ABySS 2.0 and HipMer) are either serial or shared memory parallel, incapable of taking any advantage of a distributed memory platform. We note that while ABySS 2.0 and HipMer are distributed memory parallel, running them on shared memory platform posed no disadvantage as their execution did not terminate due to runtime shortage.

Figure 6.1 shows the scalability of our algorithm on rice dataset as we vary the number of threads.

Our algorithm achieves respectable runtime scalability up until 8 threads, at which point it takes 19 mins to finish on rice dataset. Beyond 8 threads, the runtime flattens. This is due to the memory intensive nature of our algorithm where different threads are trying to access a shared resource (main memory) in order to extend their respective paths in the de-Bruijn graph. As the number of such threads increase, it increases the contention, thereby preventing the scaling beyond a point.

The main memory usage of our algorithm effectively stays constant ($\sim 75\ GB$) as we vary the number of threads, implying that memory usage for the algorithm is scalable. This is a useful feature as it indicates that the threads need a minimal thread specific bookkeeping. In addition, memory scalability makes this algorithm a good candidate for a distributed memory implementation.

6.4 Discussion

In this work, we address the problem of producing a high quality genome assembly using paired reads. With a research effort spanning over three decades, several assembly tools have been developed to tackle this complex problem. A majority of tools developed after the arrival of NGS machines use de-Bruijn graphs, a compact graph structure, to capture overlaps between input reads. Most of these tools produce an initial set of contigs corresponding to unambiguous sections of this graph. They use the paired read information at a later stage to close gaps between initial contigs, or fill the gaps with N characters indicative of a genome region that cannot be recovered by the assembly. Through experimental results, we are able to demonstrate that by traversing the de-Bruijn graph beyond the chain nodes, our algorithm is able to recover higher fraction of genome compared to other assembly algorithms for large and complex datasets. Our objective in this research is to extend contigs beyond the linear chains in the de-Bruijn graphs. Instead of filling the assembly with unknown N characters, we aim to discover these unknown regions by first embedding the

paired read information directly onto the de-Bruijn graphs and then using them as pointers to extend contigs when faced with an unclear forward path. Since we produce contigs only corresponding to paths in the graph without filling N characters, on several occasions our contigs are terminated due to disconnected regions in the graph. This is reflected in relatively short contigs produced by our algorithm. However a higher genome recovered without any N characters in our assembly indicates that our algorithm can be useful in discovering regions in the genome that correspond to highly bifurcated sections, requiring pointers to make decisions about extending contigs. We experimentally show that our algorithm is able to run in a reasonable runtime for large datasets, demonstrating its utility to assemble large genomes.

CHAPTER 7
PROBABILISTIC ESTIMATION OF OVERLAP GRAPHS FOR LARGE SEQUENCE DATASETS

In this chapter we describe a parallel algorithm to estimate overlap graphs. Sequence overlap graphs, constructed based on suffix-prefix relationships between pairs of sequences, are an important data structure in computational biology. As discussed in chapter 1, high throughput sequencers can read several million to a few billion DNA fragments in a single experiment. In chapter 2 we discussed that constructing overlap graphs for such datasets is compute-intensive. In this chapter, we present a Locality-Sensitive Hashing based parallel heuristic algorithm to construct overlap graphs for large genomic datasets. With reasonable assumptions on the characteristics of input sequences, we establish probabilistic bounds on the quality of the overlap graphs so produced. We demonstrate the validity and efficiency of our approach by comparing against true overlap graphs using datasets derived from small (*E. coli*) and large (*H. sapiens*) genomes.

7.1 Introduction

As discussed in chapter 2, given a set of reads generated by a sequencer, a *sequence overlap graph* consists of vertices corresponding to reads and edges corresponding to suffix-prefix overlaps among them. This data structure plays a central role in overlap-layout-consensus based genome assembly [45, 46, 13], and metagenomic assembly [47]. Widely used sequencers like Illumina HiSeq and TenX can sequence up to six billion reads in a single experiment that costs only a few thousand dollars. Constructing overlap graphs for such large datasets is compute-intensive and requires efficient algorithms.

Let n denote the number of reads and l denote the read length. In the absence of sequencing errors, a read overlap would be an exact match. Due to errors, one must contend with substitutions, insertions, and deletions, their types and frequency being a characteristic of the sequencing technology. For any error model, a suffix-prefix overlap can be determined by dynamic programming based alignment algorithms, which are part of standard literature. These algorithms take quadratic $O(l^2)$ time. Thus, the overlap graph can be accurately constructed by running the alignment algorithm on every pair of reads for a total of $O(n^2l^2)$ run-time. This is prohibitive for large n, and such direct computation is rarely used even prior to the era of high throughput sequencing. Besides, the number of edges in an overlap graph is typically significantly smaller than the maximum possible, and is often linear or near linear in n.

To compute an overlap graph in practice, the typical approach is to design a heuristic that first identifies a subset of all possible pairs of reads, termed *candidate pairs* from hereon, for subjecting them to rigorous suffix-prefix alignment. Such a heuristic algorithm cannot directly inspect each pair, as that would incur $\Omega(n^2)$ time. Instead, it must directly generate pairs of reads with potential for good suffix-prefix overlap. Heuristics are typically based on a simpler relationship between a pair of sequences that must be satisfied if a good suffix-prefix overlap exists. An overwhelming favorite is the existence of a common k-mer [15, 48, 14, 46]. Another measure is the existence of a maximal common substring of length $\geq k$ for a specified threshold k [16, 17]. We contend that the frequency of shared k-mers between a pair of reads is a better indicator of the existence of long suffix-prefix overlaps. However, computing the set of k-mers in each read, and comparing the corresponding sets for each pair, is computationally infeasible.

