UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

CIGARCoil: A New Algorithm for the Compression of DNA Sequencing Data

A THESIS

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCE

By

Addison Edward Womack
Norman, Oklahoma
2019

CIGARCoil: A New Algorithm for the Compression of DNA Sequencing Data

A THESIS APPROVED FOR THE
SCHOOL OF COMPUTER SCIENCE

BY

Dr. Sridhar Radhakrishnan, Chair

Dr. Chongle Pan

Dr. Christan Grant

Abstract

DNA sequencing machines produce tens of thousands to hundreds of millions of reads. Each read consists of letters from the alphabet X= {A, T, C, G, N} and varies in length between 30 to 120 characters and beyond. The DNA reads are stored in a standard FASTQ file format that contains not only the reads but also a quality score for each character in each read that corresponds to the probability that the identified character is correct. The FASTQ files vary in size between 100s of megabytes to 10s of gigabytes. The reads in the FASTQ files are processed as part of many DNA algorithms for various sequence analyses. Given the fact that the size of each file is considerable, keeping and handling multiple of these files in main memory for faster processing is not possible on commodity hardware. In this thesis, we propose a lossless compression mechanism named CIGARCoil that operates on the FASTQ files and other files that contain the DNA reads. The other salient features of CIGARCoil are:

- It is a not a reference-based algorithm in the sense that one does not need to create a reference string before the compression can begin. Reference strings are undesirable due to them not only being hard to determine, but also due to them being required for both the compression and decompression of the file.

- In this thesis, for the first time, we show that each of the reads can be accessed directly on the compressed structure created by CIGARCoil. That is, we provide access to each read without having to uncompress the file.

- Since we can provide direct access to a read on the CIGARCoil compressed structure, we have implemented a [] (square-bracket) array indexing operator. Through this implementation, we can implement a predictive caching mechanism that will make the reads available for the end-user based on their access pattern.

We have analyzed our compressed mechanism on various well-known FASTQ data sets along with synthetic data sets. In all cases, our compression method produces a compressed file that is smaller or approximately the same size as ones created by the existing DNA compression mechanisms, including BZIP, DSRC2, and LFQC.

Acknowledgements

First, I would like to thank my advisor and mentor, Dr. Sridhar Radhakrishnan, for not only supporting and guiding me through this research but also providing me with numerous opportunities to become a better computer scientist and constantly encouraging his students to think about problems more critically and thoroughly.

Second, I would like to thank the other students that have been involved in Dr. Radhakrishnan's research group for their insight, criticism, and words of encouragement over the past year-and-a-half. These students are: Sudhindra Gopal, Aditya Narasimhan, Aaron Morris, Dorian Selimovic, Dwaine Kenney, and Michael Nelson.

Third, I would like to thank Dr. Naijia Xiao from the microbiology department for assisting me in both obtaining sequencing data as well as helping me understand it.

Finally, I would like to acknowledge my parents for their constant love and support during my formative years and my time as a student here at OU.

Contents

Chapter 1

Introduction

In this chapter we provide a brief background and overview of topics related to DNA sequencing and compression. Then we discuss the motivation for this thesis. Then, we provide a set of preliminary algorithms and theorems that are referenced throughout the remainder of the paper. Finally, we list the set of contributions made in this thesis.

1.1 Background

Deoxyribonucleic Acid (DNA) or sometimes given by the double helix figure is self-replicating and the basis of all living organisms. From the computer scientist's perspective we process and manipulate strings from the chosen set of alphabets. Most DNA processing, such as genome assembly, involves string searching and replacement algorithms [24]. The most challenging problem is the genome assembly Problem wherein you are a set of strings and you are to combine these strings and/or fragments of the strings to obtain the original DNA string that corresponds to the organism.

In this section, we will introduce you in more detail DNA strings and DNA reads including providing details of how they are captured. Next, we will introduce to you the concept of lossless and lossy compression as it relates to the DNA reads. FASTA is the format in which DNA reads are stored and we will introduce them to you.

1.1.1 DNA Sequencing

This section describes in general terms the process and scale of DNA sequencing.

DNA DNA is a collection of four chemical compounds, called bases. Each base is one of four chemical compounds, adenine, cytosine, guanine, and thymine. which are commonly represented by the four characters A, C, G, and T, respectively. DNA provide instructions that tell an organism's cells how to operate. The complete set of DNA instructions for an organism is called its genome. A genome can be quite large in size. For example, The human genome consists of 2.91 billion base pairs [11].

