December 2012

A new algorithm for de novo genome assembly

Md. Bahlul Haider
The University of Western Ontario

Supervisor
Dr. Lucian Ilie
The University of Western Ontario

Joint Supervisor
Dr. Roberto Solis-Oba
The University of Western Ontario

Graduate Program in Computer Science

A thesis submitted in partial fulfillment of the requirements for the degree in Doctor of Philosophy

A NEW ALGORITHM FOR *DE NOVO* GENOME ASSEMBLY

(Spine title: A new algorithm for *de novo* genome assembly)

(Thesis format: Monograph)

by

Md. Bahlul Haider

Graduate Program in Computer Science

A thesis submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

The School of Graduate and Postdoctoral Studies

The University of Western Ontario

London, Ontario, Canada

THE UNIVERSITY OF WESTERN ONTARIO

School of Graduate and Postdoctoral Studies

CERTIFICATE OF EXAMINATION

Supervisor 1:

.....
Dr. Lucian Ilie

Supervisor 2:

.....
Dr. Roberto Solis-Oba

Examiners:

.....
Dr. Bin Ma

.....
Dr. Jagath Samarabandu

.....
Dr. Kamran Sedig

.....
Dr. Kaizhong Zhang

The thesis by

Md. Bahlul Haider

entitled:

A new algorithm for *de novo* genome assembly

is accepted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

.....
Date

.....
Chair of the Thesis Examination Board

Abstract

The enormous amount of short reads produced by next generation sequencing (NGS) techniques such as Roche/454, Illumina/Solexa and SOLiD sequencing opened the possibility of *de novo* genome assembly. Some of the *de novo* genome assemblers (e.g., Edena, SGA) use an overlap graph approach to assemble a genome, while others (e.g., ABySS and SOAPdenovo) use a de Bruijn graph approach. Currently, the approaches based on the de Bruijn graph are the most successful, yet their performance is far from being able to assemble entire genomic sequences. We developed a new overlap graph based genome assembler called Paired-End Genome ASsembly Using Short-sequences (PEGASUS) for paired-end short reads produced by NGS techniques. PEGASUS uses a minimum cost network flow approach to predict the copy count of the input reads more precisely than other algorithms. With the help of accurate copy count and mate pair support, PEGASUS can accurately unscramble the paths in the overlap graph that correspond to DNA sequences. PEGASUS exhibits comparable and in many cases better performance than the leading genome assemblers.

Keywords: genome assembly, DNA, next generation sequencing, overlap graph, mate pair.

Acknowledgements

I would like to express my sincere gratitude to everyone involved in the completion of this dissertation. First of all, I would like to express my deep and sincere gratitude to my supervisors Dr. Lucian Ilie and Dr. Roberto Solis-Oba for introducing me to the beautiful research field on Bioinformatics. Their continuous support helped me do my research in Bioinformatics, which was a new field for me when I started my graduate studies at The University of Western Ontario. I really appreciate their extraordinary patience in reading and correcting my drafts including research proposals, presentation slides, scholarship applications and finally this dissertation. I owe to my supervisors for the valuable time they spent to listen to my ideas and guide me to find the right way.

I am also thankful to The University of Western Ontario for awarding me with the Western Graduate Research Scholarship (WGRS) and to the Department of Computer Science, at The University of Western Ontario for supporting me. I am also grateful to the Ministry of Training, Colleges and Universities (Ontario) for awarding me an Ontario Graduate Scholarship (OGS).

I am also thankful to my examiners Dr. Bin Ma, Dr. Jagath Samarabandu, Dr. Kamran Sedig and Dr. Kaizhong Zhang. I am thankful to Mike Molnar, who helped me to run some of the experiments. I would like to thank all my colleagues, friends and family members, especially my wife Saina Sharmin for making the difficult task easier by giving me advice, suggestion and inspiration.

Contents

List of Tables

List of Figures

List of Algorithms

List of Acronyms

bp Base pair

DNA Deoxyribonucleic acid

emPCR Emulsion PCR

HGP Human Genome Project

kbp Kilo base pairs

Mbp Mega base pairs

NGS Next Generation Sequencing

PCR Polymerase Chain Reaction

SA Suffix Array

SNP Single Nucleotide Polymorphism

WGS Whole Genome Shotgun

Chapter 1

Introduction

It has long been known that the biological properties of an individual are inherited from its parents, though the mechanism through which this is done was unknown until the late 20th century. We now know that the biological features of an individual are inherited from its parents when the chromosomes from both parents are fused together in an egg cell during fertilization. After fertilization, the cell multiplies at an exponential rate and each of the new cells gets an identical copy of the chromosomes. The chromosomes in a cell nucleolus contain sequences of nucleotides called *DNA* (Deoxyribonucleic acid) and the entire DNA contained in all the chromosomes is called a *genome*.

The DNA in the chromosomes contains the genetic code of an organism. Three consecutive nucleotides in the DNA strand act as a unit of genetic code called a *codon* and this is the building block of proteins. The sequence of nucleotides in a codon determines which amino acid is to be added during the protein synthesis process: This process is similar to the process of reading instructions from a sequential file by a computer and performing specific tasks based on the instructions. Proteins are sequences of amino acids and the order of the amino acids in protein synthesis is responsible for all biological features of an organism. Since an offspring inherits chromosomes from two individuals, it inherits genetic code from both of its parents.

1.1 Sequencing Techniques

The process of determining the sequence of nucleotides in a DNA fragment is called *sequencing*. It is not possible to read the entire sequence of a genome at once by using current technology, which can only sequence small DNA fragments consisting of a few hundred nucleotides. The sequencing process of a genome starts by first breaking multiple copies of the target genome into many small manageable pieces called *reads*. These short fragments are then individually sequenced by DNA sequencing machines. Sequencing techniques are broadly divided into two categories: Sanger sequencing and Next Generation Sequencing (NGS). The Sanger sequencing was the first sequencing technique, which produces relatively long reads. The Sanger method has low throughput and is an expensive process. On the other hand, inexpensive NGS techniques produce an enormous amount of short reads faster than the Sanger method; however, reads produced by NGS are shorter than the Sanger reads. All the NGS techniques achieve high throughput by simultaneously sequencing many DNA fragments.

1.2 Applications of DNA Sequencing

Knowing the DNA sequence of an organism has a large number of applications in fields as diverse as forensic science, medicine and agriculture. Identifying an individual by analyzing a DNA sample is called *DNA fingerprinting*. Nowadays it is common to identify criminals by matching DNA samples from blood, hair or anything that contains DNA collected from a crime scene. Recent improvements on DNA analysis have helped to solve old unsolved criminal cases, some going back several decades. DNA analysis is also used to identify paternity of children through parental testing and to identify health risks associated with an individual. Scientists are able to genetically modify plants to increase productivity and resistance against insects. These increasing numbers of applications have created a necessity for inexpensive technologies for DNA sequencing.

1.3 Genome Assemblers

A computer program called a *genome assembler* is used to try to find the original DNA sequence from the set of DNA fragments produced by the sequencing techniques. There are several computational challenges that need to be solved in genome assembly. First, DNA sequences might contain identical or nearly identical subsequences of nucleotides which are hard to assemble, in the same way that nearly identical pieces in a jigsaw puzzle make the problem of solving the puzzle harder. Second, sequencing techniques are never perfect; errors are introduced when the reads are sequenced. Third, the enormous number of reads typically produced by sequencing techniques makes the problem more complicated in the same way in which solving a jigsaw puzzle with many pieces is harder than solving a puzzle with few pieces.

Over the past decade several genome assemblers have been designed to discover the DNA sequence of an organism. Celera assembler [36] and Arachne [2] are two assemblers designed for assembling long reads. Edena [14], Velvet [53], ABySS [46], SOAPdenovo [22], ALLPATHS [4] and SGA [45] are genome assemblers specifically designed to deal with short reads produced by NGS. All existing NGS genome assemblers face the following challenges in genome assembly.

Error Handling: NGS datasets contain more errors than datasets produced by the Sanger method. Each one of the existing assemblers has its own way of detecting and correcting errors. However, it is usually not possible to correct all the errors from the datasets.

Memory: All the genome assemblers use some kind of overlapping technique to build a graph to represent the overlapping fragments of a genome. The underlying graph used by the assemblers is the data structure that typically requires the largest amount of memory in these programs.

Time: A significant amount of time in genome assembly is spent building the graph

to model the reads and their overlaps. Moreover, genome assemblers need to find paths in this graph that represent DNA subsequences. As the graph can be very dense, finding these paths is the most time consuming step for genome assembly.

Quality: Because of the time limitation, most genome assemblers use some heuristics to reduce the number of paths to be searched in the graph. Quality of assembled sequences of the assembler is affected by this. Moreover, estimating the multiplicities of the reads in the genome from which the reads were sequenced has a significant effect on the quality of sequences produced by the assemblers. Better estimation of the multiplicities of the reads can result in better and longer subsequences.

1.4 PEGASUS

To deal with the challenges of genome assembly, we have designed a genome assembler called Paired-End Genome ASsembler Using Short-sequences (PEGASUS). PEGASUS uses a correction program called RACER (Rapid and Accurate Correction of Errors in Reads) [17], which accurately corrects a significant fraction of the errors in the NGS datasets. PEGASUS uses several new techniques to reduce the size of the overlap graph, which in turn reduces the amount of memory used. Unlike other genome assemblers, PEGASUS reduces the overlap graph while building it which is memory efficient. PEGASUS estimates the multiplicities of the reads by computing a minimum cost flow in the overlap graph, where the flow through an edge corresponds to the multiplicity of the reads represented by the edge; we use a statistical analysis to determine the cost of the flow on the edges that reflect the likelihood of the reads represented by the edges appearing a certain number of times in the genome. Accurate estimates of the multiplicities of the edges help further reduce the overlap graph and produce longer subsequences of the genome. Mate pairs (set of pairs of reads) with known distance between them in the genome are used to merge subsequences into longer ones.

1.4.1 Challenges

During the development of PEGASUS we faced several challenges. In the first implementation of PEGASUS we found out that it was using too much memory, which prevented it from assembling large datasets. This was because we were building the whole overlap graph first and then removed redundant information from the graph to reduce its size. This approach worked well for small datasets, however, for large datasets the overlap graph used too much memory. We had to modify PEGASUS in such a way that it could remove the redundant information from the overlap graph as the graph was being built. To save even more space we had to modify the data structures that we used to store the overlap graph.

At first we were using a suffix array [23, 24] to find all the overlapping reads to insert edges in the graph. We found that a suffix array is not memory efficient for large datasets. To reduce the amount of memory used, we implemented an algorithm for computing read overlaps that uses a hash table. This saved a considerable amount of space.

1.4.2 Contributions

DNA assembly is a very active area of research and every year new assemblers are developed. PEGASUS is a new assembler; many of the ideas used in PEGASUS have been proposed by other researchers. But, we also have introduced some novel ideas:

- PEGASUS uses a new technique to reduce the overlap graph while building it. This saves memory as there is no need to ever store the entire overlap graph.
- PEGASUS uses a new technique to estimate the number of times that a read appears in the genome by combining a minimum cost network flow computation and an elaborated statistical analysis.
- PEGASUS performs additional graph reductions based on the above flow.

- PEGASUS explores all paths in the overlap graph to try to determine how pairs of reads are assembled in the genome.

1.5 Thesis Overview

In Chapter 2, we discuss different DNA sequencing techniques. Two generations of sequencing techniques are discussed: Sanger sequencing and next generation sequencing (NGS). Sequencing techniques first break the target DNA into a large number of short fragments. To determine the order of the nucleotides in each one of the fragments, every fragment is converted to some order of signals (e.g., color) by chemical reactions that reveal the order of nucleotides in the sequence and the order of the signals is recorded by a computer. Next generation sequencing techniques can simultaneously record the signals from millions of fragments, thus increasing the throughput. Applications, advantages and disadvantages of existing sequencing techniques are also discussed in this chapter.

Chapter 3 focuses on genome assembly for NGS datasets. Basic genome assembly terminology is explained in this chapter. Genome assemblers use graphs such as overlap graphs or de Bruijn graphs to assemble genomes. These graphs are explained in this chapter. We also discuss some of the techniques used by existing genome assemblers.

PEGASUS is explained in Chapter 4. We describe in details the algorithms implemented in PEGASUS. First the input dataset is corrected with RACER and then PEGASUS builds an overlap graph with the help of a hash table. The use of a hash table reduces the running time for discovering overlapping reads and thus for building the overlap graph. Most of the edges of the overlap graph are redundant, so PEGASUS removes these redundant edges while building the overlap graph to reduce the memory usage. We also explain the statistical analysis used in our minimum cost flow computation for estimating the multiplicities of the reads in the graph.

We have tested PEGASUS on several NGS datasets. In Chapter 5, we present the

results of these tests and compare them with the results from ABySS [46] and SOAPdenovo [22]. Assemblers are compared based on some parameters such as the longest contig, N80, N50, and N20. At the end of this chapter we also compare the running time and memory usage of the assemblers.

In Chapter 6, we conclude the thesis and give few future directions of PEGASUS. Genome assemblers are never perfect. There are always rooms for improvements. We propose some of the possible improvements that can be made to increase the performance of PEGASUS.

The software manual for PEGASUS can be found in Appendix A.

Chapter 2

DNA Sequencing

Since the discovery of the *double helix* (i.e., spiral structure of two strands of molecules) model for DNA by James D. Watson and Francis Crick [50] in 1953, scientists have been trying to understand the information stored in DNA. Genetic information in DNA is stored as a sequence of molecules called *nucleotides*. The process by which scientists determine the order of nucleotides in DNA fragments is called sequencing.

In 1977, for the first time a full DNA was sequenced [43]. This remarkable achievement was attained by Frederick Sanger and his team, who sequenced the DNA of *bacteriophage* $\Phi X174$, which is about 5 kb (kilo bases) in size. The DNA of several organisms was sequenced in the late 1970s. Frederick Sanger [43], and Allan Maxam and Walter Gilbert [27] proposed DNA sequencing techniques independently and as a result of their DNA sequencing research Frederick Sanger and Walter Gilbert received the Nobel Prize in Chemistry in 1980. Recent technological improvements on sequencing technique allow scientists to sequence more complex DNA; for example, the first draft of human genome was sequenced in 2000 [20, 48]. Different sequencing techniques are discussed in this chapter.

2.1 DNA

The genetic information of a living organism is stored within the chemical structure of its DNA (Deoxyribonucleic acid). A molecule of DNA consists of four nucleotides: *adenine*, *cytosine*, *guanine* and *thymine*, or in short `A`, `C`, `G` and `T`, respectively. These nucleotides in the DNA molecule are also known as *bases*. The order in which these four bases appear in a DNA molecule provides the instructions for making proteins. This order spells the genetic code and controls all biological functions of a living organism. It is very important to know the sequence or order of the bases in the DNA of an organism because it helps scientists understand the biological properties of the organism. For example, scientists can use the DNA sequence of an organism to identify and predict health risks. Hence, knowing the DNA sequence of an individual could help discover diseases long before they might be identified otherwise.

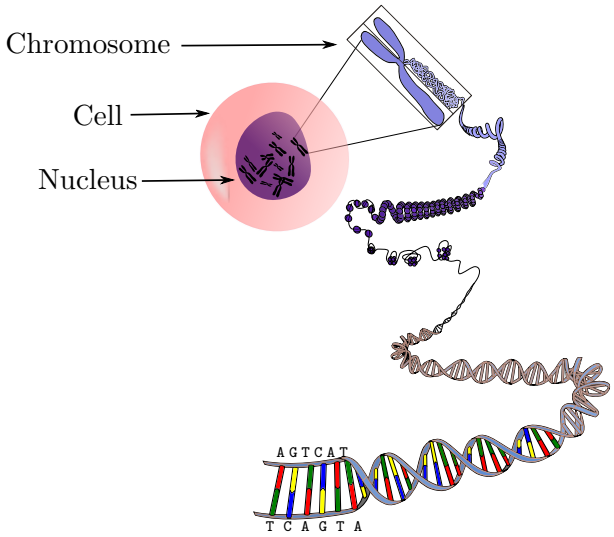


Figure 2.1: DNA inside a cell nucleus.

Almost all cells, except red blood cells, in an organism contain the same DNA, which is located in the chromosomes in the cell's nucleus as shown in Figure 2.1. The set of all chromosomes of an organism is called its genome. The human genome consists of 23 pairs of chromosomes and over 3 billion bases; more than 99% of these bases are ordered

in the exact same manner in all humans. The remaining 1% difference, which is about 30 million bases, makes an individual different from another.

The nucleotides in a DNA molecule are arranged in two long strands that form a spiral structure called a double helix as shown in Figure 2.2. Nucleotides from one strand pair up with nucleotides in the opposite strand. Adenine (`A`) pairs up with thymine (`T`) and cytosine (`C`) pairs up with guanine (`G`) to form a unit called a *base pair*, or in short *bp*. Nucleotides in a base pair are connected by hydrogen bonds. Adenine (`A`) and thymine (`T`) are connected by two hydrogen bonds, whereas cytosine (`C`) and guanine (`G`) are connected by three hydrogen bonds. The sequence of nucleotides in one strand is called the *complementary* of the other strand.

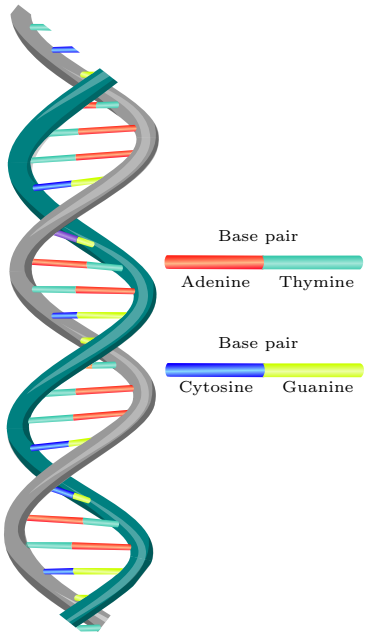


Figure 2.2: Double helix structure of DNA.

The backbone of a DNA strand is made from alternating phosphate and five-carbon sugar molecules. The third carbon atom of a sugar ring in the DNA backbone is connected to the fifth carbon atom of the next sugar ring by phosphate bonds between them. Since the bonds are asymmetric, there is a unique direction for the nucleotides in each strand of DNA. The direction of the nucleotides in one strand is opposite to the direction in the

other strand. The two ends of a DNA strand are called the 5′ (five prime) and 3′ (three prime) ends. The ends are named after the position of the carbon atoms in the bonds of the DNA backbone. The 5′-end of the DNA strand has the fifth carbon in the sugar ring, whereas the 3′-end of the DNA strand has the third carbon in the sugar ring.

Current DNA sequencing technology cannot sequence whole genomes, but only short DNA sequences (about 35 to 1000 base pairs) called reads. Often reads are generated in pairs, with known approximate distance (*insert size*) between them; one such a pair of reads is called a *mate pair* or *paired-end reads*. A set of reads sequenced from a DNA is called a *library*. A library containing paired-end reads is called a *paired-end library*. If the reads in a library are sequenced without any mate pairs then the library is called a *single-end library* (also called a *fragment library*) and the reads are called *single-end reads*. There are two, so called, generations of DNA sequencing techniques. In the first generation techniques, the sequencing was dominated by the Sanger method. The second generation of sequencing techniques, commonly known as next generation sequencing (NGS), use more sophisticated approaches to increase the throughput and reduce the cost of sequencing. DNA sequencing techniques are discussed in the following sections.

2.2 Sanger Sequencing

In 1975 Frederick Sanger developed a sequencing technology [43] known as *Sanger sequencing*. While this technology has been continuously improved over the past 30 years, it can only sequence about 500 to 1000 base pairs of DNA at a time. The process of producing reads by the Sanger method is very slow and expensive. The Sanger method dominated the world of genome sequencing for over two decades and led to a number of accomplishments, including the completion of the sequencing of the human genome [20, 48]. Sanger sequencing is usually known as the *first generation sequencing technique*. To se-

quence a genome, DNA is first broken into manageable pieces. Then the fragments are multiplied through a process called *cloning*, and finally individual fragments are sequenced. In the end, a library of DNA subsequences is generated. Figure 2.3 shows the basic steps of the Sanger sequencing technique; these steps are discussed below.

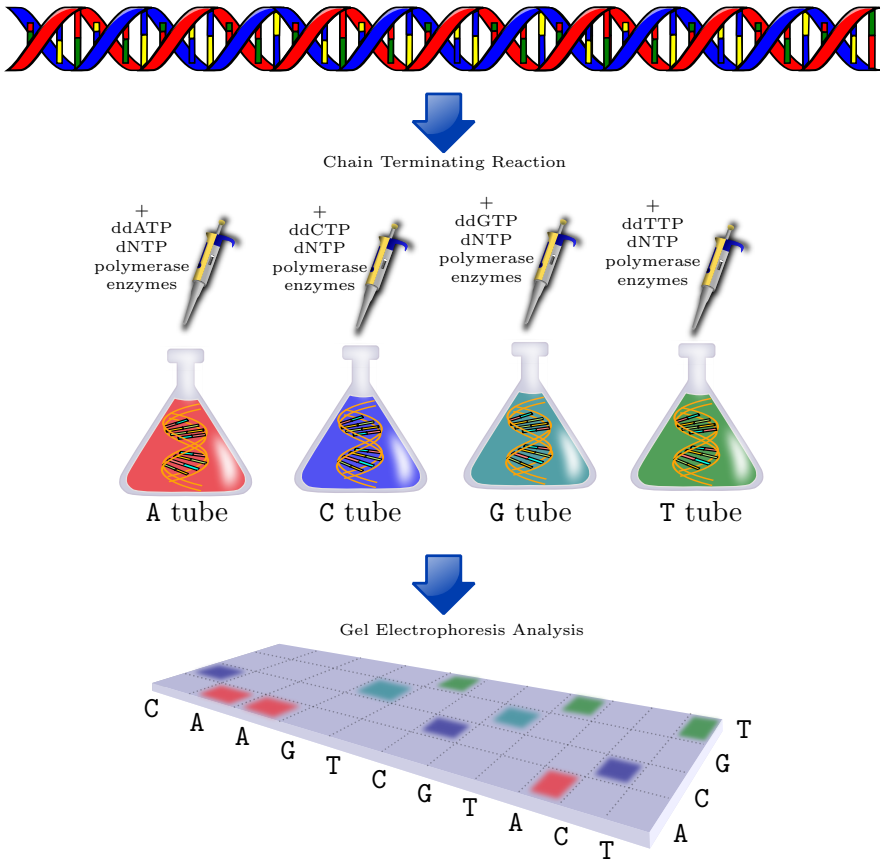


Figure 2.3: Sanger sequencing technique.

Template Preparation

First, the target DNA sequence is broken into manageable pieces and then DNA fragments are heated to denature the double strands. *DNA denaturation*, also known as *DNA melting*, is the process of unwinding the two strands of DNA and separating them by applying heat. Heat separates the hydrogen bonds between the bases of the opposite strands. One of the strands is discarded and the other strand is used as a *template*. A new DNA strand is synthesized using the template strand. Then, a primer is attached

to the template strand of the DNA; a *primer* is a short strand of nucleotides that acts as a starting point in DNA synthesis. *DNA polymerase*, an enzyme that acts as a catalyst to add DNA bases (deoxynucleotides) into an existing DNA strand, can add free floating nucleotides at the 3′-end of the primers to synthesize the complementary strand of the template strand. The temperature is then lowered to bind the primer sequence to its complementary sequence in the template strand. The primer is constructed in such a way that the 3′-end of the primer is located next to the target DNA sequence as shown in Figure 2.4; the primer is shown in blue.

Fragment 1	5′-CTTGCTTGCCA-3′
Fragment 2	5′-CTTGCTTGCCATTCGTTA-3′
Fragment 3	5′-CTTGCTTGCCATTCGTTAGCTTA-3′
Fragment 4	5′-CTTGCTTGCCATTCGTTAGCTTAGCTCCGA-3′
Fragment 5	5′-CTTGCTTGCCATTCGTTAGCTTAGCTCCGATTTA-3′
Template strand	3′-GAACGAACGGTAAGCAATCGAATCGAGGCTAAATGCA-5′

Figure 2.4: An example of potential fragments that are produced in the `A` tube.

Chain Terminating Reaction

After attaching the primers to multiple copies of a DNA fragment, the solution containing the fragments is divided into four different tubes, as shown in Figure 2.3, one for each of the nucleotides `A, C, G` and `T`; each tube is marked by the nucleotide's name. Sanger sequencing relies on *base-specific chain terminations* (i.e., sequences terminating at every position of a specific nucleotide) of four separate reactions (`A, C, G` and `T`) corresponding to the four different nucleotides in the DNA sequence. Four 2′-deoxynucleotide triphosphates (dNTPs), free floating nucleotides, and DNA polymerase are added to each of the tubes. Dideoxynucleotide adenine triphosphates (ddATPs) are added to the `A` tube, dideoxynucleotide cytosine triphosphates (ddCTPs) are added to the `C` tube, dideoxynucleotide guanine triphosphates (ddGTPs) are added to the `G` tube and dideoxynucleotide thymine triphosphates (ddTTPs) are added to the `T` tube.

As the complementary strand of the template strand is synthesized, deoxynucleotides

(dNTPs) are added to the growing chain by the DNA polymerase. This synthesized sequence is the complementary strand of the template strand. However, on occasions a dideoxynucleotide (ddNTP) instead of a deoxynucleotide is added to the chain, which stops the synthesis. Because of the chemical properties of dideoxynucleotides, no free floating nucleotides can be added after the dideoxynucleotide has been added to the synthesized sequence. Dideoxynucleotides stop the synthesis at the particular position where they are added. The strands can be terminated at any position resulting in a collection of DNA strands of different lengths. For example, the `A` tube contains segments of the template ending at every possible location of adenine, as shown in Figure 2.4. Similarly, DNA strands of different lengths ending at specific nucleotides are produced in each of the tubes. Heat is applied again to denature the double strands.

Sequencing

Combining all the sequences in the four tubes creates a collection of DNA strands ending at every possible position of the template. The strands are then transferred to a device called a *capillary electrophoresis tube* for *gel electrophoresis analysis*, a method to separate protein molecules according to their size/length using electric charge. DNA molecules in the capillary electrophoresis tube migrate from the negative pole of the tube to the positive pole as current passes through the gel. The synthesized sequences in the gel are separated according to their lengths, as the shortest molecules move the furthest. If all of the molecules from the four tubes are combined in one gel, the actual DNA sequences in the 5′-end to 3′-end direction can be determined by reading the patterns from the bottom to the top of the gel. The patterns in the gel reveal which of the bases end in each position of the template. Smaller fragments are produced when the ddNTP is added closer to the primer because then the chains are smaller and migrate further across the gel. Hence, sequences can be read from the gel base by base in order of length; this reveals the order of the bases in the synthesized sequence. Figure 2.4 shows the mixture of fragments in

the A tube where the primers are shown in blue. All the primers start sequencing from the same nucleotide and end in a specific base (e.g., adenine (A) in Figure 2.4) depending on the ddNTP used in the solution. The strands produced in the gel are complementary strands of the template strand.

2.2.1 Applications

The Human Genome Project (HGP) was an international research project that started in 1990. The goal of HGP was to sequence the complete human genome. The first draft of human genome was published in 2000 [20, 48]. The project was finished in 2003, when the first complete human genome was published [7]. About one third of the sequences of the HGP project were obtained using the Sanger sequencing method [8]. Sanger sequencing also played a major role in obtaining the DNA sequence of mice.

2.2.2 Disadvantages

The Sanger sequencing technique has a few disadvantages. The major one is that it is a costly and time consuming process. At about $1 per kbp (kilo base pairs), it would cost about $30,000,000 to sequence a complete human genome with 10x coverage (for an explanation of the notion coverage, see Section 3.1.5). In 2005, it was estimated that it would cost about $12,000,000 to sequence a mammalian genome using the Sanger method [32]. The coverage of Sanger sequencing is usually low, making it impossible to sequence many parts of a genome using single-end reads, because it is not possible to sequence parts of a genome which are not sampled. Moreover, it is not possible to clone some parts of a chromosome with the Sanger method because the cloning method used is biologically biased. In gel electrophoresis analysis, numerous strands may pile up in the gel when current is passed. These band pile-ups may introduce sequencing errors, because the machine used to read the sequence is unable to differentiate the bases if they are very close to one another in the gel.

2.3 Next Generation Sequencing (NGS)

Despite the many technological improvements to the Sanger method, the limitations of Sanger sequencing showed the need for new and improved genome sequencing technologies. Scientists have recently developed several techniques such as Roche/454, Illumina/-Solexa and SOLiD sequencing, generally referred to under the name of *next generation sequencing* (NGS), to overcome the limitations of the Sanger method. Next generation sequencing techniques produce fairly short reads (about 30 to 300 base pairs), but they are much cheaper to produce and they are produced much faster than with the Sanger method [49]. The cost of DNA sequencing using next generation sequencing techniques keeps falling rapidly [47]. However, NGS reads have more errors than the reads produced by the Sanger method. Next generation sequencing techniques consist of two major steps. In the first step, random fragments from the DNA known as templates are collected and cloned. In the second step, each of the cloned templates is sequenced and the sequencing information is read by a computer. Some of the main NGS sequencing techniques are discussed below.