Locality-Sensitive-Hashing (LSH) is a widely used technique for comparing high dimensional data. Popular applications include cryptography and analyzing social networks. Berlin *et al.* created assembly of human genome using PacBio reads based

on LSH and MinHash technique [49]. We propose an LSH based parallel algorithm to compute pairs of reads that have a high probability of sharing an edge in the overlap graph. Our algorithm is based on the widely used minhash technique [50], originally invented for document clustering. We provide probabilistic bounds on the false negative rates for the edges missed by our algorithm, when compared to true overlap graphs. Our algorithm can be viewed as a filtering technique to reduce the number of pairs of reads to be considered for a rigorous evaluation of their suffix-prefix overlap using quadratic dynamic programming based sequence alignment algorithms. Thus, the false positives generated by our algorithm have implications on run-time but pose no quality issues, as they can be successfully detected and removed using alignment algorithms. We experimentally show that the number of false positives increases the run-time only to within a small constant factor. More importantly, we present a way to tune algorithmic parameters to achieve desired precision and sensitivity. We demonstrate the quality, performance, and scalability of our parallel algorithm using large short read datasets derived from *E. coli* and *H. sapiens* genomes.

7.2 The Proposed Algorithm

Let $R = \{r_1, r_2, \ldots r_n\}$ be the set of input reads. Without loss of generality, we assume all the reads have length l. For a given read r, we use $r\,[p, q]$ to denote the substring of r from position p to q, inclusive of both ends. A suffix-prefix overlap is a sufficiently long match between the suffix of a read r_i and the corresponding prefix of another read r_j. The length is important to avoid spurious overlaps that do not correspond to genomic co-location. As the alphabet size is 4, for a genome of length m, there must be repeated occurrences of sequences of length $< \log_4 m$, hence overlaps at this size or smaller are not sufficient.

Our algorithm is based on computing signature representatives of reads. Similarity between such signatures for two reads is a good proxy for similarity between the

Algorithm 4 Finding candidate pairs.

0: Let $\Psi = \emptyset$
0: **for** $i = 1 \ldots$B **do**
0: Compute $H_{ij}(r), \forall r \in R, \forall j \in \{1, \ldots, T\}$
0: Let $V = \langle K = (H_{i1}(r), \ldots H_{iT}(r)), r \rangle, r \in R$
0: $Sort(V,\ key = K)$
0: **for each** segment S of V that share the same K **do**
0: $\Psi = \Psi \cup \{\langle u.r, v.r \rangle \mid u \in S, v \in S, u.r \neq v.r\}$
0: **end for**
0: **end for**=0

actual reads, shown by Broder *et al.* [51]. In section 7.4, we use properties of minhash signatures to estimate the quality of our proposed algorithm. We use the following notation.

- *k-spectrum*: The k-spectrum of a read r, denoted as $\kappa(r)$, is the set of all k length substrings of read r, for some constant $k < l$.

$$\kappa(r) = \{r\,[i, i+k-1] \mid 1 \leq i \leq (l-k+1)\}$$

- *Minhash*: Given a read r and a hash function h, the *minhash* of the read r, denoted as $M(h, r)$ is the minimum hash value obtained when the h is applied to each element in the $\kappa(r)$.

$$M(h, r) = min\,\{h(k_i) \mid k_i \in \kappa(r)\}$$

To account for double stranded nature of DNA sequences, minhash for a read r is calculated using k-mers in $\kappa(r)$ and their reverse complements. Our algorithm is aimed at finding pairs of reads that share a sufficient number of minhash signatures for a family of hash functions. To compute such pairs, we propose an LSH based technique for comparing minhash signatures of reads. Specifically, we use the banding technique of LSH, as described below.

Consider a family of hash functions Γ with $N = B \times T$ hash functions, divided into B bands, each band containing T hash functions. Formally:

$$\Gamma = \{h_{ij} \mid i \in [1, B], j \in [1, T]\}$$

We use Γ to calculate the minhash signatures for all the reads. Given a read r and a hash function h_{ij}, we define H_{ij} as follows:

$$H_{ij}(r) = M(h_{ij}, r)$$

i.e.,$H_{ij}(r)$ is the minhash signature of read r using the hash function h_{ij}. We compute $H_{ij}(r)$ for all $h_{ij} \in \Gamma$ and $r \in R$, resulting in $B \times T$ signatures for each read r.

We need an effective way to compare signatures for different reads such that comparison between signatures closely emulates the comparison between the actual reads. Let Ψ denote the set of all candidate pairs selected by our algorithm. A read pair (r_y, r_z) is included in the set Ψ if the following holds true:

$$\exists i \in [1, B] \mid \forall j \in [1, T], H_{ij}(r_y) = H_{ij}(r_z)$$

In other words, we consider the pair (r_y, r_z) as a candidate pair if there exists at least one band such that all the minhash signatures for r_y and r_z within that band are identical. We show the effectiveness of such comparison between minhash signatures in section 7.4.

Algorithm 4 describes our approach to compute the set of candidate pairs Ψ. First, within a band, minhash signatures are computed for all the reads using the functions in Γ. Then, using sorting, buckets of reads that share identical minhash signature tuples are identified. Each valid pair corresponding to a bucket is added to the set Ψ. This is repeated for all the B bands.

7.3 Parallelization strategy

Our algorithm permits easy parallelization that extends to multiple nodes and not limited to cores of a shared memory system, useful when dealing with large-scale high throughput sequencing data. For ease of presentation, we use the term *processor* to refer to an independent unit executing a thread of the parallel algorithm, such as a CPU core.

After loading $\frac{n}{p}$ reads into its memory, each processor generates for each read r_x assigned to it, a tuple of length $T+1$, $\langle x, H_{i,1}(r_x), \ldots, H_{i,T}(r_x)\rangle$ corresponding to a band b_i. We then use parallel sort to partition the generated tuples into buckets, such that all the tuples assigned to a bucket share identical values for the T signatures. If a bucket crosses a processor boundary, it is shifted to the lower ranked processor. Finally, the candidate pairs are generated from each bucket within a processor. We repeat these steps for all bands $b_1, \ldots, b_B$. Hence, the total parallel runtime is $O\left(\frac{BTnl}{p}\right) + O(B \times \mathsf{pSort}(nT, p))$, where $\mathsf{pSort}(m, p)$ is the time for parallel sorting of m numbers using p processors.