DNA Sequencing DNA sequencing is the process by which the order of a particular genome is determined. A special tool called a DNA sequencer takes a DNA sample, then generates anywhere from a few hundred thousand to several million reads from the sample. Each read is a string of a length ranging from a few dozen to a few hundred characters, consisting of A, C, T, G, and N. The characters A, C, T, and G correspond

Figure 1.1: Image of Illumina MiSeq Sequencer Courtesy of the University of Oklahoma's Institute of Environmental Genomics

to bases, and the character N corresponds to a base that the DNA sequencer was unsure of while processing the sample. Many millions of reads are required to piece together the genome of the source DNA sample as there are points of overlap between reads and reads have an error rate depending on the quality of the DNA sequencing machine.

DNA Sequencing Machines DNA Sequencers are specialized machines that when provided with a DNA sample are able to generate a file of reads in either FASTA or FASTQ format. Machines such as the Illumina MiSeq sequencer as seen in Figure 1.1 are able produce approximately 200 Megabytes of data per hour.

FASTA Format In FASTA format, each read occupies two consecutive lines. The first line is the id of the read and begins with the @ character. The second line of the read is the base-pair data, the string of characters A, C, T, G, and N. An example of a single FASTA read is:

```
@J00138:116:HKMFNBBXX:8:1101:23338:1033 1:N:0:NCTCTATC
GNCGGATCGTGGTTGATGGCTTCGGTGTGCATGGATTTGATGAT
```

FASTQ Format In the FASTQ format, each read occupies four consecutive lines. The first two lines are the id and base-pair lines as they are in the FASTA format. In addition to these first two lines are an additional '+' line, followed by the read's quality scores. There is one quality score character for each character in the sequence field and the

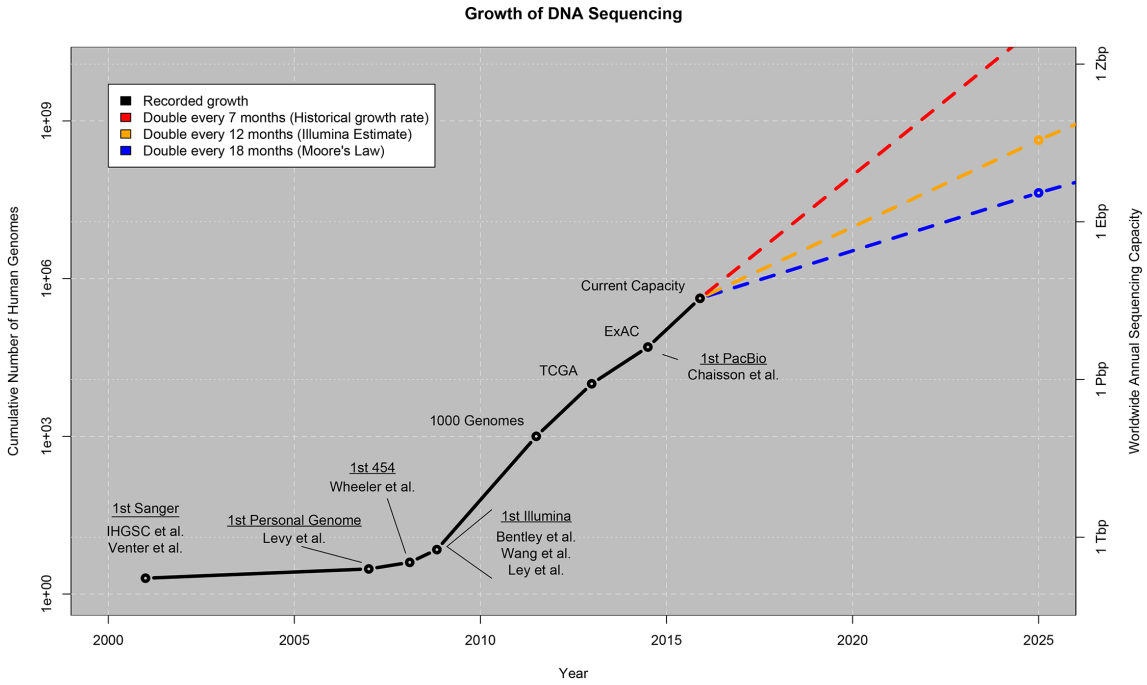

Figure 1.2: Figure 1 from [22]. This Figure shows the combined size of all of the human genomes on the left axis and the quantity of bases that the world is capable of producing in a year on the right axis.

quality score corresponds to how confident the sequencer was of that particular character in the read. The range of values in this quality score field is from [0, 255]; however, there is little standardization of quality scores and their range is often dependent on the DNA sequencing machine being used. An example of a single FASTQ read is:

```
@J00138:116:HKMFNBBXX:8:1101:23338:1033  1:N:0:NCTCTATC
GNCGGATCGTGGTTGATGGCTTCGGTGTGCATGGATTTGATGAT
+
ABAAQCACQQCCCCQQQQCCCQACCQCQCCACQBAQQCBQQQCQQ
```

Next Generation Sequencing With the advent of Next Generation Sequencing in 2008 [21], it became possible to generate huge collections of short-length reads in a massively parallel process. Ever since this innovation, DNA compression algorithms have been racing to catch up to handle this influx of data. According to [5], the costs of data storage has been outpacing innovations in DNA compressors. Additionally, according to [22] DNA sequencing is rapidly becoming Big Data's most significant challenge as the world's DNA sequencing throughput has been increasing at a rate faster than Moore's law since 2008 as seen in Figure 1.2.