2.3.1 Roche/454 Sequencing

454 sequencing was the first next generation sequencing technique introduced by 454 Life Sciences (`www.454.com`) in 2004. In 2007, 454 Life Sciences was acquired by Roche Applied Science (`www.roche-applied-science.com`). Roche/454 sequencing has higher throughput than Sanger sequencing and achieves this by using a large scale parallel pyrosequencing system capable of sequencing 400 to 600 Mbp (million base pairs) per 10 hour run [26]. The two major steps in the Roche/454 sequencing are discussed below.

Library Preparation

Roche/454 sequencing prepares a library by breaking the DNA into double-stranded fragments of 400 to 600 base pairs. Then, adapter *oligonucleotides* are attached on both ends of the DNA fragments. An *adapter* is a short double stranded DNA molecule, which is chemically synthesized to join the ends of two other double stranded DNA molecules. These adapters act as a starting point for both cloning and sequencing. The DNA library fragments are put onto micron-sized capture beads by emulsion-based clonal amplification. The amplification (i.e., multiplying each fragment) of the DNA fragments helps to easily detect the signals produced in the sequencing step. The amplification process in Roche/454 sequencing takes about eight hours, whereas the cloning process of Sanger sequencing takes several weeks.

The fragments are mixed with capture beads, enzyme reagents and synthetic oil in a cylindrical plastic container. The solution is then vigorously shaken to form droplets around the beads known as *emulsions*. Each of the emulsions contains only one DNA fragment. The enzyme in the emulsion amplifies the DNA fragments into approximately ten million identical copies. This reaction is called *polymerase chain reaction* (PCR) [52]. The fragments in each bead are double stranded, so they are denatured using heat to release one of the strands. This creates a library known as a *single-stranded template DNA* (sstDNA) library; sstDNA libraries with fewer templates than normal ($\approx$ 10 millions) are discarded. Figure 2.5 shows the library preparation procedure of Roche/454 sequencing.

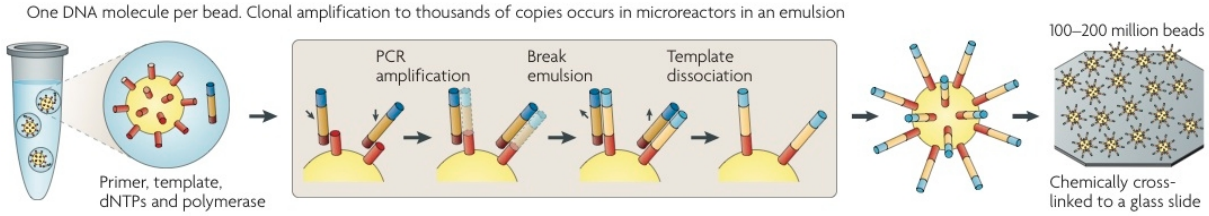


Figure 2.5: Library preparation of Roche/454 sequencing [33]. Reprinted with permission from Macmillan Publishers Ltd: *Natural Reviews Genetics*. Copyright 2009.

Pyrosequencing

A PicoTiterPlate (PTP) is used to sequence the DNA-capture beads. A *PicoTiterPlate* is a flat plate containing about 1.6 million wells in one side that act as small test tubes. Only one capture bead fits into a PTP well. The PTP spins and the centrifugal force deposits the beads into the wells. A device called *Genome Sequencer FLX* is then used to read the sequences from the PTP. Four free floating nucleotides (dNTPs) are added in the PTP in a fixed order. When a complementary nucleotide is added in the PTP, the DNA polymerase adds one or more of the nucleotides at the end of the adapter chain. When one or more of the nucleotides are added at the end of the chain, they emit a light signal which is recorded by the FLX system using a camera. This process is called *pyrosequencing* [42]. The order of the signals spells the sequence of bases in the template. Figure 2.6 shows the pyrosequencing process used in Roche/454 sequencing.

2.3.2 Illumina/Solexa Sequencing

Illumina sequencing, originally named Solexa sequencing, developed by Illumina Inc. (`www.illumina.com`) is capable of sequencing over a billion high quality bases per run. Illumina sequences DNA based on reversible dye-terminators as explained below. The reads produced by Illumina are shorter than Roche/454 sequencer. However, it has higher throughput than Roche/454 sequencer.

Amplification

First, the DNA is randomly fragmented into smaller pieces and adapters are ligated to the ends of the fragments. Adapter ligated DNA fragments are then put onto the surface of the flow cell channels. A *flow cell* is a microscope slide with eight channels, each channel holding the samples for sequencing. Free floating nucleotides and enzymes are added to the flow cell channels to initiate amplification. DNA fragments bend over to find complementary primer on the surface of the flow cell as shown in Figure 2.7.

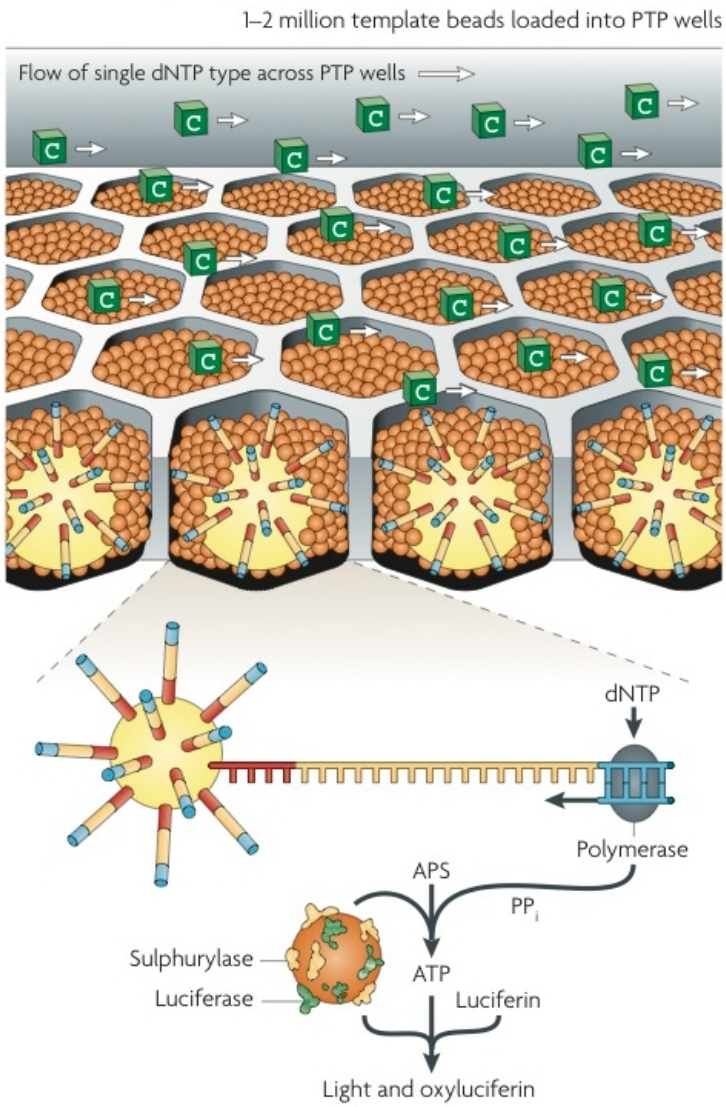


Figure 2.6: Pyrosequencing of Roche/454 sequencing [33]. Reprinted with permission from Macmillan Publishers Ltd: *Natural Reviews Genetics*. Copyright 2009.

The enzymes help the free floating nucleotides to build double-stranded sequence. The complementary strand is synthesized from the primer on the surface of the DNA. Then the strands are denatured and the replication process is repeated to synthesize more copies of each fragment. The identical copies of a DNA fragment create a dense cluster on the surface of the flow cell. Each of the channels in the flow cell contains several million such clusters.

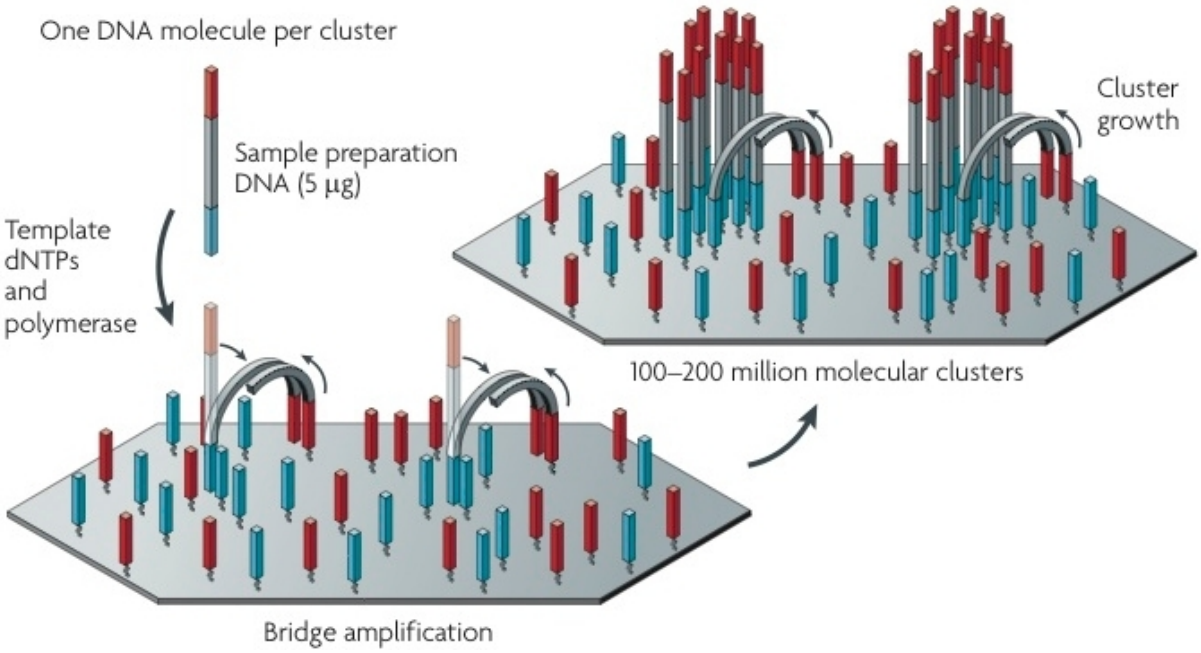


Figure 2.7: Library preparation of Illumina sequencing [33]. Reprinted with permission from Macmillan Publishers Ltd: *Natural Reviews Genetics*. Copyright 2009.

Sequencing by Synthesis

Primers, DNA polymerase enzyme, and four dye-labeled reversible terminators (colored nucleotides) are added to the flow cells. The enzyme helps to synthesize colored nucleotides. A laser is used to glow the color coded nucleotides and the color is read by a computer. The Illumina sequencer reads the sequences of all the clusters simultaneously, storing each color as a new base in the sequence. The sequence of colors in each cluster reveals the order of nucleotides in the fragments of that cluster. Figure 2.8 shows the synthesis process of Illumina.

2.3.3 SOLiD Sequencing

SOLiD (Sequencing by Oligonucleotide Ligation and Detection) sequencing was developed by Applied Biosystems (`www.appliedbiosystems.com`). This high throughput next generation sequencing technology can produce hundreds of millions to billions of short

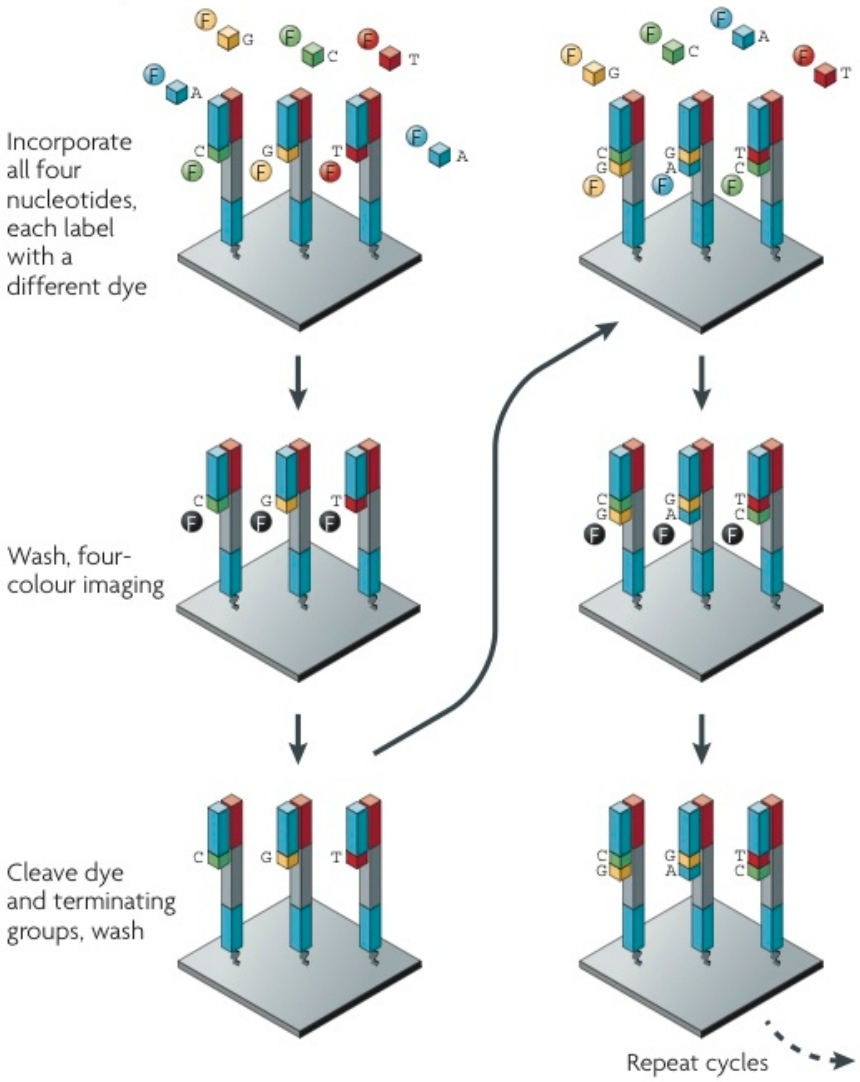


Figure 2.8: Sequencing by synthesis in Illumina sequencing [33]. Reprinted with permission from Macmillan Publishers Ltd: *Natural Reviews Genetics*. Copyright 2009.

reads at one time [25]. The robustness and flexibility of the SOLiD system enables a wide variety of applications including whole genome sequencing, and targeted re-sequencing (i.e., sequencing a particular region of interest of a DNA). The major steps in SOLiD sequencing are discussed below.

Library Preparation

Library preparation is the first step in the SOLiD sequencing. There are different methods for different types of libraries (i.e., single-end library, paired-end library, etc.). For a single-end library, the DNA is sheared into very small fragments of length 130 to 180 bp. For a paired-end library the DNA is fragmented into 600 bp to 6 kbp size fragments. In both libraries, targeted DNA is sheared into specific size fragments and adapters are ligated to the ends of the fragments. Paired-end libraries contain two pieces of DNA that are of approximately known distance apart in the target DNA. Each molecule is amplified onto beads in an emulsion PCR (emPCR), where individual molecules are isolated. Beads are then attached to a glass slide. The open slide format of SOLiD sequencing gives the flexibility of analyzing 1, 4, or 8 samples per slide. Then clone bead populations are prepared using microreactors (very small scale reactors) containing template, PCR reaction components, beads and primers. Each magnetic bead contains multiple copies of the same fragment. The SOLiD system is scalable; it can accommodate increasing densities of beads per slide to increase its throughput.

Sequencing by Ligation

Universal sequencing primer, ligates and di-base probes are added to the template beads as shown in Figure 2.9. The di-base probes are fluorescently labeled with four colors and each color represents four of sixteen possible di-base sequences. The complementary probe hybridizes to the template sequence and is ligated. After the color of the di-base probe is measured, the dye is cleaved off leaving a $5'$ phosphate group available for further reactions. This synthesis process is repeated for several cycles (two bases are sequenced in a cycle). Additional cycles can be added to extend the read length. Then the synthesized strand is removed from the fragment and a new primer is hybridized offset by one base (i.e., sequencing starts again from the next base pair in the template) and the ligation cycles are repeated. This primer reset process is repeated for several rounds, providing

multiple measurements of each base, which increases the sequencing accuracy.

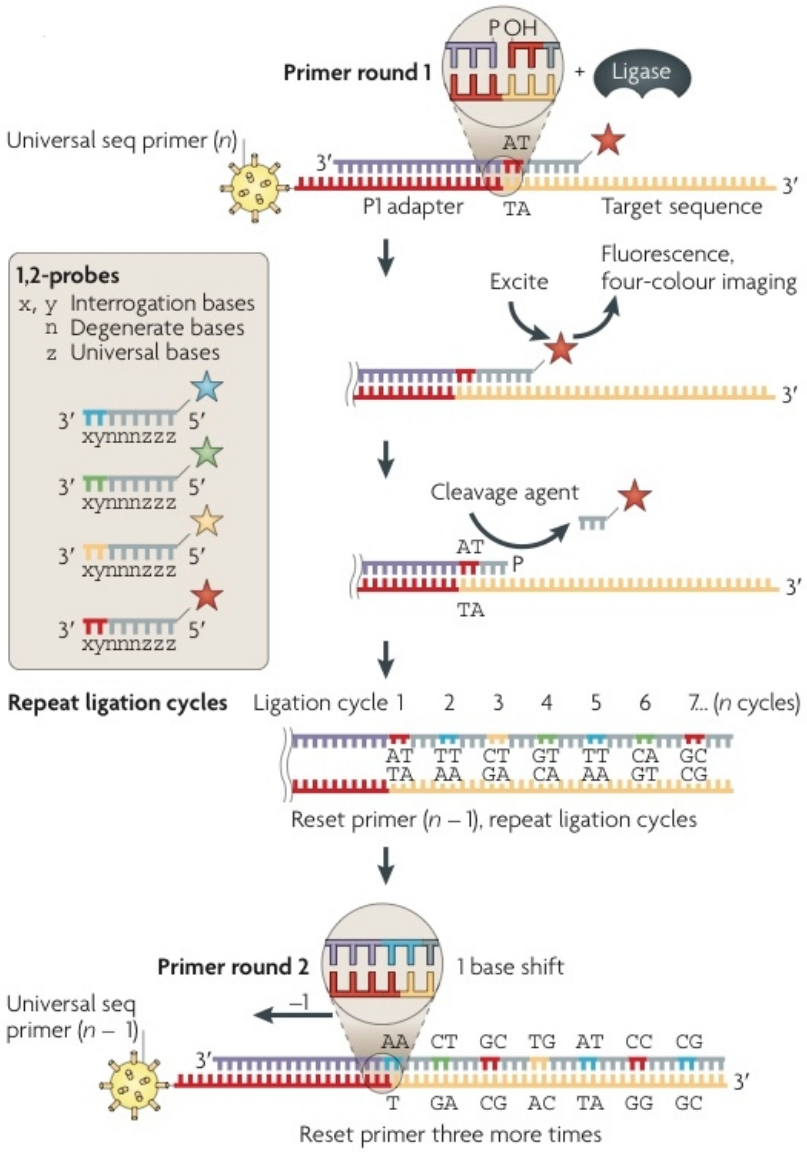


Figure 2.9: Sequence by ligation [33]. Reprinted with permission from Macmillan Publishers Ltd: *Natural Reviews Genetics*. Copyright 2009.

2.3.4 Applications

The low cost and high throughput of NGS make it useful for many applications. These include variant discovery and *de novo* assembly of bacterial and lower eukaryotic genomes.

Variant Discovery

NGS reads can be used to discover *structural variants* (i.e., difference between two genomes) by mapping a set of reads to a known reference genome. Since genomes of individuals of a species vary in only about 1%, the NGS reads of an individual can be mapped to a reference genome of that species to discover structural variants. For example, consider that a pair of reads r_1 and r_2 is 1 kbp apart in a given NGS dataset. After mapping them to a reference genome, if the distance between r_1 and r_2 is more than 1 kbp, it indicates that the NGS dataset has a structural variant called *deletion* relative to the reference genome. Similarly, if the distance between r_1 and r_2 is smaller than 1 kbp in the reference genome, then the NGS dataset has an *insertion* relative to the reference genome. Of course only one pair of reads is not enough to say that there are deletions or insertions in the genome of the dataset. Several mate pairs must agree on the insertions and deletions to accurately detect them. Since NGS reads are short and have high coverage, it is feasible to use them for discovering structural variants. Short reads in NGS are easy to map to the reference genome and high coverage allows more mate pairs to discover structural variants.

De novo Genome Assembly

One of the main applications of NGS is *de novo* genome assembly. Assembling subsequences of an unknown genome is called *de novo* genome assembly. Since NGS techniques produce high coverage reads from a genome, it is feasible to assemble the genome without any prior knowledge of it. Shorter reads and high coverage of NGS reads play an important role in *de novo* genome assembly. Details on *de novo* genome assembly are discussed in Chapter 3.

2.3.5 Advantages

The major advantage of NGS is that it can produce an enormous volume of DNA sequences faster than the Sanger method. For example, Roche/454 is capable of sequencing 400 to 600 million base pairs per 10 hour run. Library preparation in NGS can be done in few hours, whereas the same process in the Sanger method takes several weeks. Next generation sequencing is also cheaper than the Sanger sequencing. Currently it costs about $0.1 per Mbp (million base pairs) [51] and the cost is halving every two years. Figure 2.10 shows the cost of next generation sequencing in US dollars per Mbp over the last few years (y axis in logarithmic scale).

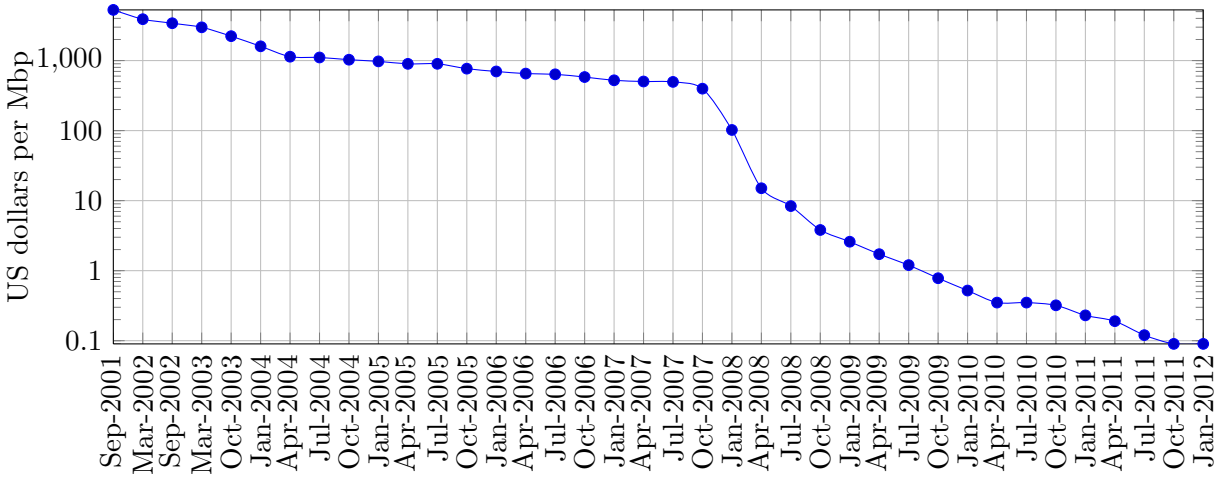


Figure 2.10: Next generation sequencing cost in US dollars per Mbp [51].

Scientists hope that in the near future whole human genomes can be sequenced using NGS for about $1000. Several companies have managed to reduce the cost of sequencing the human genome dramatically in the past decade. The human genome sequencing cost of Illumina sequencing was reduced from $100,000 in 2008 to $50,000 in 2010. Stanford university professor Stephen Quake sequenced his own genome for $48,000 in 2009 [41]. All these dramatic reductions of sequencing cost are possible because of the next generation sequencing techniques.

2.3.6 Disadvantages

Next generation sequencing techniques have some disadvantages. The reads produced by NGS contain more errors per base pair than the reads produced by the Sanger method. All genome assemblers use some overlapping technique to determine the order in which the reads appear in the genome. Errors in the reads make the process of reconstructing the genome from the reads harder because the overlapping technique used in the process is unable to find overlaps if the reads contain errors. Assemblers need a good error correction technique to correct the errors before using the reads.

Reads produced by NGS are short. Genomes might contain repeated subsequences of bases and with short reads it may not always be possible to detect repeated sequences that are longer than the reads. If the reads are very short, then it is more likely that they will be present in multiple locations in the genome from where they were taken and this might introduce false overlaps between reads (see Section 3.2). Thus longer reads are more desirable than short ones for genome assembly.

Chapter 3

NGS *de novo* Genome Assembly

As mentioned in the previous chapter, the enormous number of short reads produced by next generation sequencing techniques such as Roche/454, Illumina/Solexa and SOLiD sequencing opened the possibility of *de novo* genome assembly. The high coverage at low cost of next generation sequencing makes it possible to assemble complex genomes without any prior information about them. Some of the statistical analyses used for *de novo* genome assembly require high coverage, which is only possible with next generation sequencing techniques.

3.1 Problem Description

Whole genome shotgun (WGS) sequencing is a process in which the genome of an organism is fragmented into reads, which are then put together to reconstruct the genome. *Genome assembly* is the process of WGS in which the fragments are combined to reconstruct the original genome. Reads of a DNA fragment can be represented by strings over the alphabet $\Sigma = \{\texttt{A, C, G, T}\}$, which represents the four nucleotides present in a DNA fragment. Given a set of reads obtained from a genome $\mathscr{G}$, the genome assembly problem is to assemble the reads to reconstruct the original genome $\mathscr{G}$.

Formally, given a set $R = \{r_1, r_2, ..., r_n\}$ of n reads over the alphabet $\Sigma = \{\texttt{A, C, G, T}\}$,

where the length of the reads is $|r_i| = l$, so $r_i \in \Sigma^l$ where Σ^l is the set of all strings of length l found by the symbols in Σ. The goal is to determine the string $\mathscr{G}$ such that all reads $r_i \in R$ are substrings of $\mathscr{G}$, each read appears in $\mathscr{G}$ a pre-specified number of times, and $|\mathscr{G}|$ is minimum. Some substrings of $\mathscr{G}$ may appear multiple times in it. The pre-specified number of times that a read r_i appears in $\mathscr{G}$ is called the *copy count* of r_i.

Example

Consider the set R ={`ACG, CGA, CGC, CGT, GAC, GCG, GTA, TCG`} of $n = 8$ reads of length $l = 3$, each read has multiplicity 1. Concatenating all the reads in R produces the string of length 24 shown in Figure 3.1. Note that, this is not the shortest possible string for which each read $r_i \in R$ is a substring of it. Figure 3.2 shows the shortest possible such string that has all reads as substrings.

```
ACGCGACGCCGTGACGCGGTATCG
012345678901234567890123
```

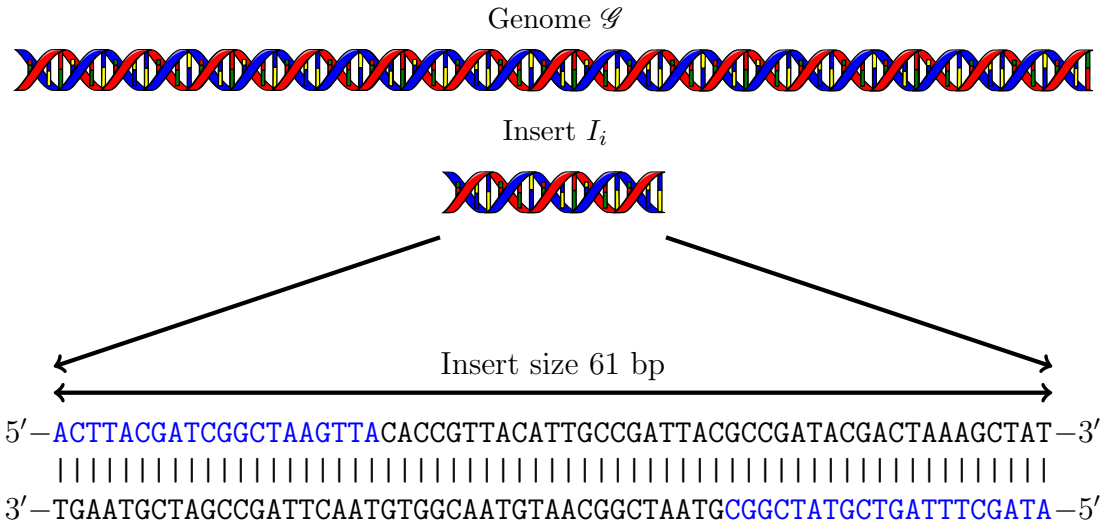
Figure 3.1: Concatenation of the reads.

```
TCGACGCGTA
0123456789
```

Figure 3.2: Shortest superstring of the reads.

3.1.1 Reads

Genome sequencing techniques cannot read a whole genome at one time. They can only read short fragments. DNA fragments, or reads, are always sequenced in the same direction, from the 5′-end to the 3′-end as explained in Section 2.1. However, it is not possible to know from which of the two strands a read was taken. Figure 3.3 shows two reads r_1 and r_2 of length 20 bp that were sequenced from a DNA fragment of length 61 bp.



Read r_1: 5′−ACTTACGATCGGCTAAGTTA−3′
Read r_2: 5′−ATAGCTTTAGTCGTATCGGC−3′
Insert size: 61 bp
Read length l: 20 bp

Figure 3.3: Reads, mate pair, and insert size.