We implemented our algorithm in C++ and using MPI for the collective communication operations. In our implementation, we use the KmerInd library [52] to load the file data in blocks of size $\approx |R|l/p$ and to generate all the k-mers. For parallel sorting, we use the implementation of distributed sample sort described in [53]. For all the $T \times B$ hash functions, h_{ij}, we choose MurMurHash with different seed values. MurMurHash functions are known to behave min-wise independently and this property is useful in deriving probabilistic bounds discussed in the next section. We use the implementation of SMHasher library [54] for computing MurMurHash values for k-mers generated from the reads.

7.4 Theoretical quality assessment

In this section, we derive a probabilistic bound on the *false negative rate (FNR)* that our algorithm incurs. False negatives (FN) correspond to pairs of reads with acceptable suffix-prefix overlap that our algorithm failed to identify, and thus will be missing from our constructed graph. Therefore we would like to keep FNR as low as possible. We derive this bound in *two stages* as follows. In the *first* stage, we evaluate the set of pairs of reads emitted by our algorithm against reads with a high Jaccard similarity coefficient between them, and estimate FNR for that comparison. In the *second* stage, we relate the Jaccard similarity to suffix-prefix overlaps and use the FNR estimated for Jaccard similarity to estimate errors with respect to suffix-prefix overlaps.

7.4.1 Relating to Jaccard similarity

We estimate the quality of our algorithm in computing the set of pairs that have a Jaccard similarity coefficient above a threshold J_{min}. The quality assessment of the algorithm relies on the assumption that the family of hash functions Γ is min-wise independent [51]. Thus, the following holds true for any hash function $h \in \Gamma$ and reads $r, s \in R$

$$P(H(r) = H(s)) = Jac(r, s)$$

where $P(E)$ denotes the probability of an event E occurring, and $Jac(r, s)$ represents the Jaccard similarity index between $\kappa(r)$ and $\kappa(s)$, denoted by J_{rs} from hereon.

Let χ_{rs} denote the event that read pair (r, s) is selected as a candidate pair by our algorithm (recall B = number of bands and T = number of hash functions per band). Therefore:

$$P(\chi_{rs}) = 1 - \left(1 - J_{rs}^{T}\right)^{B}$$

Define the event π_{rs} as true if $J_{rs} \geq J_{min}$, and false otherwise. Then, $FNR_J =$

$P(\neg\chi_{rs}/\pi_{rs})$. The subscript J denotes that these errors are defined with respect to Jaccard similarity. Consider

$$\begin{aligned}
FNR_J &= P(\neg\chi_{rs}/\pi_{rs}) \\
&= P(\neg\chi_{rs}/J_{rs} \geq J_{min}) \\
&= \frac{P(\neg\chi_{rs} \cap (J_{rs} \geq J_{min}))}{P(J_{rs} \geq J_{min})} \\
&= \frac{\sum_{J_i \geq J_{min}} P(\neg\chi_{rs} \cap (J_{rs} = J_i))}{\sum_{J_i \geq J_{min}} P(J_{rs} = J_i)} \\
&= \frac{\sum_{J_i \geq J_{min}} P(\neg\chi_{rs}/J_{rs} = J_i).P(J_{rs} = J_i)}{\sum_{J_i \geq J_{min}} P(J_{rs} = J_i)} \\
&= \frac{\sum_{J_i \geq J_{min}} \left(1 - J_i^T\right)^B .P(J_{rs} = J_i)}{\sum_{J_i \geq J_{min}} P(J_{rs} = J_i)}
\end{aligned}$$

Let η_{rs} be the number of common k-mers between reads r and s. We define function g that gives the Jaccard similarity coefficient corresponding to the number of k-mers common between two reads.

$$J_{rs} = g\left(\eta_{rs}\right) = \frac{\eta_{rs}}{2\left(l - k + 1\right) - \eta_{rs}}$$

$$\text{and} \quad \eta_{rs} = g^{-1}(J_{rs})$$

Where $\eta_{rs} \in [1, l - k + 1]$. Hence we have $l - k + 1$ different values for J_{rs}. Assuming all distinct values occur with uniform probability, and setting $\eta_{min} = g^{-1}(J_{min})$, the false negative rate is

$$FNR_J = \frac{\sum_{J_i \geq J_{min}} \left(1 - J_i^T\right)^B}{l - k + 1 - g^{-1}(J_{min})} \tag{7.1}$$

Therefore, FNR_J is framed as a function of parameters l, J_{min}, k, B and T. Given the length of the reads l and a threshold J_{min} on Jaccard similarity coefficient, the parameters k, B, T can be tuned to achieve a desirable bound for the false negative

rate. This is a key advantage of our approach.

7.4.2 Suffix-prefix overlaps

We now adapt the FNR_J estimated in the previous subsection to the suffix-prefix overlap scenario. Let l_{min} denote the minimum length of an acceptable suffix-prefix overlap. For descriptive purposes, we assume the overlap to be an exact match. It can be easily extended to non-exact matches for reads with low error rates (true for our datasets) by estimating number of k-mer mismatches in a suffix-prefix overlap due to errors.

Let l_{rs} be the length of the longest suffix-prefix overlap that reads r and s have. Assuming the overlap is the only commonality between the strings, r and s share $l_{rs} - k + 1$ common k-mers and therefore the Jaccard coefficient J_{rs} between r and s is given by

$$J_{rs} = g(l_{rs} - k + 1)$$

We relate l_{min} and J_{min} as follows. Let $l_{min} = g^{-1}(J_{min}) + k - 1$. That is, l_{min} is the minimum length of suffix-prefix overlap between r and s so as to have Jaccard similarity coefficient of J_{min} between them. Also, let ϕ_{rs} denote the event that $l_{rs} \geq l_{min}$. Clearly, $\phi_{rs} \implies \pi_{rs}$.