1.1.2 Compression and Decompression

This section defines and briefly explains the concepts of encoding and compression as well as their counter-parts, decoding and decompression.

Encoding and Decoding Encoding is placing data into a new representation that supports the compression of the file by either reducing the average number of bits required

to represent a character or representing each character in a way that will support the compression step. An ASCII character requires 8 bits to store. If we assume that there are 26 letters in the Latin alphabet (assuming all lower-case letters from a-z, for example), then the number of bits required to represent (or termed as to encode) is $\lceil \log_2 26 \rceil = 5$ bits. This is smaller than the number of bits in the ASCII encoding. There are other encoding techniques such as Huffman and Arithmetic encoding [19] that takes into account the frequencies of the occurrence of each letter in the string. Letters that are more frequent are given fewer bits compared to the ones that occur less frequently. These compression techniques satisfy a property called the *prefix* property wherein the bit string assigned to any character (or symbol) is not a prefix to bit strings of other characters. This property allows us to decode (getting the original string back). The *decoding* process is to get the letters of the original string. In the case of ASCII encoding, we need the ASCII encoding table and in the case of Huffman encoding we need the Huffman tree (which is stored in some elegant manner). The size of the encoding is the sum of the length of the bit strings of each letter in the string and the size of the encoding table.

Compression is the process of removing redundancy in the data set. This will require that data be encoded in such a way that much of the redundancy can be removed. *Compression Ratio* is defined as the ratio of the size of the compressed file to the size of the uncompressed file. Compression algorithms are measured on other factors as well. For example, one might be interested in the time it takes to complete the compression. Decompression is the process of producing the file from its compressed structure (sometimes referred to as the compression image). Now, decompression time is an additional factor that must be considered when designing compression algorithms. More recently, there has been interest in developing compression techniques that allows for data to be accessed without having to decompress them file. Also, additional efforts to incrementally add data (sometimes referred to as streaming operations [15]) directly to the compressed image are also being considered.

There are two kinds of compression techniques: lossless and lossy. In lossless compression, the original data is preserved in its entirety, for example the popular ZIP file compressor. In lossy compression, some of the original data is lost during the compression of the file, for example the Moving Picture Expert Group's MP4 video compression format. The compression algorithm CIGARCoil that we present in this paper is a lossless one. A lossless compression scheme is used because failing to preserve the original content of the DNA sequencing data would be detrimental for end-users as they attempt to piece genomes together, byte-by-byte.

Some DNA compressors, such as the KungFQ compressor [10], have experimented with applying a lossy approach to the compression of the DNA sequencing file's metadata (*i.e.*, id field and quality scores).

The lower-bound for lossless compression is known as the *information theoretic lower-bound for compression*, which comes from Shannon's source coding theorem [20]. This bound is based on the entropy of the data being compressed, which is denoted by $H(x)$, which is more formally:

$$H(x) = -\sum_{i}^{a} P_i \log_2 P_i$$

where a is the number of different symbols in the alphabet and P_i is the probability of a symbol occurring. If there are N symbols being compressed from an alphabet of size a, then the compressed file must have at least $N \times H(x)$ bits.

1.1.3 MPEG-G: A Proposed Standard for DNA Read Compression

Understanding this growing challenge of the sheer size of genetic data, and navigating the task of determining what functionality of DNA compression end users are after is a difficult task. Fortunately the Motion Picture Entertainment Group surveyed various end users and compiled features into a hypothetical standard of genetic compression, which they called MPEG-G [1]. This standard contains the following features which will be discussed in this paper:

- *Indexing to access compressed data* allows for random access to a read within the compressed image.
- *Compressing a streamed file* allows for file to be incrementally compressed as data flows into the compressor
- *Compressed file concatenation* allow for multiple compressed files to be concatenated into a new compressed file
- *Incremental update of compressed file* allow for modification of a single read of the compressed file without uncompressing the compressed file

1.2 Motivation

DNA sequencing machines produce a massive quantity of data in the order of several gigabytes of per file. The sheer size of these files makes performing research with the data from these files prohibitively cumbersome for commodity hardware to store it in main memory. We seek to reduce the quantity of resources required to store these files in memory by providing a new compressor specialized for DNA sequencing data that permits end-users to randomly access individual reads from the file, without needing to decompress the file. By providing end-users with this compression format, we intend to not only provide end-users with a new format for storing their data, but also a new way of accessing the content of such files that does not require decompression and recompression of the file.