3.1.2 Reverse Complement

Complementarity is a property of double-stranded nucleotide sequences such as DNA. Each nucleotide from one strand of a DNA molecule is connected to a nucleotide in the opposite strand via either two hydrogen bonds or three hydrogen bonds. Since each base in one stand has one complementary base in the opposite strand in a DNA sequence, then if the sequence of bases in a strand is known, the sequence of bases in the opposite strand can be easily reconstructed. DNA strands are always read from the 5′-end to the 3′-end, so for a given strand the sequence of bases in the opposite strand is read in the reverse order. This is why the strands in a DNA fragment are said to be the *reverse complement* of each other. The reverse complement of a DNA sequence s is denoted by $\bar{s}$. In Figure 3.3, the first read r_1 is taken from the 5′-end of the forward strand and the second read r_2 is taken from the 5′-end of the reverse strand. Figure 3.4 shows an example of a read r and its reverse complement $\bar{r}$.

```
5′−AGCTAAGCATTTACGATAGCCGATAGCTAAATTAC−3′
   ||||||||||||||||||||||||||||||||||
3′−TCGATTCGTAAATGCTATCGGCTATCGATTTAATG−5′

Read r: 5′−AGCTAAGCATTTACGATAGCCGATAGCTAAATTAC−3′
Read r̄: 5′−GTAATTTAGCTATCGGCTATCGTAAATGCTTAGCT−3′
```

Figure 3.4: Reverse complement $\bar{r}$ of a read r.

3.1.3 Mate Pairs and Insert Size

Sometimes DNA sequencing techniques sequence reads in pairs, one from each strand of a DNA fragment, with approximately known distance between the reads. Paired-end reads are helpful for genome assembly [9]. To sequence reads in pairs, sequencing techniques first break the genome $\mathscr{G}$ into a set $I = \{I_1, I_2, ..., I_k\}$ of small double-stranded fragments known as *inserts*. Then each insert $I_i \in I$ is sequenced from the 5′-end of each of the strands. The approximate length $|I_i|$ of each insert I_i is known and it is called the insert size. A pair of reads with known insert size is called a mate pair. In Figure 3.3, an insert I_i of length 61 bp from the genome $\mathscr{G}$ is sequenced from the 5′-end of each of the strands. The read r_1 is sequenced from the 5′-end of the forward strand of I_i, while the read r_2 is sequenced from the 5′-end of the reverse strand of I_i.

3.1.4 Repeats and Copy Counts

Repeats are multiple copies of the same sequence of bases in a DNA fragment. The number of times that a subsequence appears in a genome is called its copy count. In Figure 3.5, the sequence S_1 (`ATGCA`) appears in three different locations in genome $\mathscr{G}$, hence the copy count of S_1 is 3. Similarly the string S_2 (`TAGTT`) appears twice in $\mathscr{G}$, so the copy count of S_2 is 2. Multiple copies of the same sequence in the genome make genome assembly harder, because the reads taken from the positions close to the endpoints of a repeated sequence overlap with more than one part of the genome, which makes it more difficult to reconstruct the genome from the reads (see Section 3.2).

Figure 3.5: Repeats and copy counts.

Sometimes a sequence in the genome is repeated, except for a single base in the repeated sequence is altered. These, "almost" repeats are known as *single nucleotide polymorphisms* (SNP). For example, in Figure 3.6 the sequences AGATTACGGGA and AGATTGCGGGA in the genome $\mathscr{G}$ differ by only a single nucleotide, so they are referred to as SNPs. These SNPs create bubble-like structures in the overlap graph as discussed in Section 4.5.2.

...AGATTACGGGA.....AGATTGCGGGA...

Figure 3.6: Single nucleotide polymorphisms in the genome $\mathscr{G}$.

When repeated sequences appear in consecutive positions of the genome they are called *tandem repeats*. Tandem repeats are generally associated with non-coding DNA fragments. *Non-coding DNA* fragments are segments of DNA that are not encoded for protein synthesis. Figure 3.7 shows a tandem repeat in a genome where the string ACGT is repeated three times.

...GACGTACGTACGTT...

Figure 3.7: A tandem repeat in the genome $\mathscr{G}$.

3.1.5 Coverage

Coverage is defined as the average number of times that each base pair in a genome is sequenced. Given a dataset of n reads of length l from a genome of length L, then

$$Coverage = \frac{n \times l}{L} \tag{3.1}$$

The quality of an assembled genome depends on the coverage of the dataset. If the

coverage is low then the dataset might not have enough reads from some parts of the genome to reconstruct it accurately. Note that it is impossible to assemble the parts of the genome that are not sampled. High coverage increases the probability that every base pair in the genome is sampled. High coverage (about 50x to 400x) datasets are used in the experiments mentioned in Chapter 5.

3.2 Overlap Graph

An overlap graph $G = (V, E)$ is a bidirected graph in which each node $v \in V$ represents a read from a given collection R of reads and each edge $e = (u, v) \in E$ represents an overlap between reads u and v. Each edge e in the overlap graph has two arrowheads; one at each endpoint. The orientations of the arrowheads are used to denote the different ways in which the two reads at the ends of an edge can overlap. Since the DNA is double stranded, bidirected edges are used to represent the double stranded structure of DNA sequences. The bidirected graph representation for DNA sequences was proposed by Kececioglue [19]. Details on bidirected edges can be found in Section 3.2.4.

In a bidirected overlap graph, the string representing read r should be considered for computing overlaps when entering the node representing r through an in-edge (or when exiting the node through an out-edge) and the reverse complement $\bar{r}$ of the string representing r should be considered when entering the node representing r through an out-edge (or when exiting the node through an in-edge). Consider the set of reads R in Figure 3.8. Overlaps among the reads in R are shown in Figure 3.9 and the corresponding overlap graph is shown in Figure 3.10. Overlap graphs are used in several genome assemblers such as Edena [14] and SGA [45].

Repeats in a DNA sequence can introduce false overlapping edges in an overlap graph. In Figure 3.11, the sequence `GCATTTACGATAGC` is repeated twice in the DNA sequence. This repeated sequence introduced false overlapping edges in the overlap graph as shown

```
Read r1  : AGCTAAGCATTTACGATAGCCGATAGCTAAATTAC
Read r2  : CGTAATTTAGCTATCGGCTATCGTAAATGCTTAGC
Read r3  : AACGTAATTTAGCTATCGGCTATCGTAAATGCTTA
Read r4  : GCATTTACGATAGCCGATAGCTAAATTACGTTATA
Read r5  : GTATAACGTAATTTAGCTATCGGCTATCGTAAATG
Read r6  : ATTTACGATAGCCGATAGCTAAATTACGTTATACT
Read r7  : TTTACGATAGCCGATAGCTAAATTACGTTATACTC
Read r8  : ATATAACGTAATTTAGCTATCGGCTATCGTAAATG
Read r9  : ATTTACGATAGCCGATAGCTAAATTACGTTATATA
Read r10 : CTATATAACGTAATTTAGCTATCGGCTATCGTAAA
```

Figure 3.8: Set of given input reads.

```
Read r1   :  AGCTAAGCATTTACGATAGCCGATAGCTAAATTAC
Read r̄2   :   GCTAAGCATTTACGATAGCCGATAGCTAAATTACG
Read r̄3   :     TAAGCATTTACGATAGCCGATAGCTAAATTACGTT
Read r4   :        GCATTTACGATAGCCGATAGCTAAATTACGTTATA
Read r̄5   :         CATTTACGATAGCCGATAGCTAAATTACGTTATAC
Read r6   :          ATTTACGATAGCCGATAGCTAAATTACGTTATACT
Read r7   :           TTTACGATAGCCGATAGCTAAATTACGTTATACTC
Read r̄8   :         CATTTACGATAGCCGATAGCTAAATTACGTTATAT
Read r9   :          ATTTACGATAGCCGATAGCTAAATTACGTTATATA
Read r̄10  :           TTTACGATAGCCGATAGCTAAATTACGTTATATAG
```

Read Length l : 35 base pairs
minOverlap : 32 base pairs

Figure 3.9: Overlapping reads.

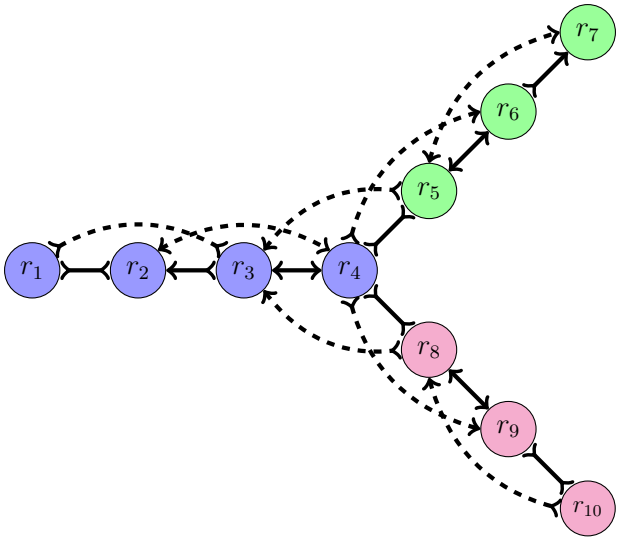


Figure 3.10: Overlap graph with 10 reads.

in dashed lines in Figure 3.11.

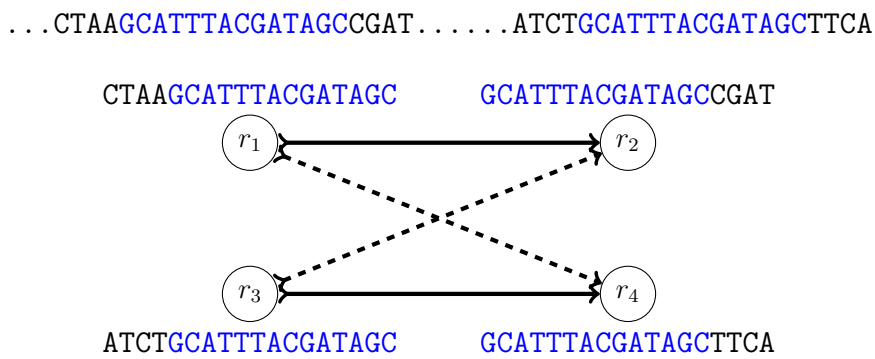


Figure 3.11: False branching in overlap graph caused by repeated sequences.

A read taken from the part of a genome where a tandem repeat appears overlaps in both directions with itself and all other reads taken from nearby positions, because the read has a long enough proper prefix which is also present as a proper suffix of the read. Figure 3.12 shows a read u containing a tandem repeat GTCT, this tandem repeat has *period* (i.e., length of the repeated pattern) four. The read u overlaps with itself and another read v from the repeat in both directions.

```
...CTTAGGTCTGTCTGTCTGTCTGTCTGTCTGTCTGTCTGTCTATATG...

Read u:  GTCTGTCTGTCTGTCTGTCTGTCTGTCT
Read v:   TCTGTCTGTCTGTCTGTCTGTCTGTCTG
Read u:      GTCTGTCTGTCTGTCTGTCTGTCTGTCT
Read v:       TCTGTCTGTCTGTCTGTCTGTCTGTCTG
Read u:          GTCTGTCTGTCTGTCTGTCTGTCTGTCT
```

Figure 3.12: Self overlapping repetitive sequence from a tandem repeat GTCT.

3.2.1 Overlap Length

The *overlap length* between two strings s_1 and s_2 is defined as the length of the longest overlapping suffix of s_1 and prefix of s_2 (or suffix of s_2 and prefix of s_1). In Figure 3.13, the last 30 base pairs of read u are the same as the first 30 base pairs of read v, meaning that they overlap by 30 base pairs. The overlap length of two reads u and v is denoted by $overlapLength(u, v)$. For the two reads in Figure 3.13, the corresponding edge

$e = (u, v)$ has overlap length $overlapLength(e) = 30$; this edge e represents the string AGCTAAGCATTTACGATAGCCGATAGCTAAATTACGTTAT which is found by overlapping the reads in e.

```
Read u: AGCTAAGCATTTACGATAGCCGATAGCTAAATTAC
Read v:      AGCATTTACGATAGCCGATAGCTAAATTACGTTAT
```

Figure 3.13: Overlap length, $overlapLength(u, v) = 30$.

3.2.2 Minimum Overlap Length

The minimum overlap length, denoted as *minOverlap*, is a parameter of the overlap graph specifying the minimum number of base pairs by which two reads $u, v \in R$ must overlap to insert an edge $e = (u, v)$ between them in the overlap graph. If, for example, the minimum overlap length for an overlap graph is 32, then no edge between the reads u and v of Figure 3.13 would be inserted since $overlapLength(u, v) < minOverlap$. If, on the other hand, the minimum overlap length is 28, then the edge $e = (u, v)$ would be inserted in the overlap graph because then $overlapLength(u, v) \geq minOverlap$.

3.2.3 Simple and Composite Edges

If a read r_1 overlaps with another read r_2 then the edge $e = (r_1, r_2)$ denoting the overlap is called a *simple edge*. In Figure 3.14, edges (r_1, r_2) and (r_2, r_3) are simple edges. A path formed by nodes of degree 2 (nodes with only 2 incident edges; one is an in-edge and the other is an out-edge) in the overlap graph can be simplified to a single edge to reduce the size of the graph. When one such path is simplified to an edge, the resulting edge is called a *composite edge*. A composite edge stores information about the overlapping reads in the original path. In Figure 3.14, the path $\{r_1, r_2, r_3\}$ can be simplified to an edge (r_1, r_3), because node r_2 has degree 2; Figure 3.15 shows the composite edge (r_1, r_3).

We say that a simple edge (r_1, r_2) *spells* the string obtained by overlapping the reads r_1 and r_2 as shown in Figure 3.14. When talking about overlaps between reads we always

```
Read r1: AGCTAAGCATTTACGATAGCCGATAGCTAAATTAC
Read r2:  GCTAAGCATTTACGATAGCCGATAGCTAAATTACG
Read r3:    TAAGCATTTACGATAGCCGATAGCTAAATTACGTT

string(r1, r2) : AGCTAAGCATTTACGATAGCCGATAGCTAAATTACG
string(r2, r3) :  GCTAAGCATTTACGATAGCCGATAGCTAAATTACGTT
```

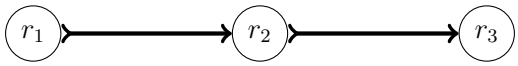


Figure 3.14: Simple edges (r_1, r_2) and (r_2, r_3) in the overlap graph.

mean the maximum overlap between them. Similarly, the string spelled by a path p is found by overlapping, in order, the reads in p. Note that, nodes in a valid path have one in-edge and one out-edge in the path. Since, composite edges store the ordered list of reads in the path that they represent, the string spelled by a composite edge can be found by overlapping the ordered reads stored in the edge including the reads in the nodes connected by the edge as shown in Figure 3.15. Length of an edge $e = (u, v)$ is defined as the index of the last read v in the string spelled by the edge e. For example, in Figure 3.15, the string spelled by the edge $e = (r_1, r_3)$ is AGCTAAGCATTTACGATAGCCGATAGCTAAATTACGTT and the location of the read r_3 on the edge e is 3. Therefore, the length of the edge $e = (r_1, r_3)$ is 3 bp.

```
Read r1: AGCTAAGCATTTACGATAGCCGATAGCTAAATTAC
Read r2:  GCTAAGCATTTACGATAGCCGATAGCTAAATTACG
Read r3:    TAAGCATTTACGATAGCCGATAGCTAAATTACGTT

string(r1, r3) : AGCTAAGCATTTACGATAGCCGATAGCTAAATTACGTT
```

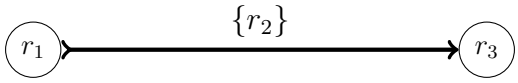


Figure 3.15: Composite edge (r_1, r_3) in the overlap graph.

3.2.4 Types of Overlaps

The use of bidirected edges to represent double stranded DNA sequences was first pro-

posed by Kececioglue [19]. We use a similar bidirected arrow notation to represent the overlap between two double stranded sequences in the overlap graph. Since each node in the overlap graph represents a read r and its reverse complement $\bar{r}$, either the read r or its reverse complement $\bar{r}$ can overlap with other reads. There are three different ways in which two reads r_1 and r_2 can overlap, which we call forward-forward, reverse-forward and forward-reverse overlap.

Forward-Forward Overlap

When a suffix of a read r_1 overlaps with a prefix of a second read r_2, the overlap is called a *forward-forward overlap*. This means that most likely both r_1 and r_2 are from the same strand of the DNA from which the reads were taken. A forward-forward overlap is represented with the bidirected edge shown in Figure 3.16. The edge (r_1, r_2) is an *out-edge* for node r_1 and an *in-edge* for node r_2. Note that when two reads r_1 and r_2 have a forward-forward overlap between them, as shown in Figure 3.16, their reverse complements $\bar{r_1}$ and $\bar{r_2}$ also overlap. The overlap between $\bar{r_1}$ and $\bar{r_2}$ is called a reverse-reverse overlap.

```
Read r1: AGCTAAGCATTTACGATAGCCGATAGCTAAATTAC
Read r2:   CTAAGCATTTACGATAGCCGATAGCTAAATTACAT
```

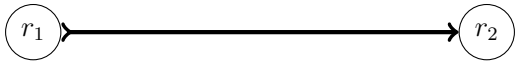


Figure 3.16: Forward-forward overlap between two reads.

Reverse-Forward Overlap

When a suffix of the reverse complement $\bar{r_1}$ of a read r_1 overlaps with a prefix of a second read r_2, the overlap is called a *reverse-forward overlap*. This means that most likely r_1 and r_2 are from opposite strands of the DNA from which the reads were taken. A reverse-forward overlap is represented with the bidirected edge shown in Figure 3.17.

```
Read r1: GTAATTTAGCTATCGGCTATCGTAAATGCTTAGCT
Read r2: CTAAGCATTTACGATAGCCGATAGCTAAATTACAT

Read r̄1: AGCTAAGCATTTACGATAGCCGATAGCTAAATTAC
Read r2:   CTAAGCATTTACGATAGCCGATAGCTAAATTACAT
```

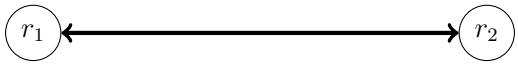


Figure 3.17: Reverse-forward overlap between two reads.

Forward-Reverse Overlap

When a suffix of a read r_1 overlaps with a prefix of the reverse complement $\bar{r_2}$ of a second read r_2, the overlap is called a *forward-reverse overlap*. This means that most likely r_1 and r_2 are taken from the opposite strands of the DNA sequence. Figure 3.18 shows a forward-reverse overlap between the reads r_1 and r_2.

```
Read r1: AGCTAAGCATTTACGATAGCCGATAGCTAAATTAC
Read r2: ATGTAATTTAGCTATCGGCTATCGTAAATGCTTAG

Read r1: AGCTAAGCATTTACGATAGCCGATAGCTAAATTAC
Read r̄2:   CTAAGCATTTACGATAGCCGATAGCTAAATTACAT
```

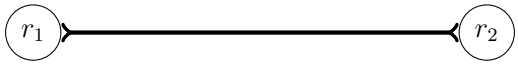


Figure 3.18: Forward-reverse overlap between two reads.

3.2.5 Transitive Edges

If the overlap graph has a triangle $\Delta r_1 r_2 r_3$, (i.e., there are edges between r_1 and r_2, r_2 and r_3, and r_1 and r_3) and the string spelled by one of the edges is the same as the string spelled by the path through the other two edges, then the first edge is called a *transitive edge* and the triangle is called a *transitive triangle*. A transitive edge contains redundant information since the information represented by such an edge can be found by traversing the other two edges in the transitive triangle. Hence, transitive edges are

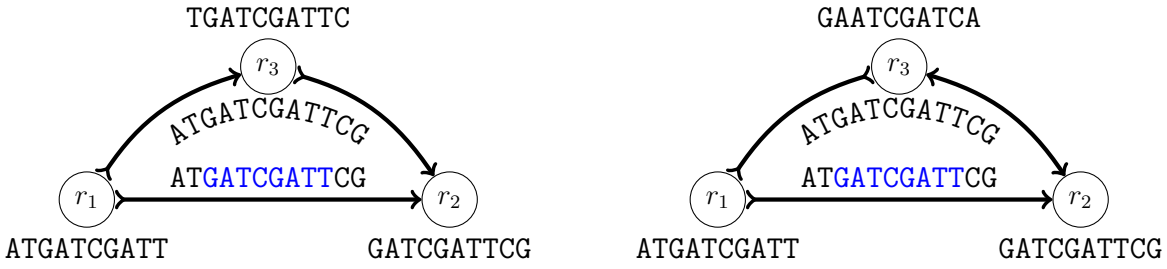


Figure 3.19: Edge $e = (r_1, r_2)$ is a transitive edge in both triangles.

removed to reduce the size of the graph. Figure 3.19 shows two such transitive triangles in the overlap graph. There are several efficient algorithms [1, 35] to remove transitive edges from a bidirected graph.

3.3 De Bruijn Graph

The k-th dimension *de Bruijn graph* $G = (V, E)$, proposed by Pevzner et al. [40], over an alphabet Σ is a directed graph in which every node represents a string of length k from Σ^k and a directed edge is added from node u to node v if the suffix of length $k-1$ of u and the prefix of length $k-1$ of v are the same. Figure 3.20 shows a simple 2-dimensional de Bruijn graph over the binary alphabet $\Sigma = \{0, 1\}$.

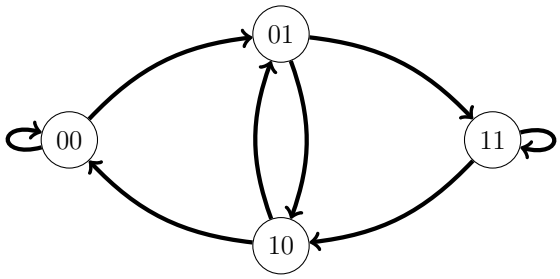


Figure 3.20: A simple 2-dimensional de Bruijn graph over the binary alphabet $\Sigma = \{0, 1\}$.

Idury and Waterman [16] introduced the *sequence graph* to represent the different ways of assembling a set of reads. Their idea was to generate a graph from all k nucleotide subsequences detected from a genome; these subsequences are known as k-mers. The sequence graph is simply a k-th dimensional de Bruijn graph whose nodes correspond

to the k-mers and edges represent $k-1$ base pairs overlap between the k-mers. Their algorithm produced contigs from the series of overlapping k-mers from the graph; where a *contig* is a continuous subsequence of bases from a DNA molecule that was constructed by overlapping reads.

Pevzner and Tang [40] then generalized the de Bruijn graph approach, where all possible substrings of length k from the set of reads are considered and the nodes in the graph represent overlapping k-mers in the graph and the edges represent $k-1$ base pairs overlap between the k-mer at one of the endpoints of the nodes. Several genome assemblers use de Bruijn graph approaches, such as EULER [5, 6, 38, 40], Velvet [53], ABySS [46] and SOAPdenovo [22]. Contigs are represented as the strings spelled by the unambiguous paths in the de Bruijn graph; here, an *unambiguous path* are paths formed by nodes of degree 2 (e.g., path $\{k_1, k_2, ..., k_9\}$ in Figure 3.21). Figure 3.21 shows the de Bruijn graph for all the 3-mers of the reads in Figure 3.8. Note that the path $\{k_1, k_2, ..., k_9\}$ will be merged to a single node representing the string `GACCTACAAGT`. The string `GACCTACAAGT` is a contig as it is found by overlapping the k-mers in an unambiguous path.

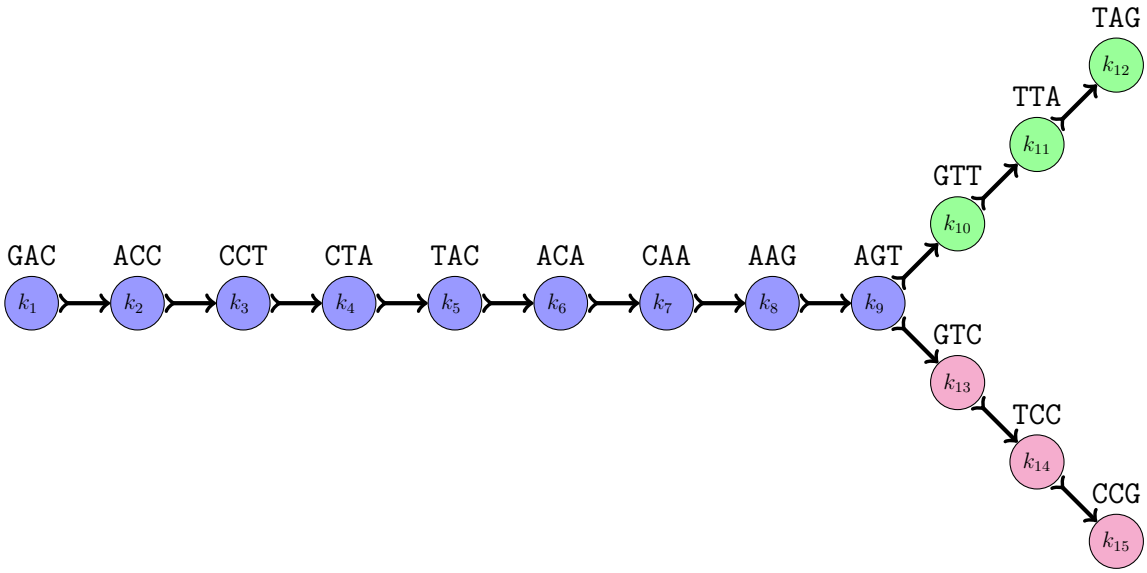


Figure 3.21: De Bruijn graph for the 3-mers of the reads in Figure 3.8.

3.4 File Formats

There are different text-based file formats for storing sequences of nucleotides from a genome. The most widely used are the FASTA and FASTQ formats. In both formats a sequence of nucleotides is stored as a string over the characters `A, C, G` and `T`; sometimes `N` is also used in a string to represent an unknown base.

3.4.1 FASTA File Format

A FASTA file can contain several DNA sequences in it. Each of the sequences in a FASTA file is represented in two lines. The first line contains a description of the sequence and the second line contains the sequence itself. The description of a sequence always starts with the symbol `">"`. Figure 3.22 shows the first few lines of the FASTA file of a dataset downloaded from the short read archive (SRA) database (`www.ncbi.nlm.nih.gov/sra`). In Figure 3.22, the first line contains some information about the first read such as *accession number* (a unique ID of the sequence) and mate pair information. In Figure 3.22, the first line contains the accession number (`ERR021957`), mate pair information (`.1` after the accession number. Each read in a mate pair has same unique number after the accession number), sequencing technique used (`IL` meaning Illumina) and read length (`37 bp`). These sequence descriptions are stored when the reads are sequenced by the machine.

3.4.2 FASTQ File Format

Sometimes while sequencing reads a score value is associated with the base pairs to denote the degree of confidence on their correctness. To specify these scores, the Wellcome Trust Sanger Institute (`www.sanger.ac.uk`) defined a FASTQ file format by combining FASTA sequences and their quality scores. Unlike the FASTA file format, FASTQ files use four lines to represent a sequence. The first line starts with a `"@"` symbol followed by a

```
>ERR021957.1 IL32_2532:2:1:0:489 length=37
NGCAGAGGATGCTGTTGCATACGCCACTGCTTTTGTA
>ERR021957.1 IL32_2532:2:1:0:489 length=37
NAACTATTCACCTCGCCCCGGATCGAACCTTCTTCCC
>ERR021957.2 IL32_2532:2:1:0:943 length=37
NTAATAGTCATGTTAAGCTCAGTCAGAGCTTCTTCTA
>ERR021957.2 IL32_2532:2:1:0:943 length=37
NCACGGTCTCCCTTGGGGACATAGTATTCCGCACGAA
>ERR021957.3 IL32_2532:2:1:0:74 length=37
NGATGCGATGAACAGTTTGCAAACAGCGTCCTTCCTA
>ERR021957.3 IL32_2532:2:1:0:74 length=37
NCTTAAAAGGGATTGCAGCTTGTCGTCCTGCTTGAGC
>ERR021957.4 IL32_2532:2:1:0:778 length=37
NAAATCGATCTCATTATGAATATCTCACCACACTCTA
>ERR021957.4 IL32_2532:2:1:0:778 length=37
NGTCTAGAGAATGCCGCTATTGAGGTAAGCTATTTCT
```

Figure 3.22: The first few lines of a FASTA file.

sequence identifier and an optional description of the sequence like the FASTA files. The second line contains the sequence of nucleotides. The third line starts with a `"+"` symbol followed by an optional identifier and description which is similar to that in the first line, but sometimes it stores additional information about the sequence. The fourth line contains quality scores of each one of the bases in the sequence. Each character in the fourth line represents the quality score Q of the corresponding base in the second line. The quality score Q is the integer mapping of the probability p for the corresponding base to be incorrect. Corresponding ASCII value of Q is stored to represent the quality score. Figure 3.23 shows the first few lines of the FASTQ file of a dataset downloaded from the short read archive (SRA) database (`www.ncbi.nlm.nih.gov/sra`).