Figure 7.1 shows the set of all pairs of reads with subsets in which events χ_{rs}, π_{rs} and ϕ_{rs} are true.

The false negative rate in calculating the suffix-prefix overlap is given by

$$FNR_O = P(\neg\chi_{rs}/\phi_{rs})$$

Given that the Jaccard similarity coefficient is independent of the position of

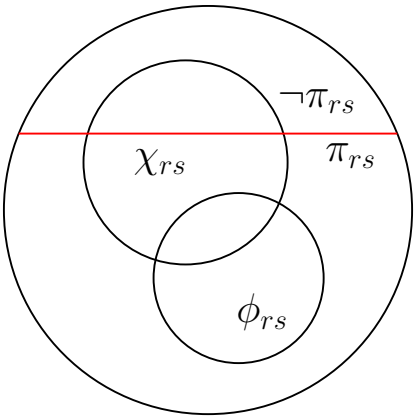

Figure 7.1: Set of all read pairs, partitioned by subsets corresponding to predicates namely, π_{rs}: Set of pairs (r, s) with $J_{rs} \geq J_{min}$, χ_{rs}: Set of pairs selected by our algorithm, ϕ_{rs}: Set of pairs (r, s) with $l_{rs} \geq l_{min}$.

overlap within the reads, we assume:

$$P(\chi_{rs} \cap \phi_{rs}/\chi_{rs} \cap \pi_{rs}) = P(\phi_{rs}/\pi_{rs})$$

That is, the pairs selected by our algorithm that have Jaccard similarity above J_{min} $(\chi_{rs} \cap \pi_{rs})$ are uniformly distributed in the spectrum of π_{rs}, and the likelihood of any such pair having a suffix-prefix overlap above l_{min} is equal to $P(\phi_{rs}/\pi_{rs})$. Using similar reasoning, we have the same assumption for the event $\neg\chi_{rs}$:

$$P(\neg\chi_{rs} \cap \phi_{rs}/\neg\chi_{rs} \cap \pi_{rs}) = P(\phi_{rs}/\pi_{rs})$$

Therefore conclude the following from Figure 1.

$$FN_O = P(\phi_{rs}/\pi_{rs}) \times FN_J$$

$$TP_O = P(\phi_{rs}/\pi_{rs}) \times TP_J$$

$$FP_O = FP_J + TP_J - P(\phi_{rs}/\pi_{rs}) \times TP_J$$

$$TN_O = TN_J + FN_J - P(\phi_{rs}/\pi_{rs}) \times FN_J$$

The variables *FN, TP, FP, TN* denote False Negatives, True Positives, False Positives and True Negatives respectively, and subscripts O and J denote the parameters are with respect to suffix-prefix overlap and Jaccard similarity index, respectively. Therefore

$$\begin{aligned} FNR_O &= \frac{FN_O}{FN_O + TP_O} \\ &= \frac{P(\phi_{rs}/\pi_{rs}) \times FN_J}{P(\phi_{rs}/\pi_{rs}) \times FN_J + P(\phi_{rs}/\pi_{rs}) \times TP_J} \\ &= \frac{FN_J}{FN_J + TP_J} \\ &= FNR_J \end{aligned} \tag{7.2}$$

Hence, FNR_J calculated with respect to Jaccard similarity computation is expected to give an accurate estimate of the desired error estimate for FNR_O.

7.5 Experimental Results

We ran our experiments on an Intel Xeon Infiniband cluster. Each node has two 2.0 GHz 8-core Intel E5-2650 processors and 128GB of main memory. Experiments were conducted on up to 64 nodes, totaling 1,024 cores. We evaluated our algorithm on three different datasets, shown in Table 7.1. Each dataset is a set of simulated Illumina reads derived from a known genome using SimSeq [55], a read simulator designed to simulate Illumina short reads while taking into account the sequencer specific error models.The reason for using simulated reads instead of using a true Illumina dataset is to be able to know true suffix-prefix overlaps without running an alignment algorithm for every pair of reads, which would not be feasible computationally. We use $D1$ and $D2$ to demonstrate the quality and scalability of our algorithm. Using $D3$, we demonstrate the ability of our algorithm to handle big genomic datasets. All the reads are of length 100.

To the best of our knowledge, no current software specifically targeting overlap graphs can estimate the graph for datasets with billions of short reads. However, in addition to standalone performance evaluation of our method, we also compared it against Minimap [56], a method that uses the MinHash technique but designed for the different problem of mapping and comparing long erroneous reads produced by PacBio and Oxford Nanopore Technologies sequencers. A recent survey [57] showed that among the currently available methods for finding the pairs of sequences with sufficient suffix-prefix overlap, Minimap performs better compared to other methods.

7.5.1 Quality Assessment

Let F_e denote the estimated/predicted FNR using equations 7.1 and 7.2 described in section 7.4. We compare this with the experimentally observed FNR value (called F_o). In addition, we experimentally calculate the ratio $\frac{FP}{TP}$, denoted by Ω. This ratio is a measure of the number of incorrect candidate pairs selected by our filtering algorithm. Note that Ω has no effect on the quality of the overlap graph produced, since the incorrect pairs will be eliminated by running an alignment algorithm on the pairs selected. The ratio Ω is the ratio of the time wasted by our algorithm to the ideal run-time (when $FP = 0$). If for example $\Omega = 1$, the total run-time is increased by a factor of 2 when compared to the ideal.

To evaluate the quality of our algorithm for dataset $D1$, we used fixed values of $T = 3$ and $B = 334$ while varying k (15, 13, 11) and l_{min} (60, 50, 40). Similar procedure was used for $D2$. Table 7.2 shows the F_e, F_o and Ω values. We compute

Table 7.1: Datasets used in our experiments.