1.3 Preliminaries

The following terms, definitions, and algorithms are used and referenced by the remainder of this thesis.

1.3.1 Edit Distance

Edit distance is a metric that measures the number of changes that need to be made to transform one string into another. A common area where edit distance is seen is in predictive text features for cell phones where similar words are suggested for the user to enter. Although there are many algorithms that exist for computing edit distance, this paper focuses on the Wagner-Fischer algorithm [25] due to its robustness and flexibility.

Wagner-Fischer Edit Distance The Wagner-Fischer algorithm is used to calculate the number of operations needed to convert one string to another. The Wagner-Fischer algorithm finds the minimum number of a combination of four operations: insertion, substitution, deletion, and match to convert one string s to t. For example given a string s of AAGGTCCC and a string t of GAAAACCCC. The edit distance is found to be 4 in Table 1.1, by deleting G, matching the first two As, Inserting GGT, and matching the final three Cs.

This algorithm constructs a two-dimensional matrix, where each cell is the number of operations needed to transform s to that position in t, and the minimum edit distance is found in the bottom-right cell. Additionally, this algorithm determines the operations used to find the edit distance, which can be used in the encoding of t relative to s.

Table 1.1: Wagner-Fischer Matrix Example

	ϵ	**A**	**A**	**G**	**G**	**T**	**C**	**C**	**C**
ϵ	0	1	2	3	4	5	6	7	8
G	1	1	2	2	3	4	5	6	7
A	2	1	1	2	3	4	5	6	7
A	3	2	1	2	3	4	5	6	7
A	4	3	2	2	3	4	5	6	7
A	5	4	3	3	3	4	5	6	7
C	6	5	4	4	4	4	4	5	6
C	7	6	5	5	5	5	4	4	5
C	8	7	6	6	6	6	5	4	4
C	9	8	7	7	7	7	6	5	4

1.3.2 Graphs and Trees

Let $G=(V, E, W)$ be a graph with vertex set V and edge set E. Additionally, let $|V|=n$ and $|E|=m$. The weight function W assigns a positive integer weight to each edge in E. $W(u, v) \in \mathbb{N}$

We will now introduce to you the concept of a similarity graph G. There is an one-to-one correspondence between a read and the node of the graph G. The graph G is a complete graph and the weight on the edges is the edit distance between the corresponding DNA reads as determined by the Wagner-Fischer algorithm.

For example given the set of reads with edge weights calculated using Wagner-Fischer edit distance as seen in table 1.1:

- **r1:** AAAAAAAA
- **r2:** AAAACCCC
- **r3:** AAAATTTT
- **r4:** GGAACCCT
- **r5:** AAGGTCCC

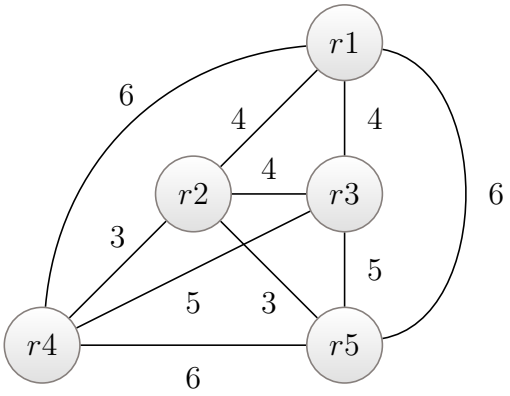

Figure 1.3: Similarity Graph

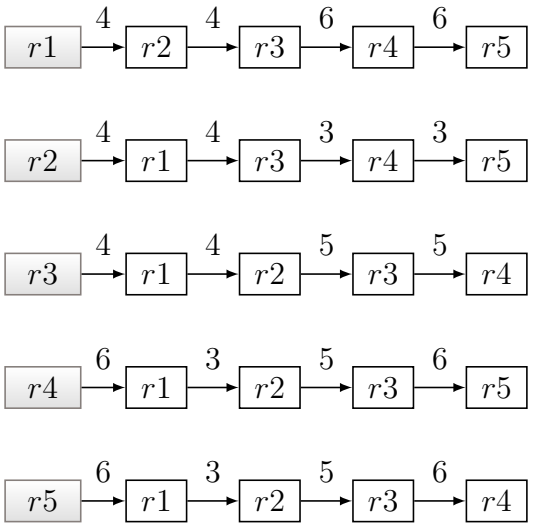

Figure 1.4: Adjacency List Representation of Figure 1.3

A similarity graph can be constructed from these five reads as seen in Figure 1.3.