3.5 Existing Genome Assemblers

Several algorithms exist for *de novo* genome assembly. Figure 3.24 shows the basic steps in genome assembly. First, the genome of an organism is cloned to produce multiple copies of it (Figure 3.24a), and then they are sheared into a large number of small fragments

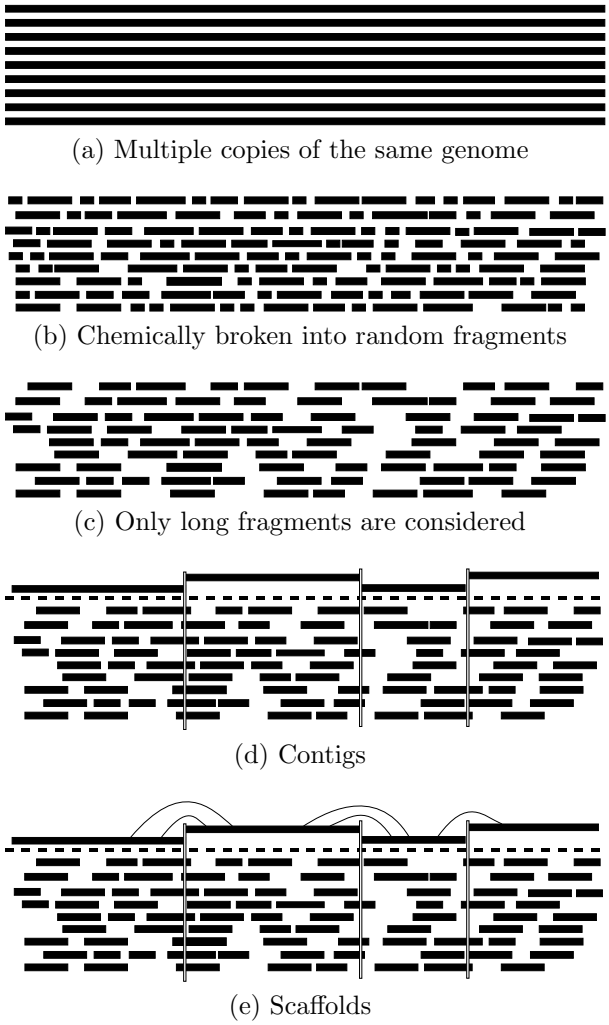
```
@SRR400550.1 717:7:1:1:1189 length=36
NATCGGAAGAGCGGTTCAGCAGGAATGCCGAGACCG
+SRR400550.1 717:7:1:1:1189 length=36
%-6556313446444544444553231463234444
@SRR400550.2 717:7:1:1:1655 length=36
NATCGGAAGAGCGGTTCAGCAGGAATGCCGAGACCG
+SRR400550.2 717:7:1:1:1655 length=36
%.566513455446555555554114444335543
@SRR400550.3 717:7:1:1:1385 length=36
NAGGCTTTGAAAACTAAACAGTGTAAGAGATTGAAC
+SRR400550.3 717:7:1:1:1385 length=36
%099<<<<758659<9757;;;:99969:9::759<
@SRR400550.4 717:7:1:1:1413 length=36
NACATAGTTGTTTTCTTGAACAAGTGTGATATGGTA
+SRR400550.4 717:7:1:1:1413 length=36
%070777:::::::::9988656778868699974
```

Figure 3.23: The first few lines of a FASTQ file.

(Figure 3.24b). The ends of "long" fragments are sequenced using the Sanger method or next generation sequencing (Figure 3.24c), and then the resulting sequences are joined together by a genome assembler to form contigs (Figure 3.24d). Finally, an ordered set of continuous contigs known as *scaffolds* is produced (Figure 3.24e). Scaffolds are produced when the order, orientation and approximate distance between a set of continuous contigs is known with the help of mate pairs. A comparison of several of the existing genome assemblers is shown in Table 3.1. Some of the leading genome assemblers such as Edena, Euler-SR, Velvet, ALLPATHS, ABySS, SOAPdenovo and SGA are discussed below.

Table 3.1: Comparisons of existing genome assemblers.

Genome assembler	**First release**	**Underlying graph**	**Error correction**	**Error removal**	**Mate pair support**	**Read type**
Edena	2008	Overlap graph	No	Yes	No	Short
Euler	2001	De Bruijn graph	Yes	Yes	Yes	Short/Long
Velvet	2008	De Bruijn graph	No	Yes	Yes	Short
ALLPATHS	2008	Unipath graph	No	Yes	Yes	Short
ABySS	2009	De Bruijn graph	No	Yes	Yes	Short
SOAPdenovo	2010	De Bruijn graph	Yes	Yes	Yes	Short
SGA	2011	String Graph	Yes	Yes	Yes	Short



(a) Multiple copies of the same genome

(b) Chemically broken into random fragments

(c) Only long fragments are considered

(d) Contigs

(e) Scaffolds

Figure 3.24: Basic steps of genome assembly [18]. Reconstructed with kind permission from Prof. Shinichi Morishita.

3.5.1 Edena

Edena stands for Exact *DE Novo* Assembler. It was developed by David Hernandez et al. [14] from the Genomic Research Laboratory at the University of Geneva Hospitals. Edena is a simple assembly algorithm that uses overlap graphs. It is based on an overlap layout assembly framework that uses exact matching (i.e., suffix of a read and prefix of another read have exactly the same sequence of base pairs) for overlaps. It supports read lengths of up to 128 base pairs; however its performance is better for small read lengths. Edena requires all reads to be of the same length. If the reads are of different lengths, it

trims the reads so that they are all of the same length. Edena is capable of assembling small bacterial genomes. Earlier versions of Edena support only single-end reads and ignore mate pair information. However, the current version (Version 3) supports paired-end reads and is still under development. Edena has two main steps for genome assembly; overlapping and assembly.

Overlapping Step

At first, duplicated reads are removed from the dataset to reduce memory usage. In the overlapping step, Edena generates a bidirected overlap graph from the input set of reads. A suffix array is used to find all overlapping pairs of reads in the dataset. A *Suffix array* [23, 24] is a data structure that stores sorted suffixes of a string. By default, half of the read length is used as minimum overlap length, but users can specify the minimum overlap length for building the overlap graph. Edena stores the overlap graph in a file (`.ovl`), which is used in the next step.

Assembly Step

The same overlap graph generated in the previous step can be used for different assembly parameters. The assembly step of Edena takes the `.ovl` file and other assembly parameters, such as overlap cutoff m as input from the user. The edges in the original overlap graph built in the previous step having at least m overlapping base pairs are considered to build the overlap graph for minimum cutoff m. The selection of the overlap cutoff is determinant for the success of the Edena assembly. A small overlap cutoff increases the possibility of false overlaps between the reads. On the other hand, a large overlap cutoff increases the number of reads that do not overlap with other reads due to sparsity of coverage (i.e., pairs of reads might not have a long enough common string between them). Transitive edges are then removed from the overlap graph to simplify it. Then some simple cleaning techniques, such as dead-end and bubble removal are used, to re-

duce the size of the graph. *Dead-ends* are short paths in the overlap graph that end at a node of degree 1. *Bubbles* are formed in the graph when two paths start and end at the same pair of nodes and spell similar sequences. Such bubbles are created by either errors or natural variations in the DNA, such as SNPs. Bubbles and dead-ends are often caused by errors in the reads and are easily detected by using statistical analyses. This step produces contigs from the edges of the overlap graph and stores them in a file. The same `.ovl` file can be used to assemble the genome for different parameters. For example, users can specify different overlap cutoff values and use the same graph in the `.ovl` file for assembly.

3.5.2 Eulerian Path Assembly

Eulerian path genome assembly was proposed by Pevzner et al. [37, 38, 39, 40] using a de Bruijn graph approach. In this approach, the goal is to find an Eulerian path in the de Bruijn graph built from the set of input reads. An *Eulerian path* of a graph $G = (V, E)$ is a path that uses every edge $e \in E$ exactly once. There are several variations of Eulerian path approaches. For example, Euler-SR [6] uses the Eulerian approach on short reads (SR) of length about 100 bp, whereas Euler-USR [5] uses the same approach on ultra short reads (USR) of length 20 to 40 bp. De Bruijn graphs are simpler than overlap graphs in repeated regions of the genome, as the number of edges incident on a node in a de Bruijn graph is constant whereas the number of edges incident to a node in an overlap graph can be arbitrarily large.

Eulerian assemblers first break the reads into k-mers and most of them correct errors in the k-mers. If the average number of errors per read d is known, then the number of unique k-mers from the input set can be reduced, by correcting at most d base pairs from each read. The main idea of the correction technique is that an error in a read produces at most $2k$ erroneous k-mers and thus, correcting one base pair reduces the number of unique k-mers.

After correcting the k-mers, the assembler builds the de Bruijn graph. The assembler first finds a set of unambiguous paths P that represent contigs in the de Bruijn graph. The paths $p \in P$ are then used to find an Eulerian path in the de Bruijn graph.

3.5.3 Velvet

Velvet was developed by Daniel Robert Zerbino and Ewan Birney [53] at the European Bioinformatics Institute. Like the Eulerian assembler, Velvet is a de Bruijn graph-based sequence assembler that uses very short paired-end reads of length 25 to 50 bp. It builds a de Bruijn graph from the input reads by breaking the reads into small k-mers. After building the initial de Bruijn graph, it simplifies the graph by merging unambiguous paths to reduce the size of the graph without any loss of information.

Unlike the Eulerian approach, Velvet removes errors after building the de Bruijn graph. First, it removes dead-ends from the de Bruijn graph that are shorter than $2k$ base pairs. Then it removes bubbles with the so-called *tour bus algorithm*, which uses coverage and length thresholds to remove one of the paths in each bubble.

After the unambiguous paths in the de Bruijn graph are simplified, each node stores a string that represent the overlapping k-mers. Nodes in a de Bruijn graph that are near repeated regions in the genome are not possible to contract, as they are not on an unambiguous path, by using short reads. Velvet uses a procedure called *breadcrumb algorithm* to merge strings in the nodes near repeated regions in the graph. A node containing a string of length longer than all the insert sizes is defined as a *long node*. The breadcrumb algorithm first pairs up long nodes in the graph using mate pair information. Two nodes u and v are paired up if there is a mate pair such that one of the reads in the mate pair is in u and the other read is in v. As the nodes in de Bruijn graph represent strings, the nodes have two sides representing the two endpoints of the strings. If a long node pairs up with only one long node in one side, then these two nodes are called unambiguous long nodes. Breadcrumb flags all short nodes that pair up with the

unambiguous long nodes. Then it extends the unambiguous long nodes by traversing as far as possible through the flagged nodes until there is no or more than one options. In best case, a simple path can be found between two unambiguous long nodes through the flagged short nodes, and the path is merged to a single node. Velvet repeats this process until there is no pair of unambiguous long nodes to merge. Finally, the strings in the long nodes are reported as contigs.

3.5.4 ALLPATHS

ALLPATHS [4] is a unipath graph-based algorithm that uses paired-end reads. A *unipath* is an unambiguous path in a de Bruijn graph. Figure 3.25 shows an example of a unipath graph. Node r_5 has two incoming edges and r_8 has two outgoing edges representing branches in the unipath graph. The graph has five unipaths shown in different colors. A *unipath graph* is a graph whose edges are unipaths. Unipaths are found by traversing the nodes until a branching is found. One end of a unipath is arbitrarily named as the left side and the other end is named as the right side. For each unipath in the graph, ALLPATHS finds its neighbors on the left side and also on the right side. If the distance between the left neighbors and the right neighbors is less than a threshold (4 kbp) then the unipath is removed. The remaining unipaths in the graph are *seed unipaths*.

ALLPATHS starts assembly from low copy count seed unipaths called *ideal seeds*. For each ideal seed unipath, ALLPATHS defines its neighborhood based on the distance on each side (10 kb) and constructs two sets of *read clouds*: the *primary read cloud* consists of the reads whose true genomic locations are most likely near the seed unipath (reads on other ideal seed unipaths in the neighbourhood) and the *secondary read cloud* consists of all other short-fragments in the neighbourhood. The reads in the neighborhood of an ideal seed unipath define a local unipath. Local unipaths are joined iteratively using mate pairs in the primary cloud, which in the end yield a sequence graph representing the genome. Then, ALLPATHS simplifies dead-ends, bubbles and loops (edges (u, u) in

the graph that have only one way of traversing) to further simplify the graph.

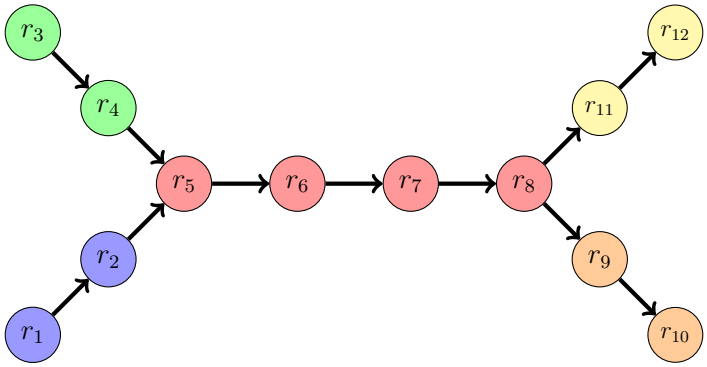


Figure 3.25: Unipaths in a unipath graph shown in different colors.

3.5.5 ABySS

ABySS stands for Assembly By Short Sequences. It was developed by Jared T. Simpson et al. [46] from the Genome Science Center, British Columbia Cancer Agency. ABySS is a parallel assembler for paired-end short reads that uses a de Bruijn graph. It has two major steps. In the first step, all possible k-mers from the input dataset are extracted and the de Bruijn graph is built in a distributed environment. In the second step, mate pair information is used to contract contigs in the de Bruijn graph.

The main idea of the distributed de Bruijn graph representation is that the overlapping k-mers do not need to be physically located on the same processor; rather they can be distributed over different processors. ABySS uses a hash function h to distribute the k-mers over a set of processors $P = \{P_1, P_2, ..., P_p\}$. A numerical value $\{0, 1, 2, 3\}$ is assigned to the bases {`A,C,G,T`} to calculate the base-4 representation x of a k-mer. For example, if values `0,1,2` and `3` are assigned to `A, C, G` and `T`, respectively, then the sequence `TACTG` is represented as $x = 30132_4$. A hash function is then used on x to index the k-mer. If there are p processors, then the k-mer is assigned to the processor $P_{h(x)=x \mod p}$. Since each k-mer can have at most eight possible neighbors in the de Bruijn graph, ABySS uses an 8-bit vector to represent the eight possible neighbors.

Each processor P_i checks for the possible neighbouring k-mers for each of the k-mers assigned to it to build the graph. The dead-ends and bubbles are then removed from the de Bruijn graph and unambiguous paths are simplified. Finally, mate pair information is used to merge ambiguous nodes (i.e., nodes having more than two edges incident on them). A pair of nodes u and v are supported if there is a path (of length close to the insert size) in the graph between two reads in a mate pair that goes through u and v and the nodes are neighbours in the path. If two nodes are supported by m mate pairs (default value of m is 5), then ABySS merges the nodes into a single node. ABySS uses heuristics to limit the number of paths to be searched between reads of a mate pair in the dense parts of a de Bruijn graph. Nodes in the resulting graph represent contigs. For each contig c_i, ABySS generates a set C_i of contigs that are paired with c_i by mate pairs (one read of a mate pair is in c_i and the other read is in the contig $c \in C_i$). The graph is then searched for a single unique path between c_i and each contig $c \in C_i$. If there is only one contig $c \in C_i$ having a unique path with c_i, then ABySS contracts the path to a single node. This process is repeated until no contigs are merged and scaffolds are reported by ordering the contigs.

3.5.6 SOAPdenovo

Short Oligonucleotide Alignment Program *de novo* (SOAPdenovo) [22] is a short read assembler that can assemble human-sized genomes. It was specially designed to assemble Illumina short reads. SOAPdenovo uses a de Bruijn graph approach.

Before building the de Bruijn graph, SOAPdenovo corrects errors based on the frequencies of each of the k-mers in the dataset. This error correction technique is similar to the error correction technique used in the Eulerian path assembly discussed in Section 3.5.2. In high coverage datasets, each k-mer appears multiple times in the reads, whereas k-mers containing random sequencing errors have very low frequencies. The frequency information of each of the k-mers is used to correct the input datasets as fol-

lows. SOAPdenovo first uses a hash table to store the frequencies of all the k-mers in the dataset. It then takes the k-mers with high frequencies and checks the frequency of the eight possible neighbouring k-mers (Four overlapping with the suffix and four overlapping with the prefix), if exist in the hash table, to find potential erroneous k-mers with low frequencies. For each potentially erroneous k-mer, it checks the impact of changing the first/last (depending the location of new base pair added to get the k-mer) base pair of the k-mer to the other three bases and the changes are made if all k-mers' frequencies from that region of the dataset are increased after changing the base. The process of error correction is done with a parallel algorithm.

A de Bruijn graph is then built from the corrected k-mers and then bubbles and dead-ends are removed. SOAPdenovo removes dead-ends shorter than $2k$ base pairs. Unambiguous paths are merged to nodes that represent the strings in the path. In SOAPdenovo, a pair of nodes is said to be supported by mate pairs if at least three mate pairs agree on the order and orientation of the strings in the nodes. A *contig linkage graph* is then built where the nodes represent the strings in the nodes (contigs) of the original de Bruijn graph and edges represent support between a pair of contigs. Assuming that the insert size of the mate pair library or distance between the reads in a mate pair is normally distributed, SOAPdenovo tries to estimate the distance between every pair of supported contigs based on the location of the reads of a mate pair on the edges. Transitive edges are then removed from the contig linkage graph. Repeat contigs are discovered by looking at the linkages. Contigs having multiple incoming and outgoing edges in the contig linkage graph are called *repeat contigs*. The repeat contigs and their links are marked. Then the unambiguous paths in the contig linkage graph are joined to form scaffolds.

SOAPdenovo supports multiple insert libraries, each library with a different insert size. For multiple insert libraries, it merges the contigs in the graph by first using short insert size library and then larger insert size libraries. The majority of the gaps in the

scaffolds are in the repeat contigs that were marked previously, as there is no unique path through the repeat contigs. SOAPdenovo tries to fill in the gaps by looking at the collection of mate pairs that have one read aligned in the contig and the other read in the repeated region. Then a local assembly for the collected reads is done.

3.5.7 SGA

The String Graph Assembler (SGA) [45] uses memory efficient data structures for genome assembly. SGA constructs an assembly string graph [35] to represent the reads. A string graph can be built as follows. First duplicate reads, contained reads (reads that are substrings of other reads) are removed from the dataset and an overlap graph is built. Then transitive edges are removed from the overlap graph. Non transitive edges in an overlap graph are called *irreducible edges*. A *string graph* is defined as the subgraph of the overlap graph that contains only irreducible edges.

The construction of an overlap graph from the set of given reads R is not memory efficient, since most of the edges in an overlap graph are transitive edges and they will be removed later. SGA uses a memory efficient way to build the string graph without building the overlap graph first. It uses an FM-index [11] to directly compute the set of irreducible edges for a given set of reads [44]. An *FM-index* is a data structure that stores compressed sorted suffixes of an input string using the Burrows-Wheeler compression algorithm [3]. Like a suffix array, an FM-index allows fast substring queries, but uses less memory than a suffix array. SGA arbitrarily divides the reads in a dataset into subsets and uses a distributed algorithm to build an FM-index for each subset of reads. Then the pairs of intermediate FM-indices are repeatedly merged together using the BWT merging algorithm [10] to get a single FM-index for the dataset. Indexing a subset of reads in SGA can easily be implemented in a parallel environment. Reads in SGA are corrected based on the k-mer frequencies and approximate overlap between reads. When two sequences overlap with insertions, deletions or substitutions, the overlap is

called *approximate overlap* and the edit distance of the overlap is the total number of mismatches, insertions and deletions. SGA tries to correct the reads to minimize the sum of edit distances of all overlapping reads in the string graph.

Each read in the string graph is represented by a node. Most of the reads in the string graph will have only two neighbours, one overlapping with a prefix and the other overlapping with a suffix of the read. The majority of the reads in the string graph are *simply connected*, meaning that they are on a path of the graph without any branching. Such unambiguous paths are simplified to single edges to reduce the size of the graph. SGA locally builds the string graph, starting from an arbitrary read $r \in R$. The irreducible edges incident on a read r are merged if r has only two neighbours, one in each side of the string in r. SGA iteratively discovers irreducible edges in the graph and reduces the nodes that are simply connected. Each edge in the simplified string graph represents a contig.

Mate pair information is then used to bridge contigs to form scaffolds by a program called *scaffolder*. The scaffolder builds a contig linkage graph that represents relationships between contigs using mate pair support and a statistical analysis is used to estimate the copy count of the contigs. Information about the support (e.g., number of mate pairs supporting the contigs, gap size between the contigs) is stored in the edges of the contig linkage graph. For each contig in the contig linkage graph with more than one link in a particular direction, it checks whether the contigs have a consistent ordering with each pairwise distance estimate and no adjacent pair of contigs in the ordering has an overlap greater than 400 base pairs. The scaffolder then removes ambiguous edges from the contig linkage graph, where the contigs cannot be ordered consistently. Terminal vertices (i.e., vertices having only one edge in the graph) are then identified and all paths between all pairs of terminal vertices in the contig linkage graph are computed and the path with longest sequence is retained as a scaffold. Experimental results [45] show that SGA is more memory efficient than other assemblers. However, SGA is not time efficient.

Chapter 4

PEGASUS

In this chapter we give a detailed description of our assembler, Paired-End Genome ASsembler Using Short-sequences (PEGASUS). Some of the existing algorithms [14, 34, 45] for genome assembly use an overlap graph-based approach. Some of these algorithms first generate a huge overlap graph from the entire collection of reads and then simplify it to reduce its size.

PEGASUS has three major improvements over other existing algorithms for genome assembly. First, it does not create a huge overlap graph for the reads; instead, it directly generates a simplified graph by reducing transitive edges while building the overlap graph. This greatly reduces the amount of memory needed. Second, we use a new technique to accurately estimate the copy counts of the reads by combining a log odds ratio analysis with a convex minimum cost flow computation on the overlap graph, as described in Section 4.8. Third, current algorithms for genome assembly [14, 46, 53] use very simple transformations to simplify the overlap graph and reduce its size. We noticed that we can perform further simplifications of the overlap graph that go beyond what other assemblers do. These reductions have a dramatic effect on the final size of the overlap graph. In the final step of PEGASUS, we use mate pair information and the copy counts of the reads to unscramble long paths in the overlap graph. We give more details about each of the

steps of PEGASUS in the following sections.

4.1 Overview of PEGASUS

Input reads are first corrected with RACER (Rapid and Accurate Correction of Errors in Reads) [17]. Then, a bidirected overlap graph is built from the input dataset using a hash table. While building the overlap graph, transitive edges are removed to reduce the size of the overlap graph. We then contract simple paths and remove dead-ends and bubbles. A convex cost function and log odds ratio analysis is used to compute the parameters for a minimum cost flow problem on the overlap graph, which is solved to estimate the copy counts of the reads. We also use the flow computed in the previous step to further simplify the overlap graph. At the end, we use mate pair information to try to build longer contigs. Some of the above steps are repeated several times.

4.2 Error Correction

Real datasets contain errors that are difficult to handle by genome assemblers. Many genome assemblers have a separate preprocessing step where the errors are corrected. PEGASUS uses RACER to correct the reads before using them for assembly. All the datasets used to test PEGASUS were corrected with RACER.

4.3 Overlap Graph Construction

To build the overlap graph from the input dataset, we need to find all the pairs of reads that overlap with each other. PEGASUS takes a parameter *minOverlap* as input which denotes the minimum number of bases that must match for adding an edge between two overlapping reads. If there are n unique reads in the input dataset then we need to compare $O(n^2)$ pairs of reads to find all overlapping pairs of reads, which is not practical.

To reduce the number of pairs of reads to compare, PEGASUS uses a hash table where prefixes and suffixes of each read r and its reverse complement $\bar{r}$ are stored. Observe that if two reads r_1 and r_2 overlap by at least $minOverlap$ base pairs, then there is a suffix or a prefix of r_1 of length at least $minOverlap$ base pairs that matches with a prefix or a suffix of r_2. Therefore, we store a prefix and a suffix of length $h = min\{minOverlap, 64\}$ bases of each read and its reverse complement into the hash table. Strings of length at most 64 base pairs are stored in the hash table because string of length 64 base pairs can be stored in two 64-bit numbers (each base is represented by 2 bits). Storing strings of length more than 64 base pairs increases the complexity of the hash function and decreases the performance of storing and searching strings in the hash table.

4.3.1 Hash Table

For each read r in the dataset R, we store in the hash table a prefix and a suffix of r and its reverse complement $\bar{r}$, so if there are n reads in R then we store $4n$ suffixes and prefixes. The hash table size is set to a prime number p such that $p > 8n$ to reduce the number of hash misses in the hash table. For efficiency we store each read as an array of 8-bit integers. Each base pair is represented by two bits in the array (`A=00`, `C=01`, `G=10`, `T=11`). Representing the base pairs in this way helps to compare strings efficiently by comparing numbers: A comparison of 4 base pairs is done by comparing two 8-bit numbers. Similarly, comparing 64 base pairs is done by comparing two pairs of 64-bit numbers. This bit representation reduces the memory usage and it is efficient to hash the reads by using their corresponding numerical values. The hash table construction is shown in Algorithm 1.

4.3.2 Inserting Edges in the Overlap Graph

Let l be the length of each read. For each read r in R, we take all its substrings ${S_r}^1, {S_r}^2, ..., {S_r}^{l-h}$ of length h bases and search for them in the hash table. This re-

Algorithm 1 buildHashTable(R,$minOverlap$): Building the hash table.

Input: Set $R = \{r_1, r_2, ..., r_n\}$ of reads and minimum overlap length $minOverlap$
Output: Hash table $hashTable$
Create an empty $hashTable$
$h \leftarrow min\{64, minOverlap\}$ ▷ Length of the strings to hash
for each read $r \in R$ **do**
    $pr \leftarrow$ first h symbols of r
    $sr \leftarrow$ last h symbols of r
    $p\bar{r} \leftarrow$ first h symbols of $\bar{r}$
    $s\bar{r} \leftarrow$ last h symbols of $\bar{r}$
    Store pr, sr, $p\bar{r}$ and $s\bar{r}$ in the $hastTable$
return $hashTable$

turns a set S of reads that contains the substrings ${S_r}^i$ as prefixes or suffixes of the reads or their reverse complements. We only need to check reads in S to find all reads that overlap with r and, hence, to determine how many edges will be incident on the node of the overlap graph representing r. For each read $r_i \in S$, we know that r and r_i have a common substring of length at least h. To add an edge between the nodes representing r and r_i, we need to verify that the overlap between the reads extends to the end or to the beginning of r. Consider the reads in Figure 4.1 and minimum overlap length of 10 bp. The prefixes and suffixes of u and v (shown in blue) of length 10 bp are stored in the hash table. To determine whether we need to add an edge between the nodes representing u and v, we take each substring of u of length 10 bp and search it in the hash table. For the substring `AGCATTTACG` of u, the search returns the read v as v has `AGCATTTACG` as a prefix. Therefore, we know that the substring for length 10 bp of u starting at position 6 is a prefix of v as shown in Figure 4.2. For an edge to be added between u and v we need to compare the remaining parts of u and v (shown in green in Figure 4.2) to verify that they match. Note that the same overlap is discovered again when we search for the substrings of v in the hash table, but only one edge is added between u and v.

```
Read u:  AGCTAAGCATTTACGATAGCCGATAGCTAAATTAC
Read v:  AGCATTTACGATAGCCGATAGCTAAATTACGTTAT
```

Figure 4.1: Hash string of the reads.

```
Read u:  AGCTAAGCATTTACGATAGCCGATAGCTAAATTAC
Read v:        AGCATTTACGATAGCCGATAGCTAAATTACGTTAT
```

Figure 4.2: Substring match using hash table.

For each pair of reads u, v in R, we insert an edge (u, v) in the overlap graph G, if either u and v, u and $\bar{v}$, or $\bar{u}$ and v overlap by at least *minOverlap* base pairs. Our overlap graph construction algorithm inserts the edges in the graph in such a way that we can remove the transitive edges incident on the current read r after inserting all the edges incident on r, as explained in the next section. The overlap graph for the set of reads in Figure 4.3 is shown in Figure 4.5. The overlap graph is bidirected, which means that arrowheads at the endpoints of an edge $e = (r_1, r_2)$ represent the orientations of the overlapping reads represented by the nodes r_1 and r_2. For example, consider reads r_1 and r_2 in Figure 4.3 and assume that $minOverlap = 32$. Since the last 34 base pairs of r_1 are identical to the first 34 base pairs of $\bar{r_2}$ then a forward-reverse bidirected edge is added from the node representing r_1 to the node representing r_2. Since r_1 and r_2, and $\bar{r_1}$ and r_2 do not overlap in at least *minOverlap* base pairs, no other edges are added between these nodes. Note that, two reads can overlap in more than one way and hence there can be multiple edges between the nodes representing them in the overlap graph.