	D1	D2	D3
Organism	*E. coli*	*H. sapiens*	*H. sapiens*
Source	Genome	*chr1*	Genome
No. Reads	1.75×10^6	87.5×10^6	1.25×10^9
Coverage	37.7X	35.4X	38.64X

Table 7.2: Estimated FNR (F_e), observed FNR (F_o) and Ω values for datasets $D1$ and $D2$ with our proposed method. Observed FNR and Ω with Minimap are also listed.

	Dataset $D1$				Dataset $D2$			
	Param. k,T,B	Overlap Threshold (l_{min})			Param. k,T,B	Overlap Threshold (l_{min})		
		60	50	40		60	50	40
F_e	15,3,334	0.0018	0.0401	0.1511	19,3,200	0.0558	0.1840	0.3146
F_o		0.0171	0.0392	0.0824		0.1482	0.1914	0.2549
Ω		0.7968	0.4668	0.2779		3.6302	2.8925	2.5146
F_e	13,3,334	0.0003	0.0173	0.1011	15,3,300	0.0028	0.0471	0.1609
F_o		0.0092	0.0248	0.0609		0.1049	0.1309	0.1769
Ω		0.8618	0.5097	0.3043		7.1090	5.6551	4.8542
F_e	11,3,334	0.0001	0.0006	0.0606	17,2,300	0.0001	0.0014	0.0410
F_o		0.0040	0.0139	0.0417		0.0909	0.1001	0.1185
Ω		1.1094	0.7002	0.4586		10.2986	8.1076	6.7351
F_o and Ω with Minimap								
F_o	$k=9$	0.2295	0.1737	0.1377	$k=13$	0.4313	0.3646	0.3141
Ω		0.1047	0.1129	0.1288		0.2340	0.3258	0.5078
F_o	$k=11$	0.2486	0.1882	0.1492	$k=15$	0.4566	0.3888	0.3372
Ω		0.1003	0.1061	0.1120		0.1091	0.1289	0.1591

the true set of pairs that have a suffix-prefix overlap greater than l_{min} using SimSeq metadata, and compare them against the set of pairs generated by our algorithm to calculate F_o and Ω.

We observe that F_e and F_o are reasonably close in most of the cases for $D1$, indicating that assumptions made while estimating the errors are reasonable. The estimates are particularly accurate for $(k, l_{min}) = (15, 50), (13, 50), (11, 40)$ indicating a need for k and l_{min} to be proportional to each other for better estimates. For reasonable estimates, the value k needs to be balanced between the two extremes. A very small value of k could lead to too many spurious matches, whereas a large k can cause a single error in the overlap region to miss any common representatives between otherwise similar reads. For dataset $D2$, we typically underestimate the FNR by at

most 0.1. In all the cases, Ω, which indicates the extra computing cost, is a small constant, implying that our filter does not add enormous amount of extra overhead.

Minimap uses the minimum hash values of minimizers to compute the candidate overlapping pairs of sequences, a lower k value in general tends to favor lower observed FNR values. Therefore for Minimap experiments, we typically select the k-mer size lower than that used for evaluation of our method. Results of these runs are listed in Table 7.2. For both the datasets, our proposed method shows much lower FNR values in all the cases compared to Minimap. While Minimap shows lower Ω values compared to the proposed method, it does so at the cost of missing a significant percentage of the true suffix-prefix overlapping pairs of sequences.

7.5.2 Evaluating Scalability

To measure the scalability of our implementation, we ran it on datasets $D1$ and $D2$, while varying the number of cores. Table 7.3 shows the scalability results of our implementation.

Table 7.3: Runtime and memory results for datasets $D1$ and $D2$

No. of. Cores	Total Runtime (s)	Relative Speedup	Max. Mem. per core (MB)
Dataset $D1$			
16	648.86	1.00X	183.45
32	328.56	1.97X	111.29
64	169.49	3.82X	69.56
128	121.31	5.34X	46.97
Dataset $D2$			
64	2874.36	1.00X	2950.77
128	1810.17	1.59X	1525.93
256	908.33	3.16X	782.89
512	528.61	5.43X	433.62

Results show that the run-time and the maximum per core memory scale up

to 128 cores for $D1$ and up to 512 cores for $D2$, in line with their relative sizes. For any parallel algorithm, if the data size is fixed and the number of processors is continually increased, diminishing returns set in at some juncture. In our algorithm, this limit transpires due to two factors. First, the pair generation for a bucket takes time proportional to the square of the size of the bucket, and therefore, there is an imbalance in the amount of work done by the processors when the partitioning is too fine-grained. Second, our algorithm relies on parallel sorting, and communication costs dominate when the per processor data size becomes too small.

To test the applicability of our parallel algorithm for large datasets, we used dataset $D3$ containing 1.25 billion reads. We were able to process $D3$ in $\approx$58 minutes using 1024 cores. This demonstrates that our implementation is able to process big genomic datasets in reasonable time.

For running Minimap, we used a machine with four 2.1 GHz 18-core Intel Xeon E7-8870 processors and 1TB of main memory. We used this large shared memory machine for Minimap runs because Minimap cannot take advantage of distributed memory but is only capable of utilizing cores available in a single machine via shared-memory threads. Though runtimes on machines with different capabilities are not directly comparable, we provide the runtimes for Minimap for the sake of completeness. Using 32 threads, Minimap took $\sim$0.9 and $\sim$37 minutes for datasets D1 and D2 respectively. For D3, even after consuming 24 hours of its allocated job time, Minimap failed to complete its run with 32 threads.

7.6 Discussion

We address the problem of computing overlap graphs for very large sequence datasets. Prior works use a fixed substring or maximal longest common substring based heuristic to restrict the number of pairs of reads which are subjected to suffix-prefix overlap tests using alignment algorithms. However, they incur too many false positives as a single

substring match may not extend to a significant suffix-prefix overlap. Besides, none of the earlier methods provide rigorous guarantees on the edges they miss. We propose a more meaningful indicator of a potential suffix-prefix overlap by finding common k-mers in the k-mer spectra of the corresponding reads. Our strategy uses minhash based techniques to directly generate candidate pairs that share common signatures without inspecting each potential pair. We established probabilistic bounds on the False Negatives generated by our algorithm, with experimental results conforming to theory. We experimentally demonstrate that the additional computational overhead associated with the False Positives generated by the algorithm is within a small constant factor. The proposed algorithm, and its parallelization, enable construction of much larger overlap graphs than previously feasible.