There are many data structures that can be used for representing graphs (*e.g.* adjacency list and adjacency matrix). The adjacency list representation is used in this thesis as it supports constant time insertion of elements and does not waste memory resources when dealing with sparse graphs. An example of an adjacency list representation of a graph can be seen in Figure 1.4.

Minimum Spanning Tree (MST) A minimum spanning tree is the set of nodes and edges of a graph that form a tree that minimizes the combined weights of all of the edges. Similarly, a maximum spanning tree is the set of nodes and edges of a graph that form a tree that maximize the combined weights of all of the edges. The Coil, ReCoil, and CIGARCoil algorithms all use such spanning trees to find the most profitable edges to use for encoding. There are many algorithms that can be used to find a MST from a graph (e.g. Kruskal's algorithm [12] and Prim's algorithm [17], which have been parallelized by Quinn and Narsingh [18] and Grama *et al.* [9], respectively). We will using Prim's algorithm which adds one vertex at at time keeping the cost of the tree constructed at any

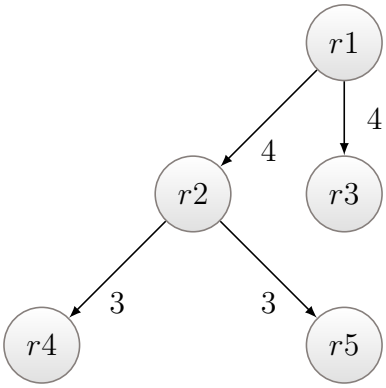

Figure 1.5: Minimum Spanning Tree

given time at a minimum. We will later see that our algorithm will take the similarity graph and construct a MST for it. The combined weights on the edges of the MST together with the size of a single read will determine the size of the resulting compressed structure. This is the crux of our overall approach and the details are presented in Chapter 2. A minimum spanning tree of the similarity graph seen in Figure 1.3 can be seen in Figure 1.5.

Prim's Algorithm Prim's algorithm is an for finding the minimum spanning tree for a graph, which was first presented in [17]. This paper utilizes an implementation of Prim's algorithm that has a time complexity of $O(E \log(v))$, where E is the number of edges and v is the number of vertices, due to its use a heap data structure.

Degree of a Node The degree of a node is the number of edges that it has connected to it. For example, using the tree in 1.5, r2 has a degree of 3 because it has edges between itself and r1, r4, and r5.

Parent Array A tree can be represented as a parent array. Each index of this array describes the node that is the parent of the node at the current index. For example if index i of the array has a value of x, then this means that node i has a parent and that node is x. In the case of the root of the tree, the node can list itself as its own parent. For example index k of the parent array could have value k to indicate that k is the root of the parent array.

1.3.3 Machine Learning

Machine learning is the usage of an algorithm that attempts to find patterns in input data. CIGARCoil utilizes two different machine learning algorithms, K-Means clustering, and Q-Learning. K-Means clustering is used during the compression of a file to improve the compression speed of CIGARCoil. Q-Learning is used to assist with CIGARCoil's random access feature by pre-fetching records that the user is likely to query in the future.

K-Means Clustering First proposed in [14], K-Means clustering is an unsupervised learning technique that organizes data into clusters based on their proximity to a set of centroids. After each iteration of this clustering technique, the centroids themselves are

updated to become an average of their current cluster. Once the centroids have been updated, the clusters are then recomputed based on the new centroids. This process continues until either no data moves from one cluster to another or a specified max number of iterations has been performed. See algorithm 1.

Algorithm 1 K-Means Clustering

```
LET K be the number of clusters
LET i be the maximum number of iterations
LET N be the number of items being clustered
LET C be the set of centroids
for all k : K do
  c = randomly initialized item
  add c to C
end for
for 0 : i do
  for all n : N do
    LET b the best similarity be 0
    LET e the best centroid be 0
    for all k : K do
      LET d be the distance between C[k] and n
      if d >b then
        SET b = d
        SET e = k
      end if
    end for
    Assign n to C[e]
  end for
  for k : K do
    Recompute k as average of assigned items
  end for
end for
return Centroid item assignments
```

The time complexity for clustering is as follows:

- Let n be the number of elements to be clustered
- Let k be the number of centroids being used
- Let i be the maximum number of clustering iterations
- Let d be the cost of finding the distance between an element and a centroid

$$O(n \times k \times i \times d) \tag{1.1}$$

Q-Learning Q-learning is a reinforcement learning strategy first proposed in [26]. Q-learning works by training a learning agent to take an action a that maximizes a reward received from a reward function based on the current state that the action is in.