4.3.3 Transitive Reduction

Recall our discussion on transitive edges in 3.2.5. If there are three overlapping edges $e_1 = (u, v)$, $e_2 = (u, w)$ and $e_3 = (w, v)$ in the overlap graph that form a triangle Δuvw and the string spelled by the edge e_1 and the string spelled by the path $p = \{e_2, e_3\}$ are the same, then we call e_1 a transitive edge. For example, in Figure 4.5 the string spelled by the edge (r_1, r_3) is AGCTAAGCATTTACGATAGCCGATAGCTAAATTACGTT and this is equal to the string spelled by the path $\{r_1, r_2, r_3\}$. So, the edge (r_1, r_3) is a transitive edge. Note that, any of the three edges in a triangle as above can be a transitive edge, depending on the orientation of the edges. As explained above, information stored in a transitive edge

(u, v) is redundant, since the string spelled by the transitive edge is also stored in the path of length two between u and v. Therefore, transitive edges can be removed from the overlap graph without losing any information.

Unlike other genome assemblers that first build the entire graph and then remove the transitive edges, we remove transitive edges while building the overlap graph. A linear time algorithm for removing transitive edges was proposed by Gene Myers [35], but it requires the whole graph to be built. Myers algorithm for exact overlaps in the graph is shown in Algorithm 2. To remove transitive edges incident on a node v, Myers algorithm first marks all the neighbours of v as *inplay*. Then for each of the neighbours w of v in increasing order of the length of the string spelled by the edge (v, w), it marks every neighbour of w as *eliminated* that was already marked as *inplay*. At the end all edges (v, x) are marked for removal, for each node x that is marked as *eliminated*. Transitive edges are removed from the graph only after all the nodes in the graph have been processed.

We use a similar algorithm as Myers; however, we modified it so it can remove transitive edges as the overlap graph is being built. The algorithm is described in Algorithm 6. Transitive edges incident on a read r are removed (Algorithm 5) after marking the transitive edges incident on r and on its neighbours (Algorithm 4). Therefore, to remove all transitive edges incident on a node r we need to build that part G_r of the transitive graph that includes all nodes at distance at most 3 from r. Then we can use Meyers algorithm on G_r to remove all the transitive edges incident on r. Observe that we need to mark the transitive edges incident on the neighbours of r before removing the transitive edges incident on r. Because the transitive edges incident on the current node r might be required to discover some of the transitive edges incident on its neighbours; it is not possible to discover some of the transitive edges incident on the neighbours of r once all the transitive edges incident on r are removed.

PEGASUS uses queue to store reads that have not yet been processed. We first

Algorithm 2 Linear time transitive edge reduction [35].

Input: Overlap graph $G = (V, E)$
Output: Transitively reduced overlap graph
for each $v \in V$ **do**
    $mark[v] \leftarrow vacant$
    for each $(v, w) \in E$ **do**
        $reduce[(v, w)] \leftarrow false$
for each $v \in V$ **do**
    for each $(v, w) \in E$ **do**
        $mark[w] \leftarrow inplay$
    for each $(v, w) \in E$ in increasing order of length of the string spelled **do**
        if $mark[w] = inplay$ **then**
            for each $(w, x) \in E$ in increasing order of length of the string spelled **do**
                if $mark[x] = inplay$ **then**
                    $mark[x] \leftarrow eliminated$
    for each $(v, w) \in E$ **do**
        if $mark[w] = eliminated$ **then**
            $reduce[(v, w)] \leftarrow true$ ▷ Mark for transitive reduction
        $mark[w] \leftarrow vacant$
for each edge $e \in E$ **do**
    if $reduce[e] = true$ **then**
        Remove e from E ▷ Remove the transitive edge e
return G

explore an arbitrary read r (i.e., insert in the overlap graph the node corresponding to r and all edges incident on this node) and put its neighbours in the queue. Then we explore the neighbours of r and the neighbours' neighbours, if not explored already. We next mark the transitive edges incident on r and on its neighbours (Algorithm 4), but remove only the transitive edges incident on r (Algorithm 5). To avoid exploring the same node multiple times we label each node as *unexplored*, *explored* or *marked*. When we are done with the current node, we pick the next read from the queue. If the queue is empty, then we pick another unexplored read from the dataset and repeat the above steps until all reads in R are processed. The algorithm for constructing the overlap graph is shown in Algorithm 6.

Algorithm 3 exploreRead($G = (V, E), minOverlap, hashTable, r$)**:** Insert in the overlap graph all edges incident on r .

```
1: Input: Overlap graph G = (V, E), hashTable, minOverlap and a read r
2: Output: Overlap graph G = (V, E) after inserting all edges incident on r
3: h ← min{64, minOverlap}
4: for each read r ∈ R do
5:     for each substring s of length h of r do
6:         list ← hashTable.get(s)          ▷ hashTable.get() returns the list of reads
7:         for each read r' ∈ list do
8:             if flag[r'] = unexplored and overlapLength(r, r') ≥ minOverlap then
9:                 Add (r, r') to G
10: return G
```

```
Read r1  :  AGCTAAGCATTTACGATAGCCGATAGCTAAATTAC
Read r2  :  CGTAATTTAGCTATCGGCTATCGTAAATGCTTAGC
Read r3  :  AACGTAATTTAGCTATCGGCTATCGTAAATGCTTA
Read r4  :  GCATTTACGATAGCCGATAGCTAAATTACGTTATA
Read r5  :  GTATAACGTAATTTAGCTATCGGCTATCGTAAATG
Read r6  :  ATTTACGATAGCCGATAGCTAAATTACGTTATACT
Read r7  :  TTTACGATAGCCGATAGCTAAATTACGTTATACTC
Read r8  :  ATATAACGTAATTTAGCTATCGGCTATCGTAAATG
Read r9  :  ATTTACGATAGCCGATAGCTAAATTACGTTATATA
Read r10 :  CTATATAACGTAATTTAGCTATCGGCTATCGTAAA
```

Figure 4.3: Set of given input reads.

Algorithm 4 markTransitiveEdges($G = (V, E), r$): Mark transitive edges incident on read r.

Input: Overlap graph $G = (V, E)$ and a node $r \in V$
Output: Overlap graph $G = (V, E)$ after marking the transitive edges incident on r
for each neighbour n of r **do**
    $mark[n] \leftarrow inplay$
for each neighbour n of r in increasing order of length of string spelled by (r, n) **do**
    if $mark[n] = inplay$ **then** ▷ For each *inplay* neighbour
        for each neighbour nn of n **do**
            if edges (r, n) and (n, nn) have opposite orientations in node n **then**
                if $mark[nn] = inplay$ **then**
                    $mark[nn] \leftarrow eliminated$
for each neighbour n of r **do**
    if $mark[n] = eliminated$ **then**
        Mark edge (r, n) as *transitive*
return G

Algorithm 5 removeTransitiveEdges($G = (V, E), r$): Remove from the overlap graph $G = (V, E)$ transitive edges incident on read r

Input: Overlap graph $G = (V, E)$ and a node $r \in V$
Output: Overlap graph $G = (V, E)$ after removing transitive edges incident on r
for each neighbour n of r **do**
    if edge (r, n) is marked as *transitive* **then**
        Remove (r, n) from G
return G

Algorithm 6 buildOverlapGraph(R,$minOvelap$): Build the overlap graph $G = (V, E)$.

Input: Set $R = \{r_1, r_2, ..., r_n\}$ of reads, $minOverlap$
Output: Overlap graph $G = (V, E)$
$hashTable \leftarrow buildHashTable(R, minOverlap)$
$V \leftarrow \emptyset$ ▷ Set of vertices
$E \leftarrow \emptyset$ ▷ Set of edges
for each read $r \in R$ **do**
    $flag[r] \leftarrow unexplored$ ▷ All reads marked as unexplored
$queue \leftarrow \emptyset$
for each read $r \in R$ **do**
    if $flag[r] = unexplored$ **then**
        $exploreRead(G, minOverlap, hashTable, r)$ ▷ Insert edges incident on r
        $flag[r] \leftarrow explored$
        $enqueue(r)$ ▷ Put r in the queue
        while $queue \neq \emptyset$ **do** ▷ This explores a connected component in the graph
            $r \leftarrow dequeue()$
            if $flag[r] = explored$ **then**
                for each unexplored neighbour u of r **do** ▷ Explore all neighbours
                    $exploreRead(G, hashTable, minOverlap, u)$ ▷ Edges incident on u
                    $flag[u] \leftarrow explored$
                    $enqueue(u)$
                $markTransitiveEdges(G, r)$ ▷ Mark transitive edges incident on r
                $flag[r] \leftarrow marked$
            if $flag[r] = marked$ **then**
                for each $explored$ neighbour u of r **do**
                    for each $unexplored$ neighbour v of u **do** ▷ Explore v
                        $exploreRead(G, hashTable, minOverlap, v)$
                        $flag[v] \leftarrow explored$
                        $enqueue(v)$
                $markTransitiveEdges(G, u)$ ▷ Mark transitive edges incident on u
                $flag[u] \leftarrow marked$
                $removeTransitiveEdges(G, r)$ ▷ Transitive edges incident on r
return G

```
Read r1    :  AGCTAAGCATTTACGATAGCCGATAGCTAAATTAC
Read r̄2    :   GCTAAGCATTTACGATAGCCGATAGCTAAATTACG
Read r̄3    :     TAAGCATTTACGATAGCCGATAGCTAAATTACGTT
Read r4    :        GCATTTACGATAGCCGATAGCTAAATTACGTTATA
Read r̄5    :         CATTTACGATAGCCGATAGCTAAATTACGTTATAC
Read r6    :          ATTTACGATAGCCGATAGCTAAATTACGTTATACT
Read r7    :           TTTACGATAGCCGATAGCTAAATTACGTTATACTC
Read r̄8    :         CATTTACGATAGCCGATAGCTAAATTACGTTATAT
Read r9    :          ATTTACGATAGCCGATAGCTAAATTACGTTATATA
Read r̄10   :           TTTACGATAGCCGATAGCTAAATTACGTTATATAG
                Read Length l  :  35 base pairs
                  minOverlap   :  32 base pairs
```

Figure 4.4: Overlapping reads.

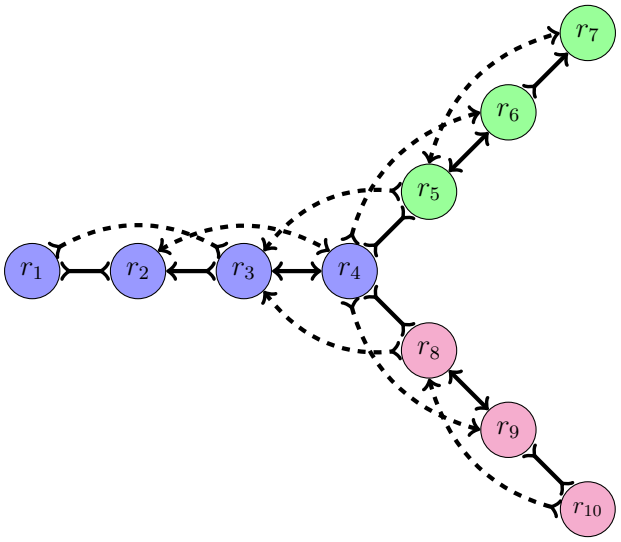


Figure 4.5: Overlap graph with 10 reads.

4.4 Contracting Composite Paths

Since most of the reads appear only once in the genome, in the transitively-reduced overlap graph most of the nodes have only two neighbours; for each of these nodes, one of the incident edges is an in-edge and the other one is an out-edge. However, some nodes in the overlap graph can have more than two neighbouring nodes. These nodes either correspond to reads located at the ends of repeated regions of the genome (see Section 3.1.4) or to reads that contain errors. In Figure 4.6, nodes r_2, r_3, r_5, r_6, r_8 and r_9 have one in-edge and one out-edge. For each node $v \in V$ in the overlap graph

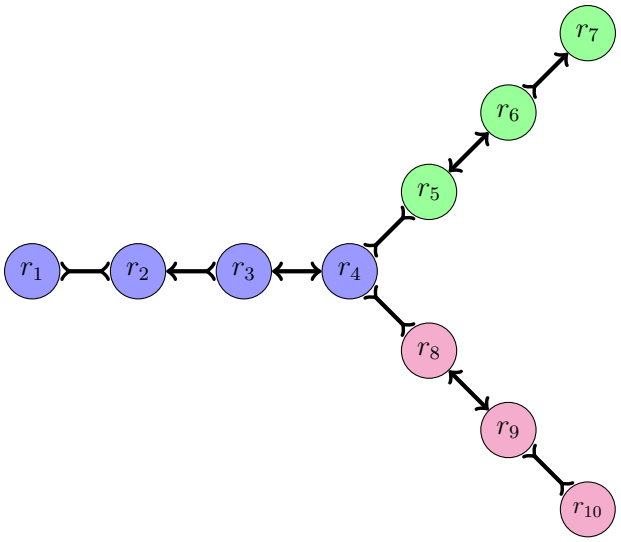


Figure 4.6: Overlap graph after transitive edge reduction.

with only one in-edge $e_1 = (u, v)$ and one out-edge $e_2 = (v, w)$, we remove node v and the edges e_1 and e_2 and insert an edge $e_3 = (u, w)$ in the overlap graph. The edge e_3 stores the information formerly stored in node v and edges e_1 and e_2. We repeatedly remove these nodes of degree two from the overlap graph until there are no more of them. This procedure is called *composite path contraction* in PEGASUS. Figure 4.7 shows the overlap graph after contracting composite paths in the overlap graph of Figure 4.6. Path $p = \{r_1, r_2, r_3, r_4\}$ in Figure 4.6 is contracted into a single edge (r_1, r_4). Note that, in the path p, the reads r_2 and r_3 are entered through an out-edge, hence, we store the reverse complements $\bar{r_2}$ and $\bar{r_3}$ of the reads r_2 and r_3 in the composite edge (r_1, r_4) to represent the orientations of the overlapping reads in p. The pseudocode for the algorithm for performing composite path contraction is shown in Algorithm 7.

Algorithm 8 shows the procedure for merging two edges in the overlap graph. Note that the orientations of the reads v and w on the resulting edge $e = (u, x)$ depend on the orientations of the edges $e_1 = (u, v)$ and $e_2 = (w, x)$.

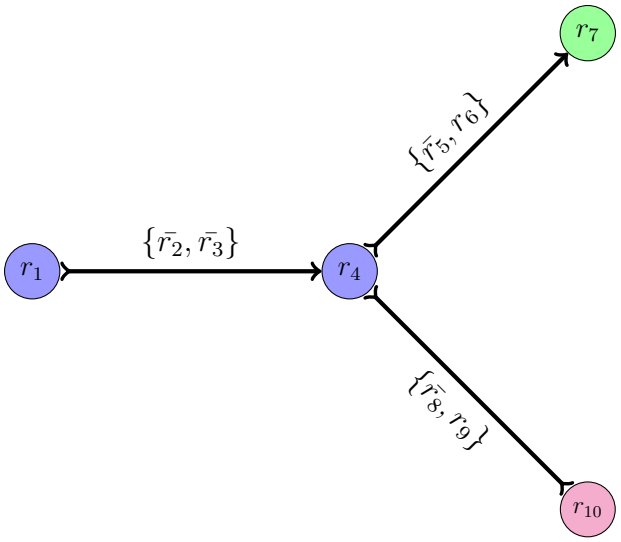


Figure 4.7: Overlap graph resulting after composite path contraction.

Algorithm 7 contractCompositePaths($G = (V, E)$): Composite path contraction.

1: **Input:** Overlap graph $G = (V, E)$
2: **Output:** Overlap graph after contracting composite paths
3: **for** each node $v \in V$ **do**
4: **if** v has exactly one in-edge $e_1 = (u, v)$ and one out-edge $e_2 = (v, w)$ **then**
5: $mergeEdges(G, e_1, e_2)$
6: **return** G

4.5 Error Removal

The error correction programs used to correct reads are never perfect, so the corrected datasets may still contain errors. After building the simplified graph, PEGASUS detects and removes erroneous reads by looking at the topology of the overlap graph. Most of the erroneous reads do not overlap with any other reads, and so are not added to the overlap graph since we insert new nodes in the overlap graph only if there is at least one

Algorithm 8 mergeEdges($G = (V, E), e_1, e_2$): Merge pair of edges.

1: **Input:** Overlap graph $G = (V, E)$ and two edges $e_1 = (u, v)$ and $e_2 = (w, x)$
2: **Output:** Merge the edges e_1 and e_2 into a single edge e
3: Add edge $e = (u, x)$ to G
4: Put the ordered reads of e_1 and e_2 in e
5: Remove e_1 and e_2 from G
6: **return** G

edge incident on them. However, sometimes by chance the erroneous reads overlap with other reads. We use simple techniques such as bubble and dead-end removal to identify and discard erroneous reads from the overlap graph. Dead-end and bubble removal techniques are also used by other genome assemblers such as Edena [14], ABySS [46], and SOAPdenovo [22].

4.5.1 Dead-End Removal

Most of the erroneous reads that overlap with other reads in the dataset create dead-ends, short paths in the overlap graph that end at a node of degree one. Dead-ends are short because it is very unlikely that reads with errors in them form a long path in the overlap graph. Figure 4.8 shows an example of a dead-end in the overlap graph caused by erroneous reads r_{11} and r_{12} (before contracting composite paths). We can remove the edges (r_5, r_{11}) and (r_{11}, r_{12}) and delete the nodes r_{11} and r_{12} to get rid of the erroneous reads from the overlap graph. Dead-end removal of PEGASUS is shown in Algorithm 9. Note that, composite paths are contracted before dead-end removal, hence, each dead-end is simplified to a single composite edge.

A composite edge containing more than 5 reads is not considered as a dead-end. Assuming uniform distribution of the reads, if a composite edge e has 5 reads in it, then the probability that all the reads in e have an error in exactly the same position is $\left(\frac{p}{3}(1-p)^{l-1}\right)^5$, where the probability that a base pair is wrong in a read is p and the read length is l . Note that the erroneous base pair must be the same for all the reads in order to overlap. Error rate of Illumina (`www.illumina.com`) is about 1%, meaning that the probability of an error in a base pair is .01. So, the probability of 5 erroneous reads to overlap is very close to 0.

Algorithm 9 removeDeadEnds($G = (V, E)$): Dead-end removal from the overlap graph.

Input: Overlap graph $G = (V, E)$
Output: Overlap graph after removing dead-ends
for each node $u \in V$ **do**
    $inDegree \leftarrow 0$ ▷ Number of in-edges
    $outDegree \leftarrow 0$ ▷ Number of out-edges
    for each neighbour v of u **do**
        if (u, v) has more than 5 reads in it **then** ▷ Unlikely to have > 5 reads with errors forming a path
            $inDegree \leftarrow 0$
            $outDegree \leftarrow 0$
            break
        if edge (u, v) is an in-edge of u **then**
            $inDegree \leftarrow inDegree + 1$
        else
            $outDegree \leftarrow outDegree + 1$
    if $inDegree = 0$ **and** $outDegree > 0$ **then**
        Remove u and all its edges from G
    if $inDegree > 0$ **and** $outDegree = 0$ **then**
        Remove u and all its edges from G
return G

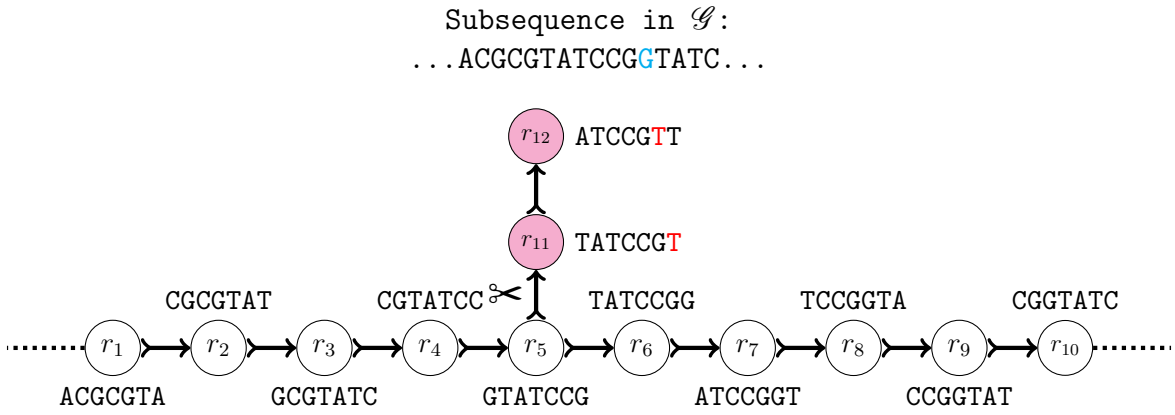


Figure 4.8: A dead-end in the overlap graph caused by erroneous reads.

4.5.2 Bubble Removal

Occasionally errors appear in certain positions of the reads creating short paths that overlap at both ends with other parts of the overlap graph. Such erroneous paths are called bubbles (see Figure 4.9 for an example). An erroneous path in a bubble will have fewer reads in it than the other correct path. We can detect and remove erroneous paths in the bubbles from the overlap graph, by comparing the number of reads in the two paths of the bubbles. For the example in Figure 4.9, the string ACGCGTAGCCGGT spelled by the path $p_1 = \{r_1, r_{11}, r_{12}, r_9\}$ is 13 bp long and the string ACGCGTATCCGGT spelled by the path $p_2 = \{r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8, r_9\}$ is also 13 bp long, but they differ in only one base pair. There are only two reads in path p_1 and seven reads in path p_2. In general, the number of reads in one of the paths in a bubble cause by erroneous reads is significantly smaller than the number of reads in the other path. If the two paths in a bubble have similar strings (more than 90% similar) and one of the paths has fewer reads (at most half of the reads in the other path), then the path with fewer reads is removed from the bubble. We choose 90% similarities between the stings, because if they are less than 90% similar then it is more likely that the path is not because of errors; reads with more errors are less likely to form a path in the graph that overlaps in both ends with other correct reads.

Bubble-like structures can also be present in the overlap graph due to single nucleotide

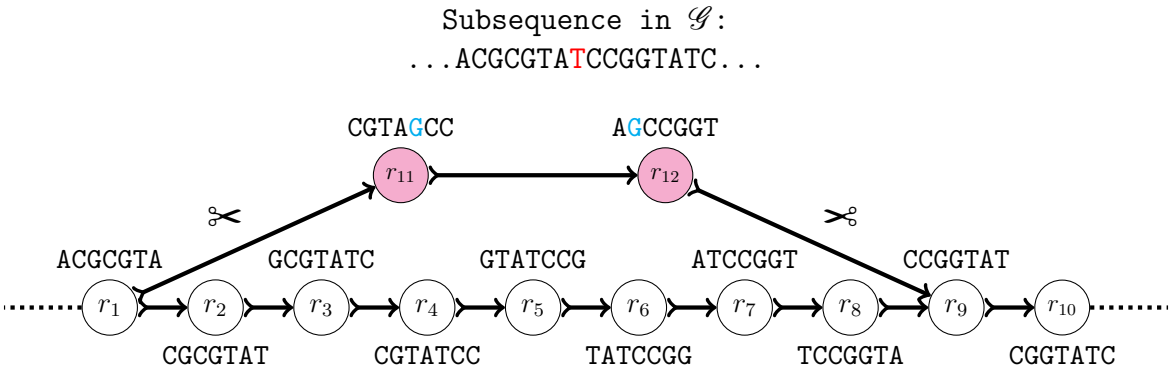


Figure 4.9: A bubble in the overlap graph caused by erroneous reads.

polymorphism (SNP). In case of SNPs, both paths in the bubble have a similar number of reads in them. To differentiate between the bubbles caused by SNPs and the bubbles caused by errors, we compare the number of reads in both the paths of the bubbles. Bubbles caused by errors appear less frequently in the overlap graph than dead-ends, as the erroneous reads have to overlap in both directions. The algorithm for bubble removal is shown in Algorithm 10. Note that before bubble removal, the two paths in a bubble are simplified to composite edges after contracting composite paths. Figure 4.10 shows the overlap graph after removing one of the paths in the bubble in Figure 4.9.

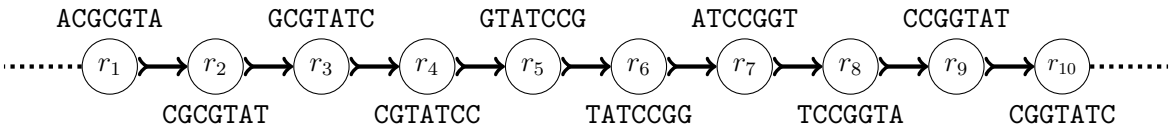


Figure 4.10: Overlap graph after removing the bubble in Figure 4.9.

Algorithm 10 removeBubbles($G = (V, E)$): Bubble removal from the overlap graph.

Input: Overlap graph $G = (V, E)$
Output: Overlap graph after removing bubbles
for each pair of edge $e = (u, v)$ **and** $e' = (u, v) \in G$ with the same endpoints **do**
    if string spelled by $e \approx$ string spelled by e' **then** ▷ More than 90% similar
        if number of reads in $e \leq \frac{1}{2}$ number of reads in e' **then**
            Remove e from G
        else if number of reads in $e' \leq \frac{1}{2}$ number of reads in e **then**
            Remove e' from G
return G

4.6 Genome Size Estimation

In *de novo* genome assembly, the size of the genome from which the reads were sampled is unknown. We estimate the genome size by using a technique proposed by Gene Myers [35]. If the length of the genome is L, there are n reads in the dataset and the read length is l, then assuming that the reads were sampled uniformly from the genome, each position of the genome is sampled on average $\frac{nl}{L}$ times. Similarly, if k reads were sampled from the part of the genome representing an edge e in the overlap graph and the length of the string spelled by e is d, then assuming that the edge represents a sequence that appears only once in the genome, each position of the sequence represented by e is sampled on average $\frac{kl}{d}$ times. If d is long enough, the above two average sampling rates will be very similar, i.e.,

$$\begin{aligned} \frac{nl}{L} &\approx \frac{kl}{d} \\ \therefore L &\approx \frac{dn}{k} \end{aligned} \tag{4.1}$$

To estimate the value of L we first compute the sum of the lengths of all the long edges (edges spelling strings of length at least 1000 bp) and the total number of the reads sampled from the part of the genome represented by these edges. Under the assumption that all sequences represented by the long edges appear once, a good estimation of the genome size can be computed by using Equation 4.1. We choose the length at least 1000 bp as a long edge because significantly more subsequences of length longer than 1000 bp are in unique regions of the genome than in repeated regions.

Note that this estimation of L could be wrong, because some of the sequences represented by the long edges considered in the previous step could appear multiple times in the genome. To identify these edges we compute for every long edge e the ratio of the probability that the sequence represented by e appears once in the genome and the probability that it appears twice in the genome. If this ratio is very large, then most

likely the sequence represented by e appears only once in the genome. These probabilities are computed as follows. The probability that the reads in an edge e appear only once in the genome is approximately $\frac{\left(\frac{dn}{L}\right)^k}{k!}e^{-\frac{dn}{L}}$ (for details see Section 4.8.3), where L is the length of the genome (approximated in the previous step), n is the number of reads in the dataset, k is the number of reads sampled from the sequence spelled by the string in e and d is the length of the string spelled by e. Similarly the probability that the reads in the edge e is present twice in the genome is approximately $\frac{\left(\frac{2dn}{L}\right)^k}{k!}e^{-\frac{2dn}{L}}$. If the probability of the reads in e appearing once in the genome is at least t times greater than the probability of the reads in e appearing twice in the genome, then we conclude that the reads in e are present only once in the genome. We carefully choose the value of t to be 20. Setting a large value for t increases the confidence that the reads in edge e appear only once in the genome. However, strings spelled by many edges will not be detected as single copy in the genome, even though they are, if the value of t is large.

To get a better estimation of L, we compute the set $\mathscr{E}$ of all the edges representing sequences that with high probability appear only once in the genome by using the above mentioned analysis and use this set in Equation 4.1. The above process is repeated until there is no change in the set of edges $\mathscr{E}$. Algorithm 11 shows the pseudocode for genome size estimation. In all our experiments at most 5 iterations of the above process were required to compute a stable set $\mathscr{E}$.

4.7 Estimating the Distribution of Insert Sizes

In some cases, input datasets may not contain information about the distribution of the insert sizes or the distance between the two reads in a mate pair. This information is needed to find paths between the reads in a mate pair in the overlap graph. If we know the mean μ and the standard deviation σ of the distribution, then we can use a depth-limited search in the overlap graph to find paths between the reads in a mate pair. The

Algorithm 11 genomeSizeEstimation($G = (V, E), n$): Genome size estimation.

Input: Overlap graph $G = (V, E)$ and number of reads n
Output: Estimated size of the genome
$\mathscr{E} \leftarrow \emptyset$ ▷ Set of edges to consider
for each edge $e \in E$ spelling a string longer than 1000 bp **do**
    Add e to $\mathscr{E}$
$d \leftarrow \sum_{e \in \mathscr{E}}$ length of the string spelled by e
$k \leftarrow \sum_{e \in \mathscr{E}}$ number of reads sampled from part of genome represented by e
$genomeSize \leftarrow \frac{nd}{k}$ ▷ First estimation of $genomeSize$ by Equation 4.1
$stepCount \leftarrow 0$
while $stepCount \leq 10$ **do** ▷ To avoid an infinite loop
    $\mathscr{E} \leftarrow \emptyset$ ▷ Set of edges considered
    for each edge $e \in E$ spelling a string longer than 1000 bp **do**
        if with high probability the string spelled by e appears once **then**
            Add e to $\mathscr{E}$
    $d \leftarrow \sum_{e \in \mathscr{E}}$ length of string spelled by e
    $k \leftarrow \sum_{e \in \mathscr{E}}$ number of reads sampled from part of genome represented by e
    $genomeSize \leftarrow \frac{nd}{k}$
    if $genomeSize$ is the same as in the previous step **then**
        break
    $stepCount \leftarrow stepCount + 1$
return $genomeSize$

minimum and maximum lengths considered by the depth-limited search are $\mu - 3\sigma$ and $\mu + 3\sigma$, respectively, as for most of the mate pairs the insert size falls within the above range.