CHAPTER 8
CONCLUSION AND FUTURE WORK

Genome assembly has been an important and challenging problem in the field of computational biology for multiple decades. Its applications are manifold and significant efforts have been dedicated to creating high quality assembly methods. In this project, we presented a novel technique to improve de-novo genome assembly quality. We used de-Bruijn graphs, a popular framework in the field, to develop our method. The problem of producing long and accurate contigs is formulated as the problem of traversing this graph accurately. Paired-end sequencing is a technology in which sequencers from companies like Illumina produce reads in pairs such that the two reads in each pair originate from nearby genomic locations with a known approximate distance between them. This information can be useful in resolving repeats and elongating contigs as one read in a pair can anchor the other to extend contigs that would have been produced without this information. Most other assemblers, after creating the de-Bruijn graph, traverse the unambiguous sections of the graph to produce initial set of contigs and use the paired end information at a later stage by mapping the reads back to these initial contigs. We propose a technique to perform graph traversal beyond the unambiguous sections of the graph. This is done by using the paired read information at bifurcation points in the graph. Specifically, we introduce inter read and intra read distance constraints between k-mers obtained from the readset and embedded these constraints in the de-Bruijn graph. These distance constraints are then used as cues to traverse through forks in the graph by scoring all the options and picking the best one, resulting in extension of paths beyond unambiguous regions and thereby recovering large fraction of genome. On encountering two similarly strong paths to extend contigs, we explore all paths from

the junction up to a certain distance to decide the correct path to take. In addition, we present a technique to detect and traverse bubbles, a structure that can frequently occur in assembly graphs. We present a model to estimate the coverage at the genomic position in a non-uniform coverage read dataset. This helps us in determining the termination point of the current path, and of the entire traversal. While most other assemblers produce sequences corresponding to unitig nodes as initial contigs, and use paired-end information in a separate stage for extending the contigs and scaffolding, our approach embeds the paired-end information directly into the de-Bruijn graph, thus collectively considering all the information present in the read set, and producing contigs in light of this information.

In order to enable us assemble large genomes, we adapt this traversal algorithm to develop a shared memory parallel strategy. This is done by first partitioning the graph and initiating a serial traversal in each partition. Extension of contigs beyond partition boundary is handled by developing a mechanism where threads can create contig extension jobs for other threads to be processed later. The serial algorithm is implemented using C++ and parallelization is achieved using OpenMP threads. Preliminary results presented in this work demonstrate that our approach has the potential to produce high quality contigs.

In addition, we propose and algorithm to estimate overlap graphs for large sequence datasets. Constructing such graphs became computationally challenging after arrival of high throughput sequencers. We propose a locality sensitive hashing based technique to identify potential suffix-prefix overlaps between reads. Our strategy directly generates candidate pairs that share common signatures without inspecting each potential pair. We establish theoretical estimates on the False Negatives generated by our algorithm, and experimentally show that computational overhead due to the False Positives generated by the algorithm is within a small constant factor. The proposed algorithm is parallelized on distributed memory architectures using MPI and enables construction

of much larger overlap graphs than previously feasible.

This research can be extended in several directions:

- *Scoring function*: While we have experimented with several scoring functions, the fact that this part (discussed in subsection 4.2.5) is the most important factor in determining assembly quality makes it worthy of more effort for its improvement and is an interesting direction for further research.

- *Parallelization*: While our current assembly algorithm is shared memory parallel and runs in reasonable time on datasets with genomes sized a few hundred million basepairs, shared memory parallel algorithms do not usually scale well. This will especially be the case here since our algorithm is memory intensive. In order to use it to assemble larger genomes like human (genome size: 3 billion bp) and Pine (genome size: 22 billion bp), distributed memory parallelization would be needed in order to achieve a reasonable runtime. Our current parallelization strategy can be implemented as a distributed algorithm where different partitions can be distributed across processors and contig extension jobs can be implemented through messages in case of MPI implementation.

- *Scaffolding*: While our current algorithm traverses the de-Bruijn graph to produce contigs, it is limited by the connectivity of the graph even when there is paired read information present. Specifically, if there is a distance constraint from the current node to a target, but a path between these two nodes is absent, the traversal stops. The algorithm can be extended to "jump" from the current node to target, filling the absent path with N characters. This can be thought of as adapting the current traversal algorithm to perform contig generation and scaffolding in a single stage.

- *Assembly using overlap graphs*: The overlap graph construction described in this work enables construction of graphs for much larger datasets than previously

feasible. This opens a new research direction in terms of extending this work to create assembly for short reads using overlap graphs.

REFERENCES

[1] D. Earl, K. Bradnam, *et al.*, "Assemblathon 1: a competitive assessment of de novo short read assembly methods," *Genome research*, vol. 21, no. 12, pp. 2224–2241, 2011.

[2] K. R. Bradnam, J. N. Fass, *et al.*, "Assemblathon 2: evaluating de novo methods of genome assembly in three vertebrate species," *GigaScience*, vol. 2, no. 1, pp. 1–31, 2013.

[3] S. L. Salzberg, A. M. Phillippy, A. Zimin, D. Puiu, T. Magoc, S. Koren, T. J. Treangen, M. C. Schatz, A. L. Delcher, M. Roberts, *et al.*, "GAGE: a critical evaluation of genome assemblies and assembly algorithms," *Genome research*, vol. 22, no. 3, pp. 557–567, 2012.

[4] E. M. P Medvedev K Georgiou, "Computability and equivalence of models for sequence assembly," in *Workshop on Algorithms in Bioinformatics*.