Algorithm 2 Q-Learning

```
LET α be the learning rate
LET ϵ be the probability of a random action
LET γ be the discount rate
LET S be the number of states
LET A be the number of actions
LET r be the reward
Initialize Q as a matrix of dimension S x A
for all episode do
  LET r = randomValue
  LET s be the current state
  if r <ϵ then
    LET c be a random action
  else
    CHOOSE c based on MAX(Q[s])
  end if
  LET prevQ = Q[s][c]
  LET prevS = s
  Take action c changing state s
  if s is good then
    reward = 1
  else
    reward = -1
  end if
  LET p = MAX(Q[s])
  LET u = (1 - α) × prevQ + α × (reward + γ × p)
  Q[prevS][c] = Q[prevS][c] + u
end for
```

1.3.4 K-mer

A K-mer is a sub-string of a DNA read such that it contains K characters. For example given the read ACTGACGGAC, its set of K-mers of length four is {ACTG, CTGA, TGAC, GACG, ACGG, CGGA, GGAC}.

1.3.5 Other DNA Sequencing Data Compressors

A wide variety of different DNA-sequence-specialized compressors has been proposed within the past couple of decades as DNA sequencing has become easier and innovations like Next-Generation Sequencing have made the process faster. Despite the effort that scientists have spent on producing these specialized compression tools, the most widely-used compressor is gzip [3]. The gzip compressor has been shown to perform worse in terms of compression ratio, compression speed, and decompression speed in comparison to not only specialized DNA compressors like LFQC [16], ReCoil [28], and DSRC [4], but also other general-purpose compressors like bzip and 7-zip as observed in a 2013 survey paper comparing different compressors [5].

gzip The gzip compressor is a free widely-used lossless general-purpose compressor that comes with most flavors of Linux and can be installed on other operating systems as well. Gzip utilizes Huffman encoding as well as LZ77, a dictionary encoder, to compress a file. Gzip does not support random access of the compressed file.

bzip2 This compressor, bzip2, is a general purpose compression algorithm that compresses files using Burrows-Wheeler transforms as well as Huffman encoding to compress files. Bzip does not provide any special random access to the compressed file, such as a square bracket operator, and it has been shown to be inferior to specialized DNA sequence compressors such as LFQC and DSRC2 in previous work [16].

LFQC LFQC is an algorithm for the compression of DNA sequences that was first proposed in 2014 in [16]. This algorithm uses lossless and non-reference based compression on FASTQ files. This compression scheme compresses the FASTQ file's identification, sequence, and quality score information separately, each using a different algorithm that performs run-length encoding. Although this algorithm achieves impressive results in terms of compression ratio and speed, it does not provide for random access of the compressed file's contents.

DSRC2 DSRC2 is an algorithm for the compression of DNA sequences that was first proposed in 2010 in [4]. Similarly to LFQC, this algorithm also treats IDs, sequences, and quality scores as separate streams during compression, making use of different forms of run-length encoding. This algorithm compresses the file in a set of blocks, which contain information at the head of each block that can be used to decompress the current block. This allows an individual block of the compressed file to be encoded; however, it is not as granular as the decompression of a single read in the file.

The approximate similarity is calculated using a heuristic where the number of k-mers in common is counted. This algorithm relies on the general-purpose compressor for finding the optimal way to reduce the encoded differences and only supports FASTA files.

ReCoil ReCoil [28] sought to improve upon its predecessor, the Coil algorithm [27] by utilizing external memory algorithms. External memory algorithms are algorithms that run while storing the bulk of the content of the file on disk rather than in main memory. This external memory algorithm was used because Coil required a prohibitively large amount of memory for the data structures that it used. Although ReCoil succeeded in circumventing the memory issues that the Coil algorithm encountered, the ReCoil algorithm became incredibly slow with the usage of its external memory algorithms as external memory algorithms. Similarly to Coil, ReCoil does not support the compression of FASTQ files.

The ReCoil algorithm works in the following manner:

1. Construct a Similarity Graph where the edge weights represent the similarity between each node, which each represents a read. This similarity is the number of shared sub-strings of a fixed length (referred to as *k-mers*).

2. Extract a MST that maximizes the similarity between each read.

3. Encode each child node relative to its parent node using a maximal exact match strategy. Since the MST has been constructed, it is more likely that a large maximal exact match exists between parent and child, increasing redundancy and leading to improved compression.

4. Apply a general-purpose compressor to the encoded tree such as gzip.

1.3.6 CIGAR String

CIGAR is an acronym for Concise Idiosyncratic Gapped Alignment Report. The general idea of a CIGAR string was first presented by Fritz *et al.* [7]. CIGAR strings describe the operations required to convert one string into another, by encoding the differences between the strings, rather than the strings themselves. CIGAR strings are a primary component of the SAM family of DNA sequencing data compressors [13].

For example, given the strings a) ACTGGGGG and b) GCAGGGGG, the string b) can be expressed relative to the string a) using the CIGAR String SGCAM5. This string is interpreted as follows: the first letter S stands for substituting and it replaces substring ACT (in string a)) with string GCA (that is in b)), next is the letter M which is a match, here we have a 5 letter substring (GGGGG) that matches both the strings.