When μ and σ are unknown for a given dataset, we estimate them as follows. After building the overlap graph, we find those mate pairs for which both of its reads map to the same composite edge of the overlap graph. For each such mate pair, we compute the distance between the reads of the mate pair on the edge that contains them. Usually there are many mate pairs in the dataset whose reads map to the same composite edge in the overlap graph. We compute their mean μ and standard deviation σ of the distances between the reads of mate pairs on the same edge.

This is our first approximation for the values of μ and σ, which is usually very close to the actual values because there usually are many long composite edges in the overlap graph. We note that, due to errors, some distances between the reads in a mate pair could be very large. These errors are introduced when mate pairs are sequenced from very large fragments than the expected insert size, or when they are sampled from multiple locations in the genome having structural variants. These erroneous distances are not considered in the subsequent steps, for which we only consider those distances $d \leq 3\mu$, where μ is the estimation of mean from the previous step. From experimental study, we found that all the correct mate pairs are within the distance 3μ in the long edges. We repeat this process until the value of μ converges. Algorithm 12 shows details about the estimation of μ and σ. In all our experiments at most 5 iterations of the above process were required for μ to converge.

4.8 Copy Count Estimation

To accurately assemble a genome we need to estimate the number of times that each read r appears in the genome. The number of times a read is present in a genome is called

Algorithm 12 meanSdEstimation($G = (V, E)$,R): Estimation of mean μ and standard deviation σ of the insert size.

Input: Overlap graph $G = (V, E)$ and set $R = \{r_1, r_2, ..., r_n\}$ of reads
Output: Estimate value of μ and σ
$S \leftarrow \emptyset$ ▷ Set of distances
for each edge $e = (u, v) \in E$ **do**
    for each mate pair $(r_1, r_2) \in e$ **do** ▷ Consider all mate pairs on same edge
        $d \leftarrow$ distance between r_1 and r_2 on e ▷ Distance between r_1, r_2 on e
        $S \leftarrow S \cup \{d\}$ ▷ Add d to S
$\mu \leftarrow$ mean of S ▷ First estimation of μ
$\sigma \leftarrow$ standard deviation of S ▷ First estimation of σ
$stepCount \leftarrow 0$
while $stepCount \leq 10$ **do** ▷ To avoid an infinite loop
    $S \leftarrow \emptyset$
    for each edge $e = (u, v) \in E$ **do**
        for each mate pair $(r_1, r_2) \in e$ **do**
            $d \leftarrow$ distance between r_1 and r_2 on e
            if $d \leq 3\mu$ **then** ▷ Only reasonable distances
                $S \leftarrow S \cup \{d\}$
    if $\mu \approx$ mean of S **then** ▷ Less than 1% different from previous value of μ
        $\mu \leftarrow$ mean of S
        $\sigma \leftarrow$ standard deviation of S
        break ▷ When μ converges
    else
        $\mu \leftarrow$ mean for S ▷ Update μ
        $\sigma \leftarrow$ standard deviation of S ▷ Update σ
    $stepCount \leftarrow stepCount + 1$
return μ and σ

its copy count. To accurately estimate the copy count of each read in the dataset, we use the combination of a statistical analysis and a minimum cost flow with a convex cost function c_e for each edge e in the overlap graph which corresponds to the likelihood of the sequence represented by e appearing k times in the genome, for every value $k \geq 1$. We also set lower bounds for the flow on each edge of the overlap graph based on a log odds ratio analysis as explained in Section 4.8.3.

4.8.1 Minimum Cost Flow

To estimate the copy counts of the reads in the dataset, PEGASUS converts the overlap graph into a flow network and solves the minimum cost flow problem on this flow network. The amount of flow through an edge represents the number of times that the reads on the corresponding edge appear in the reference genome. A generalized minimum cost flow problem on a flow network is defined as follows.

Consider flow network $G = (V, E)$ with source $s \in V$ and $t \in V$, in which every edge $(u, v) \in E$ has an upper bound of flow $flowUpperBound(u, v) > 0$, a flow $f(u, v) \geq 0$, and a cost $c(u, v) \geq 0$, the cost of sending flow $f(u, v)$ through edge (u, v) is $f(u, v)c(u, v)$. Given a flow network $G = (V, E)$ and a value d, the minimum cost flow problem is to minimize the total cost of the flow $\sum_{(u,v)\in E} f(u, v)c(u, v)$ subject to the following constraints.

Capacity constraint

$f(u, v) \leq flowUpperBound(u, v)$

Skew symmetry

$f(u, v) = -f(v, u)$

Flow conservation

$\sum_{w\in V} f(u, w) = 0$ for all $u \neq s, t$ and

Required flow

$\sum_{w\in V} f(s, w) = d$ and $\sum_{w\in V} f(w, t) = d$

The flow network used by PEGASUS has some additional constraints. For example, each of the edges in the flow network also has a flow lower bound.

4.8.2 Cost Function

The maximum likelihood genome assembler proposed by Medvedev et al. [28, 29, 30, 31] uses a bidirected network flow to model the double-stranded nature of DNA for genome assembly. Most genome assemblers try to minimize the length of the assembled genome. The problem with minimizing the length of the assembled genome $\mathscr{G}'$ is that the actual genome $\mathscr{G}$ may contain repeated subsequences, which will not appear multiple times in the assembled genome $\mathscr{G}'$.

The idea of maximum likelihood genome assembly is not to minimize the assembled genome size, but rather to assemble a genome that is the most likely source for the given set R of reads. If a subsequence is present multiple times in the genome, the reads from that subsequence are likely to be sampled more often than the reads from unique subsequences of the genome. Maximum likelihood genome assembly is designed to deal with high coverage datasets produced by NGS. This method uses a bidirected network flow-based algorithm to estimate the copy counts of the reads; these copy counts are then used to assemble the genome.

Experiments conducted on simulated dataset (i.e., randomly sampled reads from a reference genome by a computer program) from Escherichia coli show that the maximum likelihood genome assembly can very accurately (more than 99%) compute the copy counts of reads [29]. Maximum likelihood genome assembly uses a bidirected de Bruijn graph to model the input set of reads. This bidirected graph is converted into a directed graph by using an algorithm of Hochbaum [15]. A convex min-cost function $c_e : \mathbb{N} \to \mathbb{R}$, is associated with every edge e reflecting the likelihood that the sequence represented by e appears k times in the genome for each $k \geq 1$. The goal is to compute a flow function f that minimizes $\sum_e c_e(f(e))$, where the flow through edge e is $f(e)$.

We use flow network to compute copy counts of reads in the reference genome by setting cost in the edges that resembles the number of time the reads in the edges are present in the genome, given the number of times they are present in the dataset. Consider a set R of n reads from a genome $\mathscr{G}$ of length L. Let read $r \in R$ be present x_r times in the dataset R and d_r times in the genome $\mathscr{G}$. Then, if n reads are sampled uniformly from $\mathscr{G}$, the probability that r is sampled x_r times is

$$Prob\left(Freq(r) = x_r\right) = \binom{n}{x_r} \left(\frac{d_r}{L}\right)^{x_r} \left(1 - \frac{d_r}{L}\right)^{n-x_r} \tag{4.2}$$

Here, $Freq(r)$ denotes the number of times read r appears in the dataset R. Equation 4.2 is a complex function on x_r that represents the probability that read r appears x_r times in the dataset. We use this function to model the cost of pushing flow through the read r by taking the negative log of Equation 4.2.

$$\begin{aligned}
-\log(Prob\left(Freq(r) = x_r\right)) &= -\log\binom{n}{x_r} - \log\left(\frac{d_r}{L}\right)^{x_r} - \log\left(1 - \frac{d_r}{L}\right)^{n-x_r} \\
&= -\log\binom{n}{x_r} - x_r \log d_r + x_r \log L \\
&\quad - (n - x_r)\log(L - d_r) + (n - x_r)\log L \\
&= -\log\binom{n}{x_r} + n\log L - x_r \log d_r \\
&\quad - (n - x_r)\log(L - d_r) \\
&= K + c_r(d_r)
\end{aligned}$$

where,

$$K = -\log\binom{n}{x_r} + n\log L$$

and

$$c_r(d_r) = -\, x_r \log d_r - (n - x_r) \log(L - d_r) \tag{4.3}$$

Note that K does not depend on the number of times that read r appears in the genome, so for our purposes we assume that K is constant. We use $c_r(d_r)$ from Equation 4.3 as a convex cost function in our program to calculate the cost of pushing d_r units of flow through read r. In Equation 4.3 the cost function maximizes the probability that the read r is present x_r times in the dataset. To efficiently calculate a minimum cost flow we first approximate the convex cost function $c_r(d_r)$ with a piecewise linear cost function with 3 linear segments as shown in Figure 4.11. The first segment gives the cost for flow values between 1 and 3, the second linear segment gives the cost for flow values between 3 and 5 and the last one is for flow larger than 5. We choose these intervals by observing that most of the copy counts of the reads are within the range between 1 and 5.

4.8.3 A-Statistics

To set the lower bounds of flow on the edges of the overlap graph, we approximate the minimum number of times each edge should be present in the genome based on the frequencies of the reads present in the edge. This approximation in done by a statistical analysis called A-statistics [35].

The *odds ratio* z of two events, $event_1$ and $event_2$, is the ratio of the probability $event_1$ divided by the probability of $event_2$.

$$z = \frac{Prob(event_1)}{Prob(event_2)}.$$

If $z = 1$, this means that the two events are equally likely to occur, whereas $z > 1$ means that $event_1$ is z times more likely to occur than $event_2$. The *log odds ratio*, or

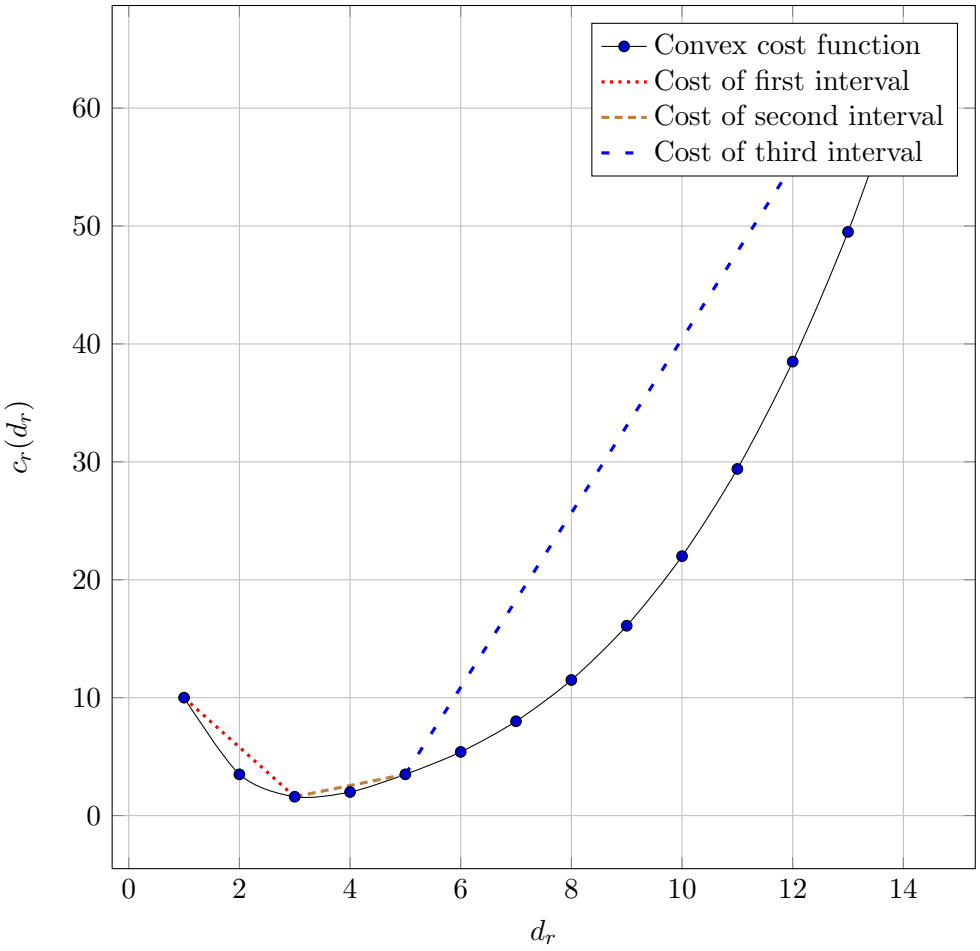


Figure 4.11: Convex cost function $c_r(d_r)$.

A-Statistics, is the natural logarithm of this odds ratio. In the overlap graph, we compute the log odds ratio of the string spelled by an edge appearing i times versus the string appearing $i+1$ times in the genome. Let the size of a given genome be L, the number of reads in the dataset for this genome be n. Consider the overlap graph for the dataset and an edge x of the overlap graph, let the length of the string spelled by edge x be d and let k be the number of reads from the dataset, including repeated reads, represented by x. Then by Equation 4.2 the probability that the string spelled by edge x appears only once in the genome is

$$Prob(copyCount(x) = 1) = \binom{n}{k}\left(\frac{d}{L}\right)^k\left(\frac{L-d}{L}\right)^{n-k}$$
$$\approx \frac{\left(\frac{dn}{L}\right)^k}{k!}e^{-\frac{dn}{L}} \quad \text{when } L \text{ is large [35]}$$

Similarly, the probability of the string spelled by edge x appearing twice in the genome is

$$Prob(copyCount(x) = 2) \approx \frac{\left(\frac{2dn}{L}\right)^k}{k!}e^{-\frac{2dn}{L}}$$

The log odds ratio of the probabilities of $copyCount(x) = 1$ and $copyCount(x) = 2$ is

$$aStat_{1,2}(x) = \ln\frac{Prob(copyCount(x) = 1)}{Prob(copyCount(x) = 2)}$$
$$= \frac{dn}{L} - k\ln 2$$

Similarly, the log odds ratio of the probabilities of $copyCount(x) = i$ and $copyCount(x) = i + 1$ is

$$aStat_{i,i+1}(x) = \ln\frac{Prob(copyCount(x) = i)}{Prob(copyCount(x) = i + 1)}$$
$$= \frac{dn}{L} - k\ln\frac{i+1}{i} \tag{4.4}$$

We use Equation 4.4 to help us estimate the number of times that string represented by every edge appears in the genome.

4.8.4 Accurate Copy Count

Based on the log odds ratio discussed in the previous section, we set lower bounds and upper bounds for the flow on each long composite edge x (longer than 1000 bp) in the

overlap graph. We choose a threshold t and if $aStat_{1,2}(x) \geq t$, then the probability that $copyCount(x) = 1$ is e^t times the probability that $copyCount(x) = 2$. We performed several experiments and found that by setting $t = 3$, whenever $aStat_{1,2}(x) \geq t$ then in most cases the string spelled by x appeared only once in the genome. Hence, if $aStat_{1,2}(x) \geq t$ we set a lower bound and upper bound on the capacity of the edge representing x to 1, thus forcing the flow on that edge to have value 1. However, if $aStat_{1,2}(x) < t$ we find the first value i such that $aStat_{i-1,i}(x) \leq -t$ and $aStat_{i,i+1}(x) \geq t$ and set the lower bound of flow on edge x to i and the upper bound of flow to 1000. We choose the upper bound of flow to be 1000 because it is highly unlikely that any read appears in 1000 different positions in the genome. Note that if $aStat_{i-1,i}(x) \leq -t$ and $aStat_{i,i+1}(x) \geq t$, this means that the probability that the string spelled by edge x appears i times in the genome is at least e^t times higher that the probability that this string appears $i - 1$ or $i + 1$ times. We only consider composite edges longer than 1000 base pairs for this log odds ratio calculation because we can only rely on this statistical analysis for edges representing large set of reads. For composite edges smaller than 1000 base pairs, or if the log odds ratio is unable to assign bounds for the flow as explained above, we set the lower bound for flow to 0 (if the edge represents a set of fewer than 30 reads) or to 1 (if the edge represents a set of more than 30 reads); for these edges we set an upper bound 1000 for the maximum flow that they can carry. By experimental study we found that setting lower bound of flow to 1 in the edges with fewer than 30 reads causes problem in the copy count estimation.

A similar statistical analysis was used by Gene Myers [35]. However, he only used the A-statistics values to find edges that are likely to appear exactly once in the genome. He did not use the A-statistics values to predict copy counts larger than 1. We obtained very accurate copy count results by using the above mentioned approach.

There are several algorithms for solving the minimum cost flow problem for directed graphs. In order to use the existing algorithms to compute a minimum cost flow in the

overlap graph, we have to first convert the overlap graph, which is a bidirected graph, into a directed graph [12]. To convert a bidirected graph $G = (V, E)$ into a directed graph $G' = (V', E')$, for each node $v \in V$, we add two nodes v_1 and v_2 to V'; V' also includes a "super" source s' and a "super" sink t'. For each bidirected edge $e = (u, v) \in V$, we add two directed edges in G' depending on the orientation of e as show in Figure 4.13 [28]. Then we duplicate each node $u \in G'$, by adding two nodes u_{in} and u_{out} to G' and connecting all incoming edges of u to u_{in} and all outgoing edges to u_{out}. We also add an edge (u_{in}, u_{out}), an edge (s', u_{in}) from the super source s' and an edge (u_{out}, t') to the super sink t'. We add an edge (t', s') connecting the super sink and super source. For completeness sake we present the pseudocode of converting G to G' in Algorithm 13. We use CS2 [13] to compute a minimum cost flow for G'.

Algorithm 14 shows how we computed the flow in the overlap graph. If the A-statistics suggest that the string spelled by an edge is present $a > 1$ times in the genome, then we make three copies e_1, e_2 and e_3 of the edge in the directed overlap graph to represent the three line segments of our convex cost function. Flow bounds and costs of the edges e_1, e_2 and e_3 are shown below.

$$flowLowerBound(e_1) \leftarrow max\{a, 3\} \tag{4.5a}$$

$$flowUpperBound(e_1) \leftarrow 3 \tag{4.5b}$$

$$cost(e_1) \leftarrow \frac{1}{3} \sum_{r \in e} \{c_r(3) - c_r(1)\} \tag{4.5c}$$

Similarly

$$flowLowerBound(e_2) \leftarrow max\{0, min\{a, 5\} - 3\} \tag{4.6a}$$

$$flowUpperBound(e_2) \leftarrow 2 \tag{4.6b}$$

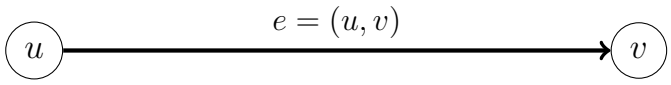
$$cost(e_2) \leftarrow \frac{1}{2} \sum_{r \in e} \{c_r(5) - c_r(3)\} \tag{4.6c}$$

And, finally

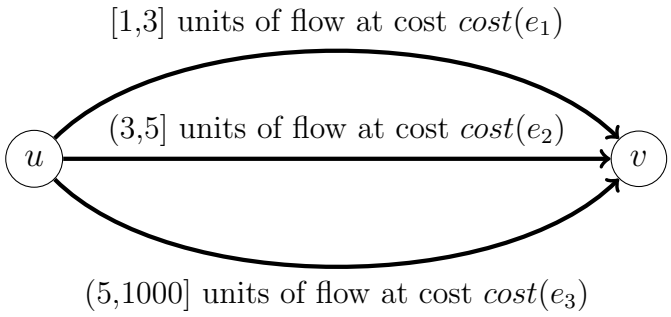
$$flowLowerBound(e_3) \leftarrow max\{0, min\{a, 1000\} - 5\} \tag{4.7a}$$

$$flowUpperBound(e_3) \leftarrow 995 \tag{4.7b}$$

$$cost(e_3) \leftarrow \frac{1}{995} \sum_{r \in e} \{c_r(1000) - c_r(5)\} \tag{4.7c}$$



(a) Composite edge in the directed overlap graph.



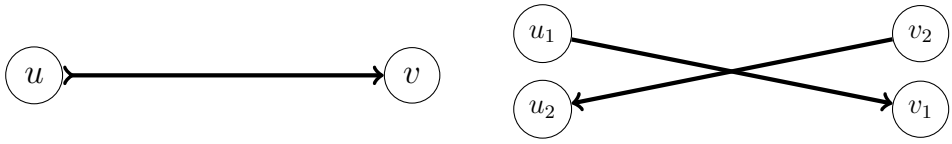
(b) Composite edges for three step convex cost function.

Figure 4.12: Putting costs in the composite edges using Equations 4.5c, 4.6c and 4.7c.

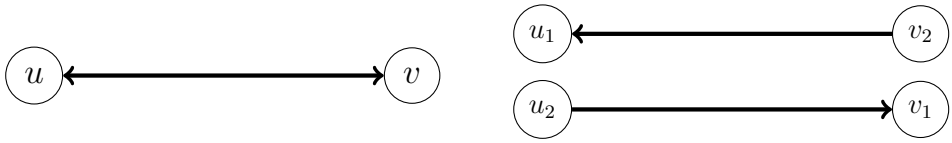
The convex cost function and the three liner segments of the piecewise linear approximation are shown in Figure 4.11. The three liner segments of the cost functions are modeled as shown in Figure 4.12.

Algorithm 13 convertGraph($G = (V, E)$): Convert a bidirected graph $G = (V, E)$ into a directed graph $G' = (V', E')$.

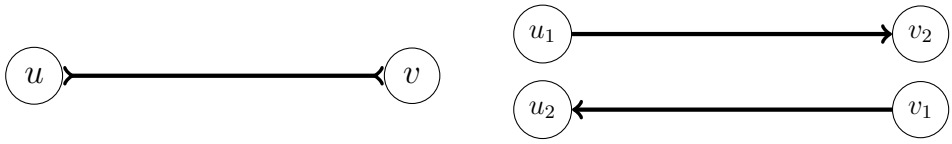
Input: Bidirected overlap graph $G = (V, E)$
Output: Directed overlap graph $G' = (V', E')$
$V' \leftarrow \{s', t'\}$ ▷ Add super-source s' and super-sink t'
$E' \leftarrow \{(t', s')\}$ ▷ Add edge from super-sink t to super-source s
for each read $v \in V$ **do**
    $V' \leftarrow V' \cup \{v_{1_{in}}, v_{1_{out}}, v_{2_{in}}, v_{2_{out}}\}$
    $E' \leftarrow E' \cup \{(v_{1_{in}}, v_{1_{out}}), (v_{2_{in}}, v_{2_{out}})\}$ ▷ Edge connections representing nodes in G
    $E' \leftarrow E' \cup \{(s', v_{1_{in}}), (s', v_{2_{in}})\}$ ▷ Add edges from super-source s'
    $E' \leftarrow E' \cup \{(v_{1_{out}}, t'), (v_{2_{out}}, t')\}$ ▷ Add edges to super-sink t'
for each edge $e = (u, v) \in E$ **do** ▷ Edge connections representing edges in G
    if e is a forward-forward edge **then** ▷ Figure 4.13a and 4.13b
        $E' \leftarrow E' \cup \{(u_{1_{out}}, v_{1_{in}}), (v_{2_{out}}, u_{2_{in}})\}$
    else if e is a reverse-forward edge **then** ▷ Figure 4.13c and 4.13d
        $E' \leftarrow E' \cup \{(u_{2_{out}}, v_{1_{in}}), (v_{2_{out}}, u_{1_{in}})\}$
    else if e is a forward-reverse edge **then** ▷ Figure 4.13e and 4.13f
        $E' \leftarrow E' \cup \{(u_{1_{out}}, v_{2_{in}}), (v_{1_{out}}, u_{2_{in}})\}$
return G'



(a) Bidirected forward-forward edge. (b) Directed edges for forward-forward edge.



(c) Bidirected reverse-forward edge. (d) Directed edges for reverse-forward edge.



(e) Bidirected forward-reverse edge. (f) Directed edges for forward-forward edge.

Figure 4.13: Bidirected edge to directed edges conversion.

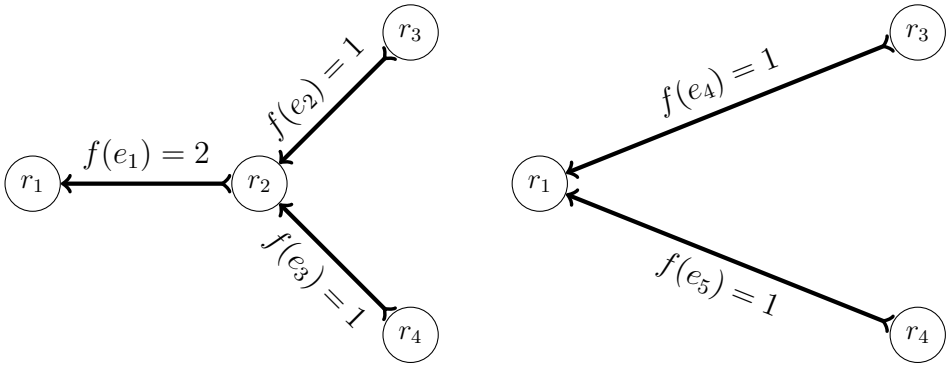
4.9 In-tree and Out-tree Reductions

A node in the overlap graph that has only one outgoing edge and more than one incoming edges is called an *in-node* (an example is node r_2 in Figure 4.14a) and the subgraph formed

Algorithm 14 computeMinCostFlow($G = (V, E)$): Minimum cost flow computation of the overlap graph $G = (V, E)$.

Input: Overlap graph $G = (V, E)$
Output: Flow on each edge $e \in V$
call *convertGraph*(G)
Set cost of flow in (t', s') to ∞
Set lower bound of flow in (t', s') to 1
Set upper bound of flow in (t', s') to ∞
for each read $v \in V'$ **do** ▷ Costs and bounds of edges in G' representing nodes in G
    Set cost of flow in $(v_{1_{in}}, v_{1_{out}})$ to 1
    Set lower bound of flow in $f(v_{1_{in}}, v_{1_{out}})$ to 0
    Set upper bound of flow in $(v_{1_{in}}, v_{1_{out}})$ to 1000
    Set cost of flow in $(v_{2_{in}}, v_{2_{out}})$ to 1
    Set lower bound of flow in $(v_{2_{in}}, v_{2_{out}})$ to 0
    Set upper bound of flow in $(v_{2_{in}}, v_{2_{out}})$ to 1000
for each edge $e = (u, v) \in E'$ **do**
    if e is a composite edge **then**
        if length of the string spelled by $e \geq 1000$ **then**
            if $aStat_{1,2}(e) > 3$ **then** ▷ Edges used exactly once
                Set cost of flow in e to 0
                Set lower bound of flow in e to 1
                Set upper bound of flow in e to 1
            else
                $E' \leftarrow E' - \{e\}$
                $E' \leftarrow E' \cup \{e_1 = (u, v), e_2 = (u, v), e_3 = (u, v)\}$
                Set cost and bounds according to Equations 4.5, 4.6 and 4.7
        else
            $E' \leftarrow E' - \{e\}$
            $E' \leftarrow E' \cup \{e_1 = (u, v), e_2 = (u, v), e_3 = (u, v)\}$
            if e has at least 30 reads **then**
                Set lower bound of flow in e_1 to 1
            else
                Set lower bound of flow in e_1 to 0
            Set cost and bounds according to Equations 4.5, 4.6 and 4.7
    else ▷ Simple edges or when A-statistics is unable to find *copyCount*
        Set cost of flow in e to 0
        Set lower bound of flow in e to 0
        Set upper bound of flow in e to 1000
call CS2(G') ▷ Minimum cost flow implementation of CS2 [13]
Convert the flows from G' to G ▷ Sum of average flow of duplicated edges

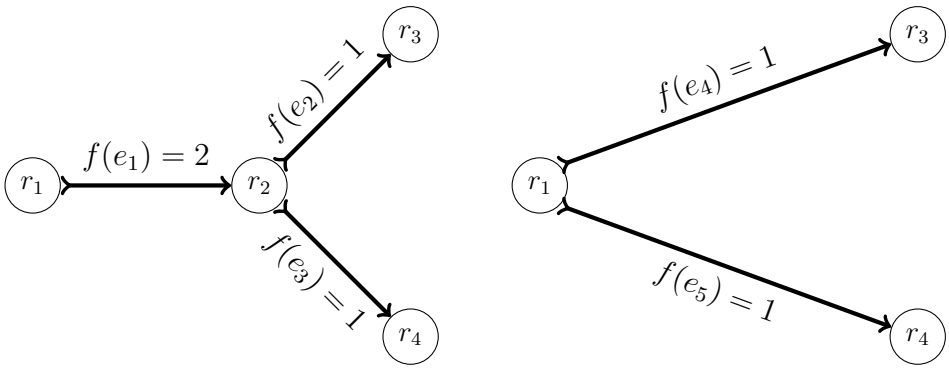
by an in-node and all its neighbours is called an *in-tree*. Similarly, a node with only one incoming edge and more than one outgoing edges is called an *out-node* (an example is node r_2 in Figure 4.15a) and the subgraph formed by an out-node and all its neighbours is called an *out-tree*.



(a) In-tree before simplification. (b) In-tree after simplification.

Figure 4.14: In-tree simplification.