[5] T. Gingeras, J. Milazzo, D. Sciaky, and R. Roberts, "Computer programs for the assembly of dna sequences," *Nucleic Acids Research*, vol. 7, no. 2, pp. 529–543, 1979.

[6] J. T. Simpson, K. Wong, *et al.*, "ABySS: a parallel assembler for short read sequence data," *Genome research*, vol. 19, no. 6, pp. 1117–1123, 2009.

[7] E. B. D.R.Zerbino, "Velvet: algorithms for de novo short read assembly using de bruijn graphs," in *Genome Research*.

[8] J. Butler, I. MacCallum, *et al.*, "ALLPATHS: de novo assembly of whole-genome shotgun microreads," *Genome research*, vol. 18, no. 5, pp. 810–820, 2008.

[9] B. D. Reinhardt JA *et al.*, "De novo assembly using low-coverage short read sequence data from the rice pathogen pseudomonas syringae pv. oryzae," *Genome Research*, vol. 19, no. 1, pp. 294–305, 2009.

[10] J. L. Weirather, M. de Cesare, Y. Wang, P. Piazza, V. Sebastiano, X.-J. Wang, D. Buck, and K. F. Au, "Comprehensive comparison of pacific biosciences and oxford nanopore technologies and their applications to transcriptome analysis," *F1000Research*, vol. 6, 2017.

[11] H. Lu, F. Giordano, and Z. Ning, "Oxford nanopore minion sequencing and genome assembly," *Genomics, proteomics & bioinformatics*, vol. 14, no. 5, pp. 265–279, 2016.

[12] E. W. Myers, "The fragment assembly string graph," *Bioinformatics*, vol. 21, no. suppl 2, pp. ii79–ii85, 2005.

[13] E. W. Myers, G. G. Sutton, A. L. Delcher, I. M. Dew, D. P. Fasulo, M. J. Flanigan, S. A. Kravitz, C. M. Mobarry, K. H. Reinert, K. A. Remington, *et al.*, "A whole-genome assembly of drosophila," *Science*, vol. 287, no. 5461, pp. 2196–2204, 2000.

[14] S. Batzoglou, D. B. Jaffe, K. Stanley, J. Butler, S. Gnerre, E. Mauceli, B. Berger, J. P. Mesirov, and E. S. Lander, "Arachne: a whole-genome shotgun assembler," *Genome research*, vol. 12, no. 1, pp. 177–189, 2002.

[15] R. L. Warren, G. G. Sutton, S. J. Jones, and R. A. Holt, "Assembling millions of short dna sequences using ssake," *Bioinformatics*, vol. 23, no. 4, pp. 500–501, 2007.

[16] D. Hernandez, P. François, L. Farinelli, M. Østerås, and J. Schrenzel, "De novo bacterial genome sequencing: millions of very short reads assembled on a desktop computer," *Genome research*, vol. 18, no. 5, pp. 802–809, 2008.

[17] X. Huang and A. Madan, "Cap3: a dna sequence assembly program," *Genome research*, vol. 9, no. 9, pp. 868–877, 1999.

[18] R. M. Idury and M. S. Waterman, "A new algorithm for DNA sequence assembly," *Journal of computational biology*, vol. 2, no. 2, pp. 291–306, 1995.

[19] P. A. Pevzner, H. Tang, *et al.*, "An eulerian path approach to DNA fragment assembly," *Proceedings of the National Academy of Sciences*, vol. 98, no. 17, pp. 9748–9753, 2001.

[20] S. Gnerre, I. MacCallum, *et al.*, "High-quality draft assemblies of mammalian genomes from massively parallel sequence data," *Proceedings of the National Academy of Sciences*, vol. 108, no. 4, pp. 1513–1518, 2011.

[21] R. Li, H. Zhu, *et al.*, "De novo assembly of human genomes with massively parallel short read sequencing," *Genome research*, vol. 20, no. 2, pp. 265–272, 2010.

[22] R. Luo, B. Liu, *et al.*, "SOAPdenovo2: an empirically improved memory-efficient short-read de novo assembler," *GigaScience*, vol. 1, no. 1, pp. 1–6, 2012.

[23] J. A. Chapman, I. Ho, *et al.*, "Meraculous: de novo genome assembly with short paired-end reads," *PloS one*, vol. 6, no. 8, e23501, 2011.

[24] J. Meng, B. Wang, Y. Wei, S. Feng, and P. Balaji, "Swap-assembler: scalable and efficient genome assembly towards thousands of cores," *BMC bioinformatics*, vol. 15, no. Suppl 9, S2, 2014.

[25] Y. Peng, H. C. Leung, *et al.*, "IDBA–a practical iterative de bruijn graph de novo assembler," in *Annual International Conference on Research in Computational Molecular Biology*, 2010, pp. 426–440.

[26] ——, "IDBA-UD: a de novo assembler for single-cell and metagenomic sequencing data with highly uneven depth," *Bioinformatics*, vol. 28, no. 11, pp. 1420–1428, 2012.

[27] J. T. Simpson and R. Durbin, "Efficient de novo assembly of large genomes using compressed data structures," *Genome research*, vol. 22, no. 3, pp. 549–556, 2012.

[28] L. Ilie, B. Haider, M. Molnar, and R. Solis-Oba, "Sage: string-overlap assembly of genomes," *BMC bioinformatics*, vol. 15, no. 1, p. 1, 2014.

[29] P. Medvedev and M. Brudno, "Maximum likelihood genome assembly," *Journal of computational Biology*, vol. 16, no. 8, pp. 1101–1116, 2009.

[30] A. V. Zimin, G. Marçais, D. Puiu, M. Roberts, S. L. Salzberg, and J. A. Yorke, "The masurca genome assembler," *Bioinformatics*, vol. 29, no. 21, pp. 2669–2677, 2013.