Although the SAM format utilizes seven different operations, this paper uses CIGAR strings with four different operations, match, insertion, substitution, and deletion, which is more similar to the reference-based approach of [7] because unlike SAM, CIGARCoil involves the compression of two reads with similar if not the same lengths, whereas SAM compares a reference string that has a number of bases in the order of millions to each read that has a number of bases in the order of hundreds.

1.4 Contributions

In this thesis we make the following contributions:

1. Construction of the similarity graph where the edge weights are the size of the smallest CIGAR string between any two reads, computed using the Wagner-Fischer edit distance algorithm

2. Integration of zpaq, an open source compressor, for the compression of read metadata, which previous compressors that used a similarity graph approach did not support

3. Application of an unsupervised machine learning technique, K-Means clustering to improve compression speeds with small reductions in compression ratio

4. Providing mechanism for random access of the sequencing data of the compressed file in the form of a [] square bracket operator

5. Extension of square bracket operator with a predictive cache utility, which uses machine learning (Q-Learning) to adapt to shifting data access patterns.

6. Providing an open source compressor to the public on GitHub free to use for both public and private entities

Chapter 2

CIGARCoil Algorithm

This chapter describes the underlying algorithms of the CIGARCoil compressor. CIGARCoil along with its predecessor compressors ReCoil [28] and Coil [27] are unique in that treat each read of the input data as a node in a graph. CIGARCoil uses the concept of a CIGAR string to represent the edge weights inbetween each pairing of nodes in the graph [7]. Additionally, CIGARCoil uses a general-purpose compressor zpaq to handle each read's meta-data. At the end of this chapter a set of special CIGARCoil features (*i.e.*, file concatenation and incremental update) is described that provide additional utility for end-users.

2.1 Inspiration

CIGARCoil at its core is the synthesis and extension of ideas found in three different approaches for the compression of DNA sequencing data. These ideas are as follows:

CIGAR String The first of these three approaches is the reference-based compression idea of Fritz *et al.* [7], which has become a core component in the SAM family of DNA sequence compressors [13]. A key difference between our approach and Fritz's approach is that Fritz utilizes one large reference string that is external to the data for encoding and our approach uses the reads that are already present in the data set as reference strings.

LFQC Second, in addition to the reference-based idea of Fritz *et al.* this approach uses a common approach found in many other DNA sequence compressors, the splitting of the input file into different streams and processing them differently as to take advantage of type of data found in each stream. One such approach is the LFQC paper, which separates the DNA base-pair data from the meta-data, performs different transformations on the data, then applies the zpaq compressor, which they found to be the most effective general-purpose compressor, to the data [16].

ReCoil Third, this approach re-imagines the unique similarity graph approach taken in compressing base-pair data by the ReCoil [28] and Coil [27] compressors. Changes include the support of meta-data, encoding edges using CIGAR size rather than the number of shared sub-strings, and leveraging the nature of the compressed file's tree structure to add support for a few operations to be performed on the compressed file: random access, file concatenation, and update of the compressed file.

2.2 Encoding

This section describes the mechanisms that CIGARCoil uses in its encoding and compression of DNA sequencing data. An overview of this process can be seen in Figure 2.1.