In the in-tree in Figure 4.14, the flow values in the edges are $f(e_1 = (r_1, r_2)) = 2$, $f(e_2 = (r_2, r_3)) = 1$ and $f(e_3 = (r_2, r_4)) = 1$. Node r_2 has only one outgoing edge and two incoming edges. Edges e_1 and e_2 in Figure 4.14a can be merged into a single edge and also edges e_1 and e_3 can be merged into a single edge. Hence, the in-tree in Figure 4.14a can be simplified as shown in Figure 4.14b. Similarly, an out-tree can be simplified as shown in Figure 4.15. In-tree and out-tree simplifications are described in Algorithm 15.



(a) Out-tree before simplification. (b) Out-tree after simplification.

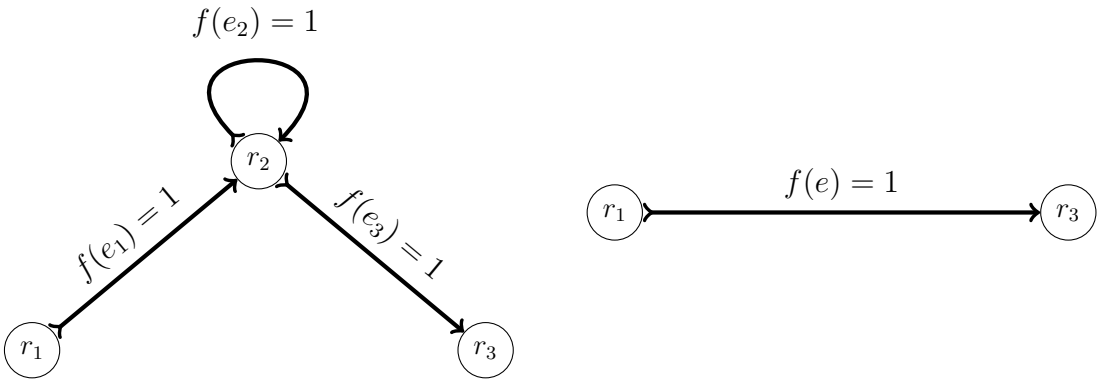
Figure 4.15: Out-tree simplification.

Algorithm 15 reduceTrees($G = (V, E)$): In-tree and out-tree reduction.

```
1: Input: Overlap graph G = (V, E)
2: Output: Tree reduced overlap graph
3: for each node u ∈ V do
4:     inDegree ← number of incoming edges incident on u
5:     outDegree ← number of outgoing edges incident of u
6:     if inDegree = 1 and outDegree > 1 then                    ▷ Out-tree
7:         for each out-edge e_1 = (u, u_{i_out}) do
8:             mergeEdges(G, e = (u_in, u), e_1)
9:     else if inDegree > 1 and outDegree = 1 then               ▷ In-tree
10:        for each in-edge e_2 = (u_{i_in}, u) do
11:            mergeEdges(G, e_2, e = (u, u_out))
12: return G
```

4.10 Loop Reductions

An edge in the overlap graph of the form (u, u) is called a *loop*. Consider the loop of the overlap graph in Figure 4.16a. There is only one valid way in which the flow can travel through these edges: through the path $p = \{r_1, e_1, r_2, e_2, r_2, e_3, r_3\}$. Hence, we can remove the path p and replace it by an edge $e = (r_1, r_3)$ as shown in Figure 4.16b. We copy the information that was stored in p into the edge e. Algorithm 16 shows the procedure for loop path reduction.



(a) Before reducing loop. (b) After reducing loop.

Figure 4.16: Reducing loop in the overlap graph.

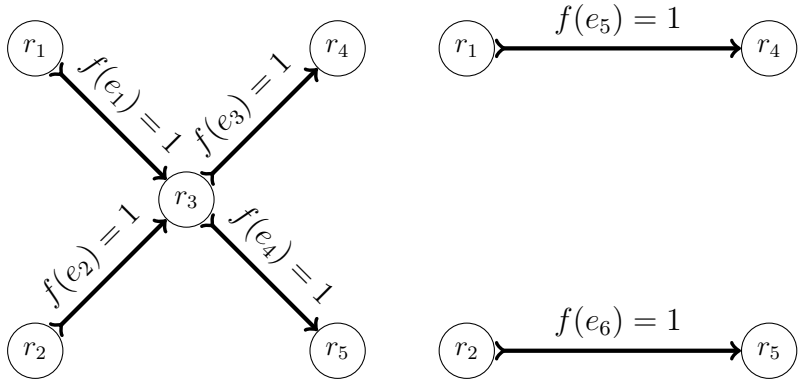
Algorithm 16 reduceLoops($G = (V, E)$): Reduce loops in the overlap graph.

Input: Overlap graph $G = (V, E)$
Output: Loop-reduced overlap graph
for each loop (u, u) in G **do**
    if u has only two edges (x, u) and (u, y) incident on it **then**
        if there is only one possible path $p = \{x, u, u, y\}$ for the flow **then**
            Remove p from G
            Add (x, y) to G and copy information stored in p into (x, y)
return G

4.11 Resolving Nodes by Mate Pairs

In the overlap graph, there might be many "ambiguous nodes" where the edges could be merged in more than one way to reconstruct a longer part of the genome. Mate pairs might be very helpful to resolve ambiguity in these nodes. Consider for example, node r_3 in Figure 4.17a, which has two units of incoming flow and two units of outgoing flow. Only considering the flow, we are unable to reconstruct the parts of the genome around node r_3, since the incoming flow through edge $e_1 = (r_1, r_3)$ can go through any of the two outgoing edges $e_3 = (r_3, r_4)$ or $e_4 = (r_3, r_5)$. Similarly the incoming flow through edge $e_2 = (r_2, r_3)$ can go through any of the outgoing edges e_3 or e_4, and hence the node r_3 is an *ambiguous node*. If there are enough mate pairs supporting a pair of edges e_1 and e_3, then we merge the edges e_1 and e_3. Similarly we merge the edges e_2 and e_4, if they are supported by enough mate pairs. Figure 4.17b shows the overlap graph after node r_3 is resolved by using mate pair support, assuming that e_1 and e_3 are supported by mate pairs or e_2 and e_4 are supported by mate pairs.

After building and simplifying the overlap graph as explained in the previous sections, we calculate the support of all pairs of the adjacent edges in the graph. While building the overlap graph we keep track of the locations of all the reads in the edges. For each mate pair, we search for all the paths in the overlap graph between the reads of the mate pairs having length between $\mu - 3\sigma$ and $\mu + 3\sigma$, where μ is the mean of the insert size and σ is the standard deviation of the insert size that we estimated before (Section 4.7).



(a) Ambiguous node r_3. (b) Node r_3 is resolved.

Figure 4.17: Ambiguous node resolved in the overlap graph by using mate pairs.

If for the reads of a mate pair, all these paths use an edge $e_1 = (u, v)$ followed by an edge $e_2 = (v, w)$, we say that the mate pair supports these adjacent edges (e_1, e_2) and increase the support of the edge pair (e_1, e_2). After finding the supports of all the pairs of adjacent edges in the overlap graph, we merge the adjacent edges that have enough support (at least 5). To avoid merging pair of edges supported by erroneous mate pairs or erroneous paths in the graph, we choose the support threshold to be 5. Resolving ambiguous nodes by mate pair supports is described in Algorithm 17.

To find support between all pairs of adjacent edges in the overlap graph, we need to find all paths between the reads of every mate pairs. To reduce the amount of memory used by PEGASUS, we find paths in the graph starting from a specific node. For a node u in the overlap graph, we find all paths of length $\mu + 3\sigma$ from u and store them in a list. Then for each read r in the edges incident on u, we check if the mate pair m of r is on any of the paths in the list. If such a path exists, we mark the path. Then we check if there are more paths between the reads in the list of paths. We only mark the pair of edges that are adjacent on all such paths found. When we finish searching the list, we have a list of supported pair of edge. For each supported pair of edges (e_1, e_2), we store the pair of edges in a hash table. If the pair of edges is stored in the hash table for the first time then we set the support of the pair of edges to 1, otherwise we increase the

Algorithm 17 resolveNodes($G = (V, E), R, \mu, \sigma$): Resolve nodes by mate pairs.

Input: Overlap graph $G = (V, E)$, set R of reads, mean μ and standard deviation of σ insert size
Output: Node resolved overlap graph
for each node $v \in V$ **do**
    Find set P of paths from v of length at most $\mu + 3\sigma$
    for each mate pair $(r_1, r_2) \in R$ **and** r_1 in on edge incident on u **do**
        for each path $p \in P$ from r_1 to r_2 of length $\geq \mu - 3\sigma$ **and** $\leq \mu + 3\sigma$ **do**
            if p is the first path **then**
                $p' \leftarrow p$
                Mark all pair of adjacent edges in p' as *supported*
            else
                for each adjacent pair of edges $(e_1, e_2) \in p'$ **do**
                    if (e_1, e_2) is not adjacent in p **then**
                        Mark (e_1, e_2) in p' as *unsupported*
            if no adjacent pair of edges in p' are *supported* **then**
                break
        for each adjacent supported pair of edges $(e_1, e_2) \in p'$ **do**
            Store (e_1, e_2) in a hash table ▷ Hash table stores the support count
Sort the supported pairs of edges in the hash table
for each pair of supported edges (e_1, e_2) in the hash table **do**
    if support count of the pair $(e_1, e_2) \geq 5$ **then** ▷ Default support threshold is 5
        $mergeEdges(G, e_1, e_2)$
return G

support of the pair of edges in the hash table. We do these steps for all the reads that are on the edges incident to u. When we are done processing the nodes u, we free the memory used to store the paths. Then we go to another node in the overlap graph and do the same steps until all the nodes in the graph are processed.

Storing only the paths starting from the current node reduces the amount of memory used by the program. Moreover, we sort the list of paths from the current node to efficiently do binary search to find desired paths from the list. This reduces the running time to search paths in the overlap graph.

4.12 Contig Extraction

In the previous section, we explained how to compute the support between pairs of adjacent edges by finding paths between the reads of a mate pair. However due to lack of coverage, there may not be a path in the graph between the two reads of some mate pairs. Even in these cases we might still use mate pairs to further simplify the overlap graph as follows. For each pair of non adjacent edges e_1 and e_2, if there is a mate pair (r_1, r_2) such that r_1 is in e_1 and r_2 is in e_2, and $distanceOnEdge(\overleftarrow{e_1}, r_1) + distanceOnEdge(e_2, r_2) \leq \mu + 3\sigma$, then we increase the support for the edge pair (e_1, e_2). Here, $\overleftarrow{e_1} = (v, u)$ is the reverse of the edge $e_1 = (u, v)$ and $distanceOnEdge(e = (u, v), r)$ is the position of the read r in the string spelled by the edge $e = (u, v)$. After finding the support for all such pairs of edges, we combine each pair of edges according to their support: First we calculate the approximate distance between the strings spelled by the edges in a pair. We know that the average distance between the reads in a mate pair is μ. So if the average distance between the reads in the mate pairs on the pair of edges (e_1, e_2) is d, then the size of the gap is approximately $\mu - d$.

If there are n contigs in the overlap graph, then we need to check $O(n^2)$ pair of contigs to find supports. To reduce the number of pairs of contigs to check, we only consider

pair of contigs that share mate pairs. For each contigs, we first find the contigs which share mate pair with the current contigs. Then we search for the distances on the contigs for reads in mate pairs. Any spaces between the strings spelled by two edges are filled with `N`'s. Algorithm 18 shows details of the contig extraction procedure. Note that, to be considered as a contig the edge e must have flow > 0 or the length of the string spelled by e should be at least 100 base pairs. These constraints are used to avoid false supports among contigs.

Algorithm 18 mergeContigs$(G = (V, E), R, \mu, \sigma)$**:** Merge contigs using mate pairs.

Input: Overlap graph $G = (V, E)$, set R of reads, mean μ and standard deviation σ of insert size
Output: Overlap graph after merging contigs
for each pair of composite edge $e_1 = (u, v), e_2 = (w, x) \in E$ either having nonzero flow or length at least 100 bp **do**
    for each pair $(r_1, r_2) \in R$ of reads in a mate pair **do**
        if $r_1 \in e_1$ **and** $r_2 \in e_2$ **then**
            if $distanceOnEdge(\overleftarrow{e_1}, r_1) + distanceOnEdge(e_2, r_2) \leq \mu + 3\sigma$ **then**
                Increase support of the pair of edges (e_1, e_2)
Sort the supported pairs of edges according to their support
for each pair of supported edges (e_1, e_2) **do**
    if support count of the pair $(e_1, e_2) \geq 5$ **then** ▷ Default support threshold is 5
        $d \leftarrow$ average distance between the mate pairs in (e_1, e_2)
        $g \leftarrow \mu - d$
        Merge the edges (e_1, e_2) with a gap g filled with N's
return G

4.13 The Algorithm

In this section we give high a level description of the whole algorithm used by PEGASUS to assemble a genome. First we correct the reads using RACER, sort the unique corrected reads and store the frequency of each one of them. The unique reads are stored in an array called $readsForward$. For each unique read $r \in R$, we only store either r or $\bar{r}$ in $readsForward$, whichever is lexicographically smaller. The index of each read in the sorted list $readsForward$ is assigned as the read's ID. We compute the reverse

complement of all reads in the array $readsForward$ and store them in another array, $readsReverse$. We also store the reads of each mate pair in a linked list.

PEGASUS stores in the hash table the prefixes and suffixes of each read $r \in R$ and its reverse complement $\bar{r}$ of length $h = min\{minOverlap, 64\}$ bases to minimize the number of pairs of reads to compare for overlaps as explained in Section 4.3.1. For each pair of reads u and v that overlaps by at least $minOverlap$ base pairs, we insert an edge $e = (u, v)$ in the overlap graph. While inserting the edge e in the overlap graph, we always check for any transitive edges in the overlap graph introduced by the newly inserted edge e. We remove all transitive edges while inserting edges in the overlap graph.

After building the overlap graph, we contract composite paths as explained in Section 4.4. Dead-ends and bubbles caused by erroneous reads are then removed from the overlap graph. We estimate the genome size and distribution of the insert size and then compute a minimum cost flow in the overlap graph using CS2 [13] to estimate the copy counts of the reads in the dataset. In-trees and out-trees are reduced and we compute the support for all pairs of adjacent edges, merging those pairs of edges that have enough support.

In the final step of PEGASUS, we compute the support between all pairs of nearby, but not adjacent edges. Because of the lack of coverage, sometimes adjacent contigs from the genome can be disconnected in the overlap graph, but they can still be merged if there is enough mate pair support between them. Finally we output the strings in all long composite edges in the overlap graph and save them in a FASTA file. This file contains the final output of our program. Algorithm 19 shows the pseudocode for PEGASUS.

Algorithm 19 PEGASUS(R,$minOvelap$): Paired-End Genome ASsembly Using Short-sequences.

Input: Set $R = \{r_1, r_2, ..., r_n\}$ of reads, minimum overlap length $minOverlap$
Output: Set $C = \{c_1, c_2, ..., c_k\}$ of contigs
$C \leftarrow \emptyset$
call $RACER(R)$ ▷ Read correction using RACER
call $buildOverlapGraph(R, minOverlap)$
repeat
    call $contractCompositePaths(G)$ ▷ Contract composite paths
    call $removeDeadEnds(G)$ ▷ Remove dead-ends
    call $removeBubbles(G)$ ▷ Remove bubbles
until no edge is removed from G
$L \leftarrow genomeSizeEstimation(G, n)$
call $meanSdEstimation(G, R)$ ▷ Estimate μ and σ
call $computeMinCostFlow(G, L)$
repeat
    call $reduceTrees(G)$ ▷ Reduce in-trees and out-trees
    call $reduceLoops(G)$ ▷ Remove loops
until no edge is removed from G
repeat
    call $resolveNodes(G, R, \mu, \sigma)$ ▷ Resolve ambiguous nodes using mate pairs
until no edge is removed from G
repeat
    call $mergeContigs(G, R, \mu, \sigma)$ ▷ Resolve disconnected contigs using mate pairs
until no edge is removed from G
for each edge $e \in E$ **do**
    if length of $e \geq 100$ bp **then**
        $C \leftarrow C \cup$ {string spelled by e}
Output C in a FASTA file

Chapter 5

Experiments

In this chapter, we show our experimental results and compare the output produced by PEGASUS with that of two of the top assemblers, ABySS [46] and SOAPdenovo [22]. The contigs produced by the assemblers do not always perfectly align to the *reference genomes* (i.e., sequence of base pairs assembled as a representative example of an organism). This happens mainly because of misassemblies by the assemblers, errors in the datasets, under-sampled parts of the genome and differences between the reference genome and the source genome for the dataset. To check the quality of the assembled contigs, we align (approximate string matching) the contigs to the corresponding reference genome. Sometimes contigs produced by assemblers are broken into smaller contigs after aligning them to the reference genome. We align the original contigs produced by the assemblers to the reference genome by using BWA-SW [21] allowing for a few base pairs mismatches, insertions and deletions. If there are many mismatches, insertions and deletions in a contig, the contig is broken into smaller contigs.

5.1 Datasets

To compare the performance of PEGASUS with that of other genome assemblers, we have used 16 datasets obtained from 14 different organisms. The lengths of the reference

genomes of the datasets vary from about 1 Mbp to 102 Mbp. The read lengths of the datasets vary from 36 bp to 101 bp. The datasets used in the experiments are from bacterial genomes, except for the dataset SRR065390 which is from a worm C.elegans. The datasets were downloaded from short read archive (SRA) database (`www.ncbi.nlm.nih.gov/sra`). Table 5.1 shows details (accession number, read length, number of reads, reference genome, organism name and size of the genome) about the datasets used in out tests. An *accession number* is the unique ID of a dataset. The SRA database stores datasets in `.sra` files, which contain compressed representations of the reads. The downloaded `.sra` files were converted to FASTA/FASTQ file formats by using the `sratoolkit` (`www.ncbi.nlm.nih.gov/books/NBK56560/`) provided in the SRA website. For each of the datasets, `sratoolkit` produces one file containing all the paired-end reads. The reads in the files are arranged in such a way that the reads in a mate pair are consecutive.

Table 5.1: Datasets downloaded from the Short Read Archive (SRA) Database.

Accession Number	**Read Length (bp)**	**Number of Reads**	**Reference Genome**	**Organism Name**	**Genome Size (Mbp)**
SRR065390	100	67,617,092	Build WS222	C.elegans	102
DRR000852	75	3,519,504	NC_000964.3	B.subtilis	4.2
SRR387785	100	5,211,082	NC_015291.1	S.oralis1	1.96
SRR446554	100	8,735,304	NC_015291.1	S.oralis2	1.96
SRR402006	101	8,169,824	NC_014256.1	H.pylori	1.6
SRR413299	100	9,497,946	NC_002950.2	P.gingivalis	2.34
SRR387784	100	4,407,248	NC_015875.1	S.pseudopneumonie	2.19
SRR387794	100	4,932,870	NC_015964.1	H.parainfluenzae	2.09
SRR397962	100	7,127,250	NC_005823.1	L.interrogans	4.2
SRR072099	36	30,355,432	NC_000913.2	E.coli	4.6
SRR387776	100	4,925,280	NC_013853.1	S.mitis	2.15
SRR400550	36	31,994,160	NC_009012.1	C.thermocellum	3.84
ERR021957	37	7,825,944	NC_000117.1	C.trachomatis1	1.04
ERR021958	37	11,504,594	NC_000117.1	C.trachomatis2	1.04
SRR063416	101	6,907,220	NC_006570.2	F.tularemia	1.89
SRR387738	100	4,997,274	NC_015678.1	S.parasanguinis	2.15

5.2 Experimental Settings

All the three assemblers were executed in the SHARCNET (www.sharcnet.ca) cluster redfin. The cluster redfin has 98 GB of memory and 552 processors running CentOS 5.4. The three assemblers were executed in serial mode. Running time and memory usage for each of the assemblers for each of the datasets were reported by redfin. Both ABySS and SOAPdenovo can be executed in parallel mode. However, we compared the running time and memory usage in serial mode as PEGASUS runs in serial mode only.

5.3 Definitions

In this section, we discuss some of the evaluation parameters used to analyze and compare the results of the different genome assemblers.

5.3.1 Gaps

A *gap* is a space introduced into an aligned sequence for a deletion relative to the reference sequence. Figure 5.1 shows a gap in sequence s_2 relative to sequence s_1. This gap is called a deletion for s_2 relative to s_1 and an insertion for s_1 relative to s_2. Assembled sequences might contain gaps relative to the reference genome. Gaps are introduced due to errors in the dataset, lack of coverage in the dataset or assembly errors.

```
s1  :  AGCTAAGCATTTACGATAGCCGATAGCTAAATTAC
       ||||||||||||||| ||||||||||||||||||
s2  :  AGCTAAGCATTTACG-TAGCCGATAGCTAAATTAC
```

Figure 5.1: A gap in sequence s_2 relative to sequence s_1.

5.3.2 Mismatches

When two sequences are aligned they sometimes do not match perfectly. Some of the bases in the sequences may differ while others align. The positions of the bases where

the two sequences differ are called *mismatches*. Figure 5.2 shows a mismatch between two sequences s_1 and s_2, where both the sequence s_1 and s_2 align perfectly except for one base pair. Mismatches are introduced due to errors in the datasets or biological variations in the genome such as SNPs. The mismatch rate is a parameter to evaluate the quality of assembled contigs. The *mismatch rate* of a set C of contigs relative to a reference genome $\mathscr{G}$ is the ratio of the total number of mismatches in all the contigs when aligned to $\mathscr{G}$ to the sum of the lengths of all the contigs in C.

```
s1 : AGCTAAGCATTTACGATAGCCGATAGCTAAATTAC
     ||||||||||||||| ||||||||||||||||||
s2 : AGCTAAGCATTTACGCTAGCCGATAGCTAAATTAC
```

Figure 5.2: A mismatch between sequences s_1 and s_2.

5.3.3 Contigs

A contig is a continuous subsequence of a DNA molecule. Due to the complex structure of DNA, sequencing errors, and lack of coverage in certain parts of the DNA, genome assemblers are many times unable to reconstruct the entire DNA sequence of the genome. However, they can assemble long subsequences or contigs. For example, the string spelled by each edge of the overlap graph represents a subsequence of the genome, so each one of the strings represents a contig. Similarly, the string represented by a node of a de Bruijn graph is a contig. The goal of the genome assemblers is to produce contigs that are as long as possible and that reconstruct the largest possible part of the genome.

5.3.4 Scaffolds

Sometimes genome assemblers can determine the order and orientation of the contigs based on mate pair information; such ordered contigs are called scaffolds. There can be gaps between the consecutive contigs in the scaffolds; however, the lengths of the gaps are usually smaller than the insert size of the dataset from which the scaffolds were

produced. In other words, a scaffold is a set of contigs that are in the right order but not necessarily connected in one continuous stretch of a DNA sequence. Sometimes, genome assemblers are unable to connect contigs because of lack of coverage between a pair of contigs. However, when a pair of contigs is supported by mate pairs, their order and orientation can be determined in the scaffolds. In Figure 5.3, there are two contigs c_1 and c_2 that were generated by overlapping several reads from the dataset. These two contigs are not connected by overlapping reads, because there is no overlapping reads from the genome that was sampled between the two contigs. However, the contigs c_1 and c_2 are supported by mate pairs. From the mate pair support, we know the order and orientation of the contigs c_1 and c_2 in the reference genome. Hence, c_1 and c_2 are merged together to form a scaffold. However, we do not know the bases between c_1 and c_2 in the scaffold as there is no read present from that region of the genome. When merging two contigs with known gap length between them, genome assemblers usually inset `Ns` between them to represent the unknown base pairs. The terms contig and scaffold are often used interchangeably to denote a subsequence of the genome.

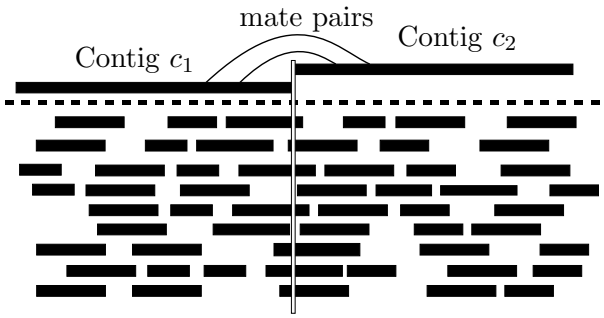


Figure 5.3: A pair of contigs supported by mate pairs to form a scaffold.

5.3.5 N50

N50 is the main parameter for comparing assembly results. Assuming that an assembler produces a set $C = \{c_1, c_2, ..., c_k\}$ of contigs, where the length of the contig c_i is l_i. If the sum of the lengths of all of the contigs is $L = \sum_{i=0}^{k} l_i$, then the N50 of the set C

is $N50(C) = \max_l \sum_{l_i \geq l} l_i \geq \frac{1}{2}L$. In other words, the N50 is defined as the maximum length l for which at least 50% of all bases in C are in contigs of length $\geq l$. Similarly, the N80 is defined as the maximum length l for which at least 80% of all bases in C are in contigs of length $\geq l$ and the N20 is defined as the maximum length l for which at least 20% of all bases in C are in contigs of length $\geq l$.

5.4 Discussion

In this section, we compare the assembly results for all the 16 datasets mentioned in Table 5.1 produced by PEGASUS, ABySS 1.3.4 [46], and SOAPdenovo 1.0.5 [22]. Different overlap lengths for PEGASUS and k-mer sizes for ABySS and SOAPdenovo were used to run the programs on each of the datasets. For each of the datasets, the best results for the three assemblers are reported in Table 5.2 and Table 5.3.

To test the quality of assembly, the assembled contigs were aligned to their corresponding reference genomes by using BWA-SW [21]. Default parameters of BWA-SW were used to align the contigs to the reference genome. In the tables of this section, `n` denotes the total number of contigs, `n:N50` denotes the number of contigs of length at least N50, `max` denotes the length of the longest contig, `sum` denotes the sum of the lengths of all the contigs in Mbp. Finally, `MMR` denotes the average number of mismatches per 1000 base pairs. For some of the datasets, the reference genomes seem to be rather different from the genomes from which the datasets were obtained. For this reason parts of the contigs did not align properly to the reference genome. For mismatch rate calculation, we consider only the contigs having less than 1% mismatch rate as usually reported in the literature.

The results reported in the tables for each of the datasets are calculated using the `abyss-fac` tool provided by ABySS 1.3.4. The tool `abyss-fac` assumes the genome size to be the sum of the lengths of all the contigs and calculate N80, N50, and N20 accord-

ingly. The row labelled `Orig.` in the tables represents the results for the original output files reported by the assemblers and the row labelled `BWA` denotes the same results after aligning the contigs to their corresponding reference genomes using BWA-SW. BWA-SW breaks the contigs from the original contig files if there are certain number of insertions, deletions or mismatches. The best results in Table 5.2 and Table 5.3 are shown in bold.

The dataset (`SRR065390`) of C.elegans is the largest of all the datasets that we have tested. This is also the most important dataset as it is used for many of the genome assemblers to test the performance of their assembly results. The size of the input file is about 22 GB. The contigs produced by PEGASUS for this dataset are the best, both before and after aligning to the reference genome (`build WS222`, `www.wormbase.org`). The assembly results did not change much (compared to the other datasets) after aligning the contigs to the reference genome. This means that the dataset was likely sequenced from a very similar genome of the reference genome (`build WS222`). PEGASUS and ABySS produced similar longest contigs (about 384 kbp). However, after aligning the contigs to the reference genome the longest contig of ABySS is broken into smaller pieces than the longest contig of PEGASUS. For this dataset PEGASUS performs the best in all the evaluation parameters used. PEGASUS also performs the best for mismatch rate (0.385 mismatches per 1000 base pairs) for this dataset.

For the B.subtilis dataset, the assembly results of PEGASUS are better than those of ABySS and SOAPdenovo, both before and after BWA alignment. PEGASUS has only 2 contigs of length at least N50 both before and after BWA alignment. On the other hand, ABySS and SOAPdenovo both have 3 contigs of length at least N50. For this dataset, PEGASUS also performs better than the other two assemblers when comparing N80, N20 and max, both before and after BWA alignment. Similarly contigs produced by PEGASUS are better that the contigs of ABySS and SOAPdenovo for the datasets of S.oralis, H.pylori, P.gingivalis, S.pseudopneumonie and H.parainfluenzae in terms of N80, N50, N20 and max. In all of these datasets PEGASUS contigs are better than the contigs

produced by ABySS and SOAPdenovo after BWA alignment in terms of N50. All the assemblers produced nearly identical results for S.oralis2 dataset after BWA alignment.

For the datasets of L.interrogans, E.coli and S.mitis contigs produced by PEGASUS are the best in terms of N50 after aligning the contigs to the reference genome. The N20s of PEGASUS after BWA alignment are better than those of ABySS and SOAPdenovo for E.coli and S.mitis datasets. For C.thermocellum dataset the max of PEGASUS is the best both before and after BWA alignment; however N80, N50 and N20 are not as good as the other assemblers. For C.tracomatis1 dataset PEGASUS produces very good contigs. But after BWA alignment the contigs are broken into smaller pieces. For tracomatis1 dataset N20 and max of PEGASUS is the best after BWA alignment and the other evaluation parameters are comparable. For the C.tracomatis2 dataset ABySS produces by far the best contigs. ABySS also produced the best assembly results for the datasets of F.tularemia and S.parasanguinis datasets. However, the results of PEGASUS after aligning the contigs to the reference genome are comparable.