[31] N. I. Weisenfeld, S. Yin, T. Sharpe, B. Lau, R. Hegarty, L. Holmes, B. Sogoloff, D. Tabbaa, L. Williams, C. Russ, *et al.*, "Comprehensive variation discovery in single human genomes," *Nature genetics*, vol. 46, no. 12, p. 1350, 2014.

[32] S. D. Jackman, B. P. Vandervalk, H. Mohamadi, J. Chu, S. Yeo, S. A. Hammond, G. Jahesh, H. Khan, L. Coombe, R. L. Warren, *et al.*, "Abyss 2.0: resource-efficient assembly of large genomes using a bloom filter," *Genome research*, vol. 27, no. 5, pp. 768–777, 2017.

[33] E. Georganas, A. Buluç, J. Chapman, S. Hofmeyr, C. Aluru, R. Egan, L. Oliker, D. Rokhsar, and K. Yelick, "Hipmer: an extreme-scale de novo genome assembler," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ACM, 2015, p. 14.

[34] S. Koren, B. P. Walenz, K. Berlin, J. R. Miller, N. H. Bergman, and A. M. Phillippy, "Canu: scalable and accurate long-read assembly via adaptive k-mer

weighting and repeat separation," *Genome research*, vol. 27, no. 5, pp. 722–736, 2017.

[35] G. M. Kamath, I. Shomorony, F. Xia, T. A. Courtade, and N. T. David, "Hinge: long-read assembly achieves optimal repeat resolution," *Genome research*, vol. 27, no. 5, pp. 747–756, 2017.

[36] B. G. Jackson, M. Regennitter, *et al.*, "Parallel de novo assembly of large genomes from high-throughput short reads," *International Parallel and Distributed Processing Symposium*, 2010.

[37] P. Medvedev, S. Pham, *et al.*, "Paired de bruijn graphs: a novel approach for incorporating mate pair information into genome assemblers," *Journal of Computational Biology*, vol. 18, no. 11, pp. 1625–1634, 2011.

[38] A. Bankevich, S. Nurk, *et al.*, "SPAdes: a new genome assembly algorithm and its applications to single-cell sequencing," *Journal of Computational Biology*, vol. 19, no. 5, pp. 455–477, 2012.

[39] S. K. Pham, D. Antipov, *et al.*, "Pathset graphs: a novel approach for comprehensive utilization of paired reads in genome assembly," *Journal of Computational Biology*, vol. 20, no. 4, pp. 359–371, 2013.

[40] C. Jain, S. Koren, A. Dilthey, A. M. Phillippy, and S. Aluru, "A fast adaptive algorithm for computing whole-genome homology maps," *Bioinformatics*, vol. 34, no. 17, pp. i748–i756, 2018.

[41] P. Sanders and C. Schulz, "Think Locally, Act Globally: Highly Balanced Graph Partitioning," in *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, ser. LNCS, vol. 7933, Springer, 2013, pp. 164–175.

[42] T. Pan, R. Nihalani, and S. Aluru, "Fast de bruijn graph compaction in distributed memory environments," *IEEE/ACM transactions on computational biology and bioinformatics*, 2018.

[43] *SRA*, https://www.ncbi.nlm.nih.gov/sra.

[44] A. Gurevich, V. Saveliev, *et al.*, "QUAST: quality assessment tool for genome assemblies," *Bioinformatics*, btt086, 2013.

[45] M. Egholm, M. Margulies, *et al.*, "Genome sequencing in open microfabricated high density picoliter reactors," *Nature*, vol. 437, pp. 376–380, 2005.

[46] X. Huang and S.-P. Yang, "Generating a Genome Assembly with PCAP," in *Current Protocols in Bioinformatics*. Wiley Online Library, 2002.

[47] B. Haider, T.-H. Ahn, *et al.*, "Omega:an overlap-graph de novo assembler for metagenomics," *Bioinformatics*, vol. 30, no. 19, pp. 2717–2722, 2014.

[48] J. Dohm, C. Lottaz, *et al.*, "SHARCGS, a fast and highly accurate short-read assembly algorithm for de novo genomic sequencing," *Genome research*, vol. 17, no. 11, pp. 1697–1706, 2007.

[49] K. Berlin, S. Koren, *et al.*, "Assembling large genomes with single-molecule sequencing and locality-sensitive hashing," *Nature biotechnology*, vol. 33, no. 6, pp. 623–630, 2015.

[50] A. Broder, S. Glassman, *et al.*, "Syntactic clustering of the web," *Computer Networks and ISDN Systems*, vol. 29, no. 8, pp. 1157–1166, 1997.

[51] A. Broder, M. Charikar, *et al.*, "Min-wise independent permutations," in *Proceedings of STOC*, 1998.

[52] T. Pan, P. Flick, *et al.*, "Kmerind: A flexible parallel library for k-mer indexing of biological sequences on distributed memory systems," in *Proceedings of the ACM-BCB*, 2016.

[53] P. Flick and S. Aluru, "Parallel distributed memory construction of suffix and longest common prefix arrays," in *Proceedings of SC'15*, 2015.

[54] A. Appleby, *SMHasher*, https://github.com/aappleby/smhasher, 2016.

[55] J. S. John, *SimSeq*, https://github.com/jstjohn/SimSeq, 2010.

[56] H. Li, "Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences," *Bioinformatics*, vol. 32, no. 14, pp. 2103–2110, 2016.

[57] J. Chu, H. Mohamadi, *et al.*, "Overlapping long sequence reads: current innovations and challenges in developing sensitive, specific and scalable algorithms," *Bioinformatics*, vol. 33, no. 8, pp. 1261–1270, 2017.

VITA

Rahul Nihalani is a computer scientist who works in the area of high performance computing and computational biology. Rahul was born in India and obtained his Bachelor's degree in communication and computer engineering from The LNM Institute of Information Technology. He later attended the Indian Institute of Technology Bombay to obtain his master's degree in computer science. He joined Dr. Srinivas Aluru's group as a Ph.D. student at Iowa State University and later moved to Georgia Tech where he obtained his Ph.D. in computational science and engineering.