For many of the datasets we observed that the contigs produced by all the three assemblers were broken into many smaller contigs. This is because the reference genome is very different from the genome from which the reads were taken.

Based on the original assembly results for the 16 datasets, the N50 of PEGASUS is the best in 8 of the datasets, and the N50 of ABySS is the best in 8 of the datasets. After aligning the contigs to the reference genomes, PEGASUS performs the best in 11 of the datasets in terms of N50, while ABySS performs the best in 7 of the datasets and SOAPdenovo performs the best in 1 of dataset. Note that, there was a tie in one of the datasets for the best N50.

Table 5.2: Assembly results of the first 8 datasets. For each of the assemblers the original results and the results after BWA alignments are shown. Best results of each of the datasets are shown in bold.

			n	**n:N50**	**N80**	**N50**	**N20**	**max**	**sum**	**MMR**
C.ele	PEG	Orig.	14663	746	**13,181**	**37,455**	**82,842**	383,317	102.80	
		BWA	17699	941	**9,789**	**28,007**	**63,882**	**360,365**	99.19	**0.385**
	ABy	Orig.	56127	1027	9,346	26,012	59,794	**384,441**	101.50	
		BWA	59830	1249	6,789	20,964	48,194	213,123	99.70	0.526
	SOA	Orig.	35187	798	12,238	33,101	74,486	236,372	99.10	
		BWA	43084	1217	7,367	21,696	48,946	173,090	98.90	0.813
B.sub	PEG	Orig.	66	2	**440,869**	**1,046,994**	**1,193,680**	**1,193,680**	4.18	
		BWA	74	2	**359,408**	**1,015,697**	**1,123,279**	**1,123,279**	4.18	**0.046**
	ABy	Orig.	74	3	210,409	897,049	1,046,847	1,046,847	4.19	
		BWA	71	3	210,409	650,686	976,267	976,267	4.45	0.112
	SOA	Orig.	104	3	250,276	918,328	1,015,275	1,015,275	4.16	
		BWA	112	3	250,276	918,328	1,015,171	1,015,171	4.16	0.061
S.ora1	PEG	Orig.	26	2	**422,338**	**622,598**	**738,904**	**738,904**	1.97	
		BWA	256	35	6,504	**16,368**	26,714	**44,056**	1.68	**3.963**
	ABy	Orig.	40	2	260,869	424,811	732,484	732,484	1.99	
		BWA	257	36	**7,670**	16,113	**26,816**	**44,056**	1.71	7.190
	SOA	Orig.	78	2	259,131	371,414	734,867	734,867	1.96	
		BWA	276	35	6,554	15,756	**26,816**	**44,056**	1.67	5.567
S.ora2	PEG	Orig.	24	1	**174,123**	**1,339,138**	**1,339,138**	**1,339,138**	2.08	
		BWA	314	45	4,736	**10,321**	**19,991**	**52,494**	1.57	**1.792**
	ABy	Orig.	41	2	148,666	542,469	883,474	883,474	2.14	
		BWA	311	45	**4,980**	**10,321**	**19,991**	**52,494**	1.58	6.987
	SOA	Orig.	89	2	78,444	519,053	883,455	883,455	2.04	
		BWA	343	44	4,784	**10,321**	**19,991**	**52,494**	1.56	3.489
H.py1	PEG	Orig.	91	9	**32,409**	**66,326**	**132,391**	**205,501**	1.67	
		BWA	298	35	**5,772**	**13,670**	22,943	**66,209**	1.58	9.804
	ABy	Orig.	87	11	28,276	59,423	82,140	145,043	1.70	
		BWA	311	36	5,520	13,306	**25,409**	**66,209**	1.63	**5.102**
	SOA	Orig.	187	11	26,323	58,234	83,260	113,967	1.65	
		BWA	384	34	5,516	13,496	30,490	**66,209**	1.59	7.092
P.gin	PEG	Orig.	141	11	**25,126**	**58,241**	**161,578**	**186,339**	2.30	
		BWA	178	12	**19,224**	**50,986**	**161,578**	168,386	2.31	0.248
	ABy	Orig.	287	14	20,005	48,462	87,408	176,481	2.33	
		BWA	318	16	16,711	35,564	87,408	**172,567**	2.34	0.154
	SOA	Orig.	481	13	19,982	49,125	142,720	185,001	2.25	
		BWA	528	15	15,252	35,789	87,448	167,717	2.25	**0.149**
S.pse	PEG	Orig.	165	11	**29,542**	62,912	152,815	**203,490**	2.19	
		BWA	545	44	6,851	**14,762**	25,919	**55,462**	2.11	4.344
	ABy	Orig.	239	11	22,771	**63,534**	120,870	203,373	2.17	
		BWA	612	46	6,399	13,427	25,716	**55,462**	2.11	3.745
	SOA	Orig.	558	11	22,498	59,758	**153,195**	202,356	2.12	
		BWA	882	44	**6,796**	14,220	**26,980**	53,588	2.05	**3.454**
H.par	PEG	Orig.	81	3	**183,716**	**347,776**	**529,188**	**529,188**	2.14	
		BWA	232	31	8,360	**17,943**	33,434	52,260	1.79	4.874
	ABy	Orig.	49	3	137,600	346,176	528,624	528,624	2.31	
		BWA	206	33	8,360	**17,943**	**35,135**	52,260	1.93	8.442
	SOA	Orig.	117	3	153,303	345,717	528,306	528,306	2.12	
		BWA	244	31	**8,545**	17,940	33,662	**52,266**	1.78	**3.799**

Table 5.3: Assembly results of the next 8 datasets. For each of the assemblers the original results and the results after BWA alignments are shown. Best results of each of the datasets are shown in bold.

			n	n:N50	N80	N50	N20	max	sum	MMR
L.int	PEG	Orig.	124	8	**86,549**	183,250	350,144	550,958	4.74	
		BWA	197	14	41,883	**103,148**	183,250	205,999	4.32	**0.131**
	ABy	Orig.	192	7	63,178	**196,390**	**509,227**	**529,258**	4.82	
		BWA	280	14	**42,240**	92,880	**196,390**	**336,051**	4.44	0.198
	SOA	Orig.	438	10	53,553	133,049	256,391	350,138	4.59	
		BWA	548	22	35,969	70,972	107,887	195,369	4.52	0.328
E.col	PEG	Orig.	270	11	57,505	156,402	217,762	326,281	4.48	
		BWA	371	21	31,716	**69,667**	**98,808**	269,676	4.31	0.341
	ABy	Orig.	473	10	**69,165**	**176,414**	**260,541**	**329,829**	4.51	
		BWA	493	26	30,712	56,914	98,641	245,741	4.64	0.398
	SOA	Orig.	159	12	57,623	136,185	178,083	326,225	4.45	
		BWA	262	22	**32,514**	63,694	112,265	**273,657**	4.42	**0.137**
S.mit	PEG	Orig.	87	5	**88,557**	148,205	252,512	273,544	2.05	
		BWA	521	76	2,921	**6,318**	**10,812**	**26,758**	1.56	**4.644**
	ABy	Orig.	169	4	86,885	**193,795**	**436,949**	**436,949**	2.07	
		BWA	530	78	**3,044**	6,232	**10,812**	**26,758**	1.57	6.703
	SOA	Orig.	322	5	81,094	146,871	252,370	270,672	2.01	
		BWA	568	75	2,996	6,233	10,747	**26,758**	1.52	5.913
C.the	PEG	Orig.	402	20	**23,922**	55,448	103,270	**186,731**	3.75	
		BWA	633	25	19,457	43,311	87,921	**186,731**	3.80	0.554
	ABy	Orig.	3219	18	23,858	**59,484**	**136,725**	186,547	3.76	
		BWA	1693	19	**23,276**	**56,326**	**120,707**	186,547	3.76	0.178
	SOA	Orig.	332	18	23,398	59,433	102,610	186,442	3.60	
		BWA	431	20	20,753	52,130	102,610	176,549	3.64	**0.159**
C.tra1	PEG	Orig.	27	1	**1,010,825**	**1,010,825**	**1,010,825**	**1,010,825**	1.06	
		BWA	46	3	56,438	154,558	**245,614**	**245,614**	1.12	6.250
	ABy	Orig.	3922	1	213,253	798,261	798,261	798,261	1.05	
		BWA	318	4	**62,806**	**165,071**	178,526	207,913	1.19	6.681
	SOA	Orig.	109	5	24,688	84,969	118,975	177,268	1.04	
		BWA	112	13	10,230	19,628	54,236	89,681	1.05	**5.356**
C.tra2	PEG	Orig.	46	2	145,330	223,073	310,060	310,060	1.05	
		BWA	44	6	29,194	62,513	123,527	168,826	1.14	6.023
	ABy	Orig.	5600	1	**198,739**	**688,592**	**688,592**	**688,592**	1.07	
		BWA	291	3	**59,180**	**170,857**	**198,739**	**223,595**	1.15	7.039
	SOA	Orig.	99	4	54,168	142,863	168,057	171,997	1.04	
		BWA	58	8	22,627	37,507	122,229	153,885	1.09	**5.858**
F.tul	PEG	Orig.	222	24	13,000	25,457	36,823	87,443	1.80	
		BWA	341	34	8,009	17,767	33,545	50,708	1.84	4.301
	ABy	Orig.	131	24	15,066	**27,975**	**44,111**	**88,166**	1.87	
		BWA	284	32	**8,583**	**19,329**	**34,887**	**56,155**	1.90	4.412
	SOA	Orig.	310	23	**16,053**	26,799	43,438	87,411	1.77	
		BWA	475	37	7,379	16,020	27,293	50,657	1.79	**4.256**
S.par	PEG	Orig.	129	5	36,666	158,576	325,943	335,606	2.26	
		BWA	330	34	6,478	17,002	35,557	**63,509**	1.94	**4.674**
	ABy	Orig.	47	3	**128,047**	**327,687**	**447,335**	**447,335**	2.12	
		BWA	235	29	**8,297**	**20,236**	**37,795**	**63,509**	1.86	5.756
	SOA	Orig.	119	3	121,581	325,452	445,528	445,528	2.11	
		BWA	280	30	8,095	19,751	35,466	53,085	1.84	5.148

To summarize the results, we combined in Table 5.4 all the assembly evaluation parameters from Table 5.2 and Table 5.3. From Table 5.4, we can see that PEGASUS performs better for all the evaluation parameters than ABySS and SOAPdonovo for the 16 datasets. The sum of original N50s for PEGASUS is about 12% larger than that of ABySS and about 38% larger than that of SOAPdenovo. Whereas after BWA alignment the sum of N50s for PEGASUS is about 16% larger than ABySS and about 18% larger than SOAPdenovo. Similarly, the sum of original max values for PEGASUS is about 3% larger than that of ABySS and about 24% larger than that of SOAPdenovo. Whereas after BWA alignment the sum of max values for PEGASUS is about 0.5% larger than ABySS and about 11% larger than SOAPdenovo.

Table 5.4: Summary of the assembly results.

		n:N50	**N80**	**N50**	**N20**	**max**	**sum**
PEG	Original	114	**2,782,137**	**5,525,233**	**6,897,812**	**7,481,236**	37,763,230
	BWA	422	**606,951**	**1,614,031**	**2,232,892**	**2,779,967**	35,576,143
ABy	Original	116	1,626,787	4,850,522	6,803,588	7,221,263	38,033,755
	BWA	416	498,187	1,345,205	2,073,239	2,765,584	36,049,870
SOA	Original	113	1,242,115	3,377,250	5,106,530	5,678,908	32,474,325
	BWA	416	438,286	1,311,785	1,813,291	2,471,142	32,769,668

5.4.1 Running Time and Memory Usage Comparison

Table 5.5 shows the running time and Table 5.6 shows the memory used by the assemblers for assembling different datasets measured in `SHARCNET`'s cluster `redfin`. The running times in minutes and the memory usage in megabytes (MB) were reported by the job scheduler of `redfin`. Note that, `redfin` does not continuously monitor the memory usage of running programs. It checks the memory usage by a running program in different intervals. The actual memory usage by a program can vary, though it should be very close to the reported memory usage. From Table 5.5 and Table 5.6 we can see that SOAPdenovo is the most time efficient, but it uses the most amount of memory.

PEGASUS used the least amount of memory in 6 of the datasets, whereas ABySS used the least amount of memory in 9 of the datasets.

Table 5.5: Running time comparison in minutes.

Organism	**PEGASUS**	**ABySS**	**SOAPdenovo**
C.elegans	**216.0**	360.0	558.0
B.subtilis	**4.5**	13.0	5.0
H.parainfluenzae	**5.9**	15.0	6.0
H.pylori	**6.1**	27.0	13.0
F.tularemia	11.0	24.0	**10.0**
S.mitis	5.7	11.0	**4.0**
S.parasanguinis	6.6	11.0	**4.0**
S.oralis	5.5	13.0	**4.2**
S.pseudopneumonie	5.8	10.0	**3.3**
S.oralis2	7.0	16.0	**6.4**
L.interrogans	10.0	19.0	**6.5**
P.gingivalis	11.0	18.0	**6.2**
C.trachomatis	12.0	18.0	**4.4**
C.trachomatis2	20.0	30.0	**6.1**
E.coli	45.0	51.0	**12.0**
C.thermocellum	150.0	72.0	**16.0**

With respect to the N50 and max parameters, we can see that PEGASUS performs better than ABySS and SOAPdenovo on the majority of the datasets. However, the running time of SOAPdenovo is better than the running time of ABySS and PEGASUS.

Table 5.6: Memory usage comparison in megabytes.

Organism	**PEGASUS**	**ABySS**	**SOAPdenovo**
B.subtilis	**687**	1,260	5,841
C.trachomatis1	**738**	2,550	6,398
C.trachomatis2	**979**	3,822	9,876
F.tularemia	**730**	750	9,657
H.parainfluenzae	**730**	785	9,742
H.pylori	**753**	1246	9,689
C.elegans	16,660	**15,082**	42,125
E.coli	2,589	**1,786**	11,663
S.parasanguinis	719	**416**	6,733
S.mitis	695	**448**	4,574
S.pseudopneumonie	672	**386**	4,650
S.oralis1	725	**555**	6,913
S.oralis2	928	**426**	6,910
L.interrogans	1,218	**736**	8,460
C.thermocellum	6,505	**1,454**	9,330
P.gingivalis	996	**589**	7,144

Chapter 6

Conclusions and Future Research

As mentioned before DNA assembly plays an important role in a wide variety of fields such as forensic science, medicine and agriculture. There are increasing demands for better genome assemblers. Many new genome assemblers have been developed over the past few years to better assemble the reads produced by next generation sequencing techniques.

Many research groups worldwide are working on building better genome assemblers. A group of researchers at the Beijing Genomic Institute (`www.genomics.cn`) developed the de Bruijn graph-based genome assembler SOAPdenovo. The Genome Science Center in Vancouver (`www.bcgsc.ca`) developed ABySS. These research groups are still working on improving their assemblers and they periodically release new versions of their assemblers. De Bruijn graph-based genome assemblers such as ABySS and SOAPdenovo are considered to be the best genome assemblers.

Since PEGASUS is an overlap graph based genome assembler, we faced several challenges when building the overlap graph. Initially we used a suffix array [23] to find all pairs of overlapping reads in the datasets. This required $O(nl)$ memory, where n in the number of reads in the dataset and l is the read length. For the C.elegans dataset our program required about 108 GB of memory just to store the suffix array. When working

with the C.elegans dataset we realized that a suffix array is not a good choice for finding overlapping pairs of reads. We then implemented a hash table to find overlaps between reads, which requires only $O(n)$ memory.

Moreover, the number of edges in an overlap graph can be very large. For large datasets the overlap graph is the bottleneck with respect to memory consumption. The largest amount of memory in PEGASUS is used by the overlap graph. We did not face any memory issues for small datasets, however, for the C.elegans dataset PEGASUS was using more than 100 GB of memory to store the overlap graph. We observed that a significant portion of the memory in the overlap graph is used to store the edges. Most of these edges are removed from the overlap graph in the transitive edge removal step. Earlier implementations of PEGASUS first built the overlap graph without removing transitive edges while building it. To save memory, we modified the algorithm in such a way that it removes the transitive edges while building the overlap graph. This way the number of edges in the graph is reduced, which in turn reduced the memory usage of PEGASUS.

Better estimation of the copy counts for the reads in the dataset can produce better contigs. Instead of explicitly computing copy counts of the reads, ABySS and SOAPdenovo rely on mate pairs to assemble contigs. This approach may perform poorly in case of lack of coverage in some regions. PEGASUS uses a minimum cost flow network to overcome this problem.

Each of the existing assemblers has its limitations. For example, experimental results in Chapter 5 show that SOAPdenovo is the most time efficient. However, SOAPdenovo requires more space and the assembled contigs are not as good as the other assemblers. On the other hand, ABySS uses less space, but for some of the datasets the contigs produced by ABySS are not good.

As mentioned before, the main goal of PEGASUS is to build an assembler that produces more accurate result in feasible amount of time and memory. Experimental results

in Chapter 5 show that the contigs produced by PEGASUS are the best for the majority of the datasets used in terms of the N50. PEGASUS performs better than ABySS and SOAPdenovo for the most important dataset of C.elegans in terms of N80, N50, N80 and max. The running time and memory usage of PEGASUS are also reasonable. PEGASUS performs better than other top assemblers due to the accurate estimation of the copy counts and the way in which it searches for paths in the overlap graph.

Some of the algorithms used in PEGASUS (e.g., bubble and dead-end removal) are also used by other assemblers. However, there are many novel algorithms, for example the algorithm to build the overlap graph and the algorithm to compute the copy counts for the reads. These algorithms work together to assemble genomes more efficiently than other existing genome assemblers.

Contigs assembled by PEGASUS are better than ABySS and SOAPdenovo in most of the input datasets used for our experiments. The major goal of any assembler is to produce quality contigs with better N50 using a reasonable amount of time and memory. Running time and memory used by PEGASUS are comparable with other genome assemblers.

6.1 Future Research

There are several ways in which PEGASUS could be improved. Since the reads in a dataset contain errors, the number of edges incident on a node in the overlap graph could be large. PEGASUS searches for all paths between the two reads of a mate pair to decide whether some of the contigs could be merged. The running time of PEGASUS is not good if the underlying graph has densely connected regions. Searching for all paths in the overlap graph is not always feasible as there can be exponentially many of them. We could try different heuristics to reduce the number of paths to be searched in the overlap graph.

The current version of PEGASUS works if all the reads in the input dataset have the same length. Most datasets from the SRA database have the same length for all the reads, however, read lengths vary from one dataset to another. If we take two datasets of the same organism from the SRA database that have different lengths, then PEGASUS will not work. In the future PEGASUS should work with reads of different lengths.

When a pair of contigs are merged leaving a gap between them, assemblers try to infer the sequence of bases in a gap. ABySS and SOAPdenovo can infer the bases in a gap better than PEGASUS. Procedures for filling the gaps between the contigs in PEGASUS can be improved.

To improve the running time of PEGASUS, in the future we plan to implement the algorithm in parallel mode. Most of the time in PEGASUS is spent in building the overlap graph and searching for paths in the graph. Both the graph building algorithm and path searching algorithm in PEGASUS need to be parallelized in the future.

Appendix A

PEGASUS Software Manual

A.1 Running PEGASUS

The command line to run PEGASUS is:

```
> ./pegasus <numberOfLibraries> <inputFile> ... <dirName> <minOverlap>
```

Where `numberOfLibraries` is the number of input datasets followed by the libraries, `inputFile` is the FASTA/FASTQ file that contain the reads, `dirName` is a name given by the user for output file and folder names. A folder `dirName` is created in the current directory, and all the files generated by PEGASUS are stored in the `dirName` directory. Finally, the `minOverlap` is the minimum length two reads must overlap to have an edge in the overlap graph. Note that `minOverlap` must be smaller than the read length.

A.1.1 Output Files

PEGASUS produces several output files in the directory `dirName`. There are many intermediate files generated by PEGASUS. At the end of each intermediate step, it outputs the contig files (`.fasta`) and the overlap graph files (`.gdl`). Some verbose output is generated in the `dirName/dirName.txt` file. Final contig file is produced in the `dirName/dirName-PE.fasta` file. The overlap graphs in `.gdl` files can be viewed in

freely available software aisee. Aisee can be downloaded from `http://www.absint.com/aisee/`.

A.2 External Function

PEGASUS uses external function CS2 [13] for computing minimum cost flow in the overlap graph. CS2 was downloaded from `http://www.igsystems.com/cs2/` and modified to fit into PEGASUS.

Bibliography

[1] A.V. Aho, M.R. Garey, and J.D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2):131–137, 1972.

[2] 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*, 12(1):177–189, 2002.

[3] M. Burrows and D.J. Wheeler. A block-sorting lossless data compression algorithm. *SRC Research Report*, 124, 1994.

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

[5] M.J. Chaisson, D. Brinza, and P.A. Pevzner. De novo fragment assembly with short mate-paired reads: Does the read length matter? *Genome research*, 19(2):336–346, 2009.

[6] M.J. Chaisson and P.A. Pevzner. Short read fragment assembly of bacterial genomes. *Genome research*, 18(2):324–330, 2008.

[7] F.S. Collins, E.S. Lander, J. Rogers, R.H. Waterston, and I. Conso. Finishing the euchromatic sequence of the human genome. *Nature*, 431(7011):931–945, 2004.

[8] F.S. Collins, M. Morgan, and A. Patrinos. The Human Genome Project: lessons from large-scale biology. *Science*, 300(5617):286–290, 2003.

[9] A. Edwards, H. Voss, P. Rice, A. Civitello, J. Stegemann, C. Schwager, J. Zimmermann, H. Erfle, C.T. Caskey, and W. Ansorge. Automated DNA sequencing of the human HPRT locus. *Genomics*, 6(4):593–608, 1990.

[10] P. Ferragina, T. Gagie, and G. Manzini. Lightweight data indexing and compression in external memory. *Algorithmica*, 63(3):707–730, 2012.

[11] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, page 390. IEEE Computer Society, 2000.

[12] H.N. Gabow. An efficient reduction technique for degree-constrained subgraph and bidirected network flow problems. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 448–456. ACM, 1983.

[13] A.V. Goldberg. An efficient implementation of a scaling minimum-cost flow algorithm. *Journal of Algorithms*, 22(1):1–29, 1997.

[14] 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*, 18(5):802–809, 2008.

[15] D.S. Hochbaum. Monotonizing linear programs with up to two nonzeroes per column. *Operations Research Letters*, 32(1):49–58, 2004.

[16] R.M. Idury and M.S. Waterman. A new algorithm for DNA sequence assembly. *Journal of Computational Biology*, 2(2):291–306, 1995.

[17] L. Ilie and M. Molnar. RACER: Rapid and Accurate Correction of Errors in Reads. Submitted 2012.

[18] M. Kasahara and S. Morishita. *Large-scale genome sequence processing*. Imperial College Press, 2006.

[19] J.D. Kececioglu. Exact and approximation algorithms for DNA sequence reconstruction. 1991.

[20] E.S. Lander, L.M. Linton, B. Birren, C. Nusbaum, M.C. Zody, J. Baldwin, K. Devon, K. Dewar, M. Doyle, W. FitzHugh, et al. Initial sequencing and analysis of the human genome. *Nature*, 409(6822):860–921, 2001.

[21] H. Li and R. Durbin. Fast and accurate short read alignment with Burrows–Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[22] R. Li, H. Zhu, J. Ruan, W. Qian, X. Fang, Z. Shi, Y. Li, S. Li, G. Shan, K. Kristiansen, et al. De novo assembly of human genomes with massively parallel short read sequencing. *Genome research*, 20(2):265–272, 2010.

[23] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[24] G. Manzini and P. Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40(1):33–50, 2004.

[25] E.R. Mardis. The impact of next-generation sequencing technology on genetics. *Trends in genetics*, 24(3):133, 2008.

[26] M. Margulies, M. Egholm, W.E. Altman, S. Attiya, J.S. Bader, L.A. Bemben, J. Berka, M.S. Braverman, Y.J. Chen, Z. Chen, et al. Genome sequencing in microfabricated high-density picolitre reactors. *Nature*, 437(7057):376–380, 2005.

[27] A.M. Maxam and W. Gilbert. A new method for sequencing DNA. *Proceedings of the National Academy of Sciences*, 74(2):560–564, 1977.

[28] P. Medvedev and M. Brudno. Ab initio whole genome shotgun assembly with mated short reads. In *Research in Computational Molecular Biology*, pages 50–64. Springer, 2008.

[29] P. Medvedev and M. Brudno. Maximum likelihood genome assembly. *Journal of computational Biology*, 16(8):1101–1116, 2009.

[30] P. Medvedev, M. Fiume, M. Dzamba, T. Smith, and M. Brudno. Detecting copy number variation with mated short reads. *Genome research*, 20(11):1613–1622, 2010.

[31] P. Medvedev, K. Georgiou, G. Myers, and M. Brudno. Computability of models for sequence assembly. *Algorithms in Bioinformatics*, pages 289–301, 2007.

[32] M.L. Metzker. Emerging technologies in DNA sequencing. *Genome research*, 15(12):1767–1776, 2005.

[33] M.L. Metzker. Sequencing technologiesthe next generation. *Nature Reviews Genetics*, 11(1):31–46, 2009.

[34] E.W. Myers. Toward simplifying and accurately formulating fragment assembly. *Journal of Computational Biology*, 2(2):275–290, 1995.

[35] E.W. Myers. The fragment assembly string graph. *Bioinformatics*, 21(suppl 2):ii79–ii85, 2005.

[36] 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.J. Reinert, K.A. Remington, et al. A whole-genome assembly of Drosophila. *Science*, 287(5461):2196–2204, 2000.

[37] P.A. Pevzner, M.Y. Borodovsky, and A.A. Mironov. Linguistics of nucleotide sequences II: stationary words in genetic texts and the zonal structure of DNA. *Journal of Biomolecular Structure and Dynamics*, 6(5):1027–1038, 1989.

[38] P.A. Pevzner and H. Tang. Fragment assembly with double-barreled data. *Bioinformatics*, 17(1):S225–S233, 2001.

[39] P.A. Pevzner, H. Tang, and M.S. Waterman. A new approach to fragment assembly in DNA sequencing. In *Proceedings of the fifth annual international conference on Computational biology*, pages 256–267. ACM, 2001.

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

[41] D. Pushkarev, N.F. Neff, and S.R. Quake. Single-molecule sequencing of an individual human genome. *Nature biotechnology*, 27(9):847–850, 2009.

[42] M. Ronaghi, S. Karamohamed, B. Pettersson, M. Uhlén, P. Nyrén, et al. Real-time DNA sequencing using detection of pyrophosphate release. *Analytical biochemistry*, 242(1):84–89, 1996.

[43] F. Sanger, S. Nicklen, and A.R. Coulson. DNA sequencing with chain-terminating inhibitors. *Proceedings of the National Academy of Sciences*, 74(12):5463–5467, 1977.

[44] J.T. Simpson and R. Durbin. Efficient construction of an assembly string graph using the FM-index. *Bioinformatics*, 26(12):i367–i373, 2010.

[45] J.T. Simpson and R. Durbin. Efficient de novo assembly of large genomes using compressed data structures. *Genome research*, 22(3):549–556, 2012.

[46] J.T. Simpson, K. Wong, S.D. Jackman, J.E. Schein, S.J.M. Jones, and İ. Birol. ABySS: a parallel assembler for short read sequence data. *Genome research*, 19(6):1117–1123, 2009.

[47] L.D. Stein. The case for cloud computing in genome informatics. *Genome Biology*, 11(5):207, 2010.

[48] J.C. Venter, M.D. Adams, E.W. Myers, P.W. Li, R.J. Mural, G.G. Sutton, H.O. Smith, M. Yandell, C.A. Evans, R.A. Holt, et al. The sequence of the human genome. *Science*, 291(5507):1304, 2001.

[49] K.V. Voelkerding, S.A. Dames, and J.D. Durtschi. Next-generation sequencing: from basic research to diagnostics. *Clinical chemistry*, 55(4):641–658, 2009.

[50] J.D. Watson and F.H.C. Crick. Molecular structure of nucleic acids: a structure for deoxyribose nucleic acid. *Nature*, 171:737–738, 1953.

[51] K.A. Wetterstrand. DNA Sequencing Costs: Data from the NHGRI Large-Scale Genome Sequencing Program. Available at: `www.genome.gov/sequencingcosts`. Accessed [November 2, 2012].

[52] R. Williams, S.G. Peisajovich, O.J. Miller, S. Magdassi, D.S. Tawfik, and A.D. Griffiths. Amplification of complex gene libraries by emulsion PCR. *Nature methods*, 3(7):545–550, 2006.

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

Index

Curriculum Vitae

Name:	Md. Bahlul Haider
Post-Secondary Education and Degrees:	The University of Western Ontario London, Canada 2009 - 2012 PhD in Computer Science University of Tokyo Tokyo, Japan 2006 - 2008 MSc in Mathematical Informatics Islamic University of Technology Gazipur, Bangladesh 2000 - 2003 BSc in Computer Science and Information Technology
Honours and Awards:	Ontario Graduate Scholarship 2011-2012
Related Work Experience:	Teaching Assistant Department of Computer Science The University of Western Ontario 2009 - 2012