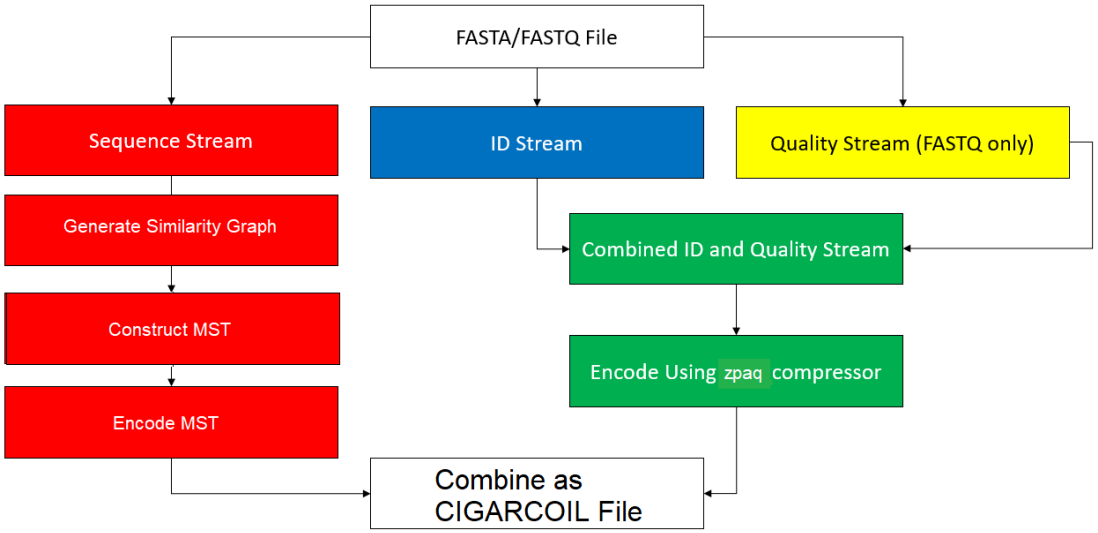

Figure 2.1: CIGARCoil Encoding Work-flow

Our algorithm starts with a set of n reads $r_0, r_1, ..., r_{n-1}$. For each read $r_i, 0 \leq i \leq n-1$, we create a node n_i in a directed graph G (*directed similarity graph*. There is a directed arc from n_i to n_j to indicate the changes to make in read r_i to make it equivalent to r_j. The weight w on the arc (n_i, n_j) denoted $w(n_i, n_j)$ is the Wagner-Fischer edit distance. We will have another arc from n_j to n_i and we note that $w(n_i, n_j) \neq w(n_j, n_i)$. This is true if we consider all the operations that are part of the editing process (insert, substitute, delete, and match).

To explain CIGARCoil's encoding the step following set of five reads is used as part of a short example throughout this section:

1. R_0: AAAAAAAAAAAAAAAA
2. R_1: AAAAAAAACCCCCCCC
3. R_2: CCCCCCCCACTGACNN
4. R_3: ACTGACTGACTGACTG
5. R_4: CCCCCCCCACTGNNCA

Each of our reads in the example data set of reads will now be represented by a node in a similarity graph. An image of such a graph is shown in Figure 2.2.

A directed similarity graph G contains $n \times (n-1)$ arcs. Once the similarity graph is constructed we will find a minimum spanning tree of this directed graph. The minimum spanning tree of the directed graph has the property that there exists a node such that there is a directed path from that node to all the other in the spanning tree, such a tree is referred to as an *arborescence* [8]. Finding the MST of an arborescence is known as the *optimum branching problem*. There are a few approaches for this optimum branching

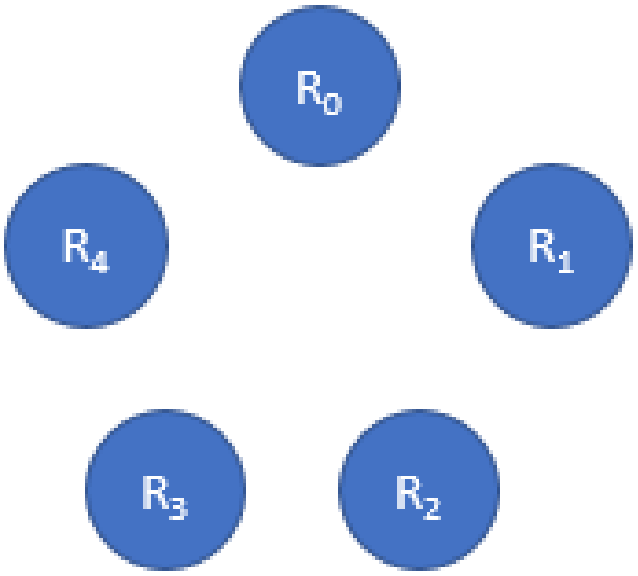

Figure 2.2: An edge-less similarity graph for reads R_0, R_1, R_2, R_3, and R_4

problem, such as Edmond's algorithm, which runs in $O(EV)$ where E is the number of edges and V is the number of vertices in the arborescence [6], and Tarjan's algorithm, which runs in $O(E \log V)$ for sparse graphs and $O(V^2)$ for dense graphs [23].

Undirected Similarity Graph Restricting the operations to matches and substitutions will assure that the Wagner-Fischer edit distance between two reads r_i and r_j will be symmetrical that is, in the directed similarity graph $w(n_i, n_j) = w(n_j, n_i)$. With this assumption, we can now treat the directed similarity graph as a undirected one, by replacing arcs on both directions by a single edge. The weight on the edge will be our edit distance (or CIGAR size).

Adding Edges Edges are added to the similarity graph using the heuristic as described in the Node Compartmentalization Heuristic Section. Note that by definition a similarity graph is a complete graph. We use a heuristic we have developed to reduce the number of edges added. Continuing with our example set of five reads, we will now create an empty hashbucket index data structure that will be used later to query a reduced number of edges for each node to add an edge to. Initially this index structure looks like Figure 2.3; however, we will next populate it based on the partitions of our set of reads.

In order to determine which set of buckets to emplace a read's ID into within the hashbucket index structure, a set of partition values is computed for each read. The set of partition values for a read are the number of occurrences of each base A,C,T, and G within each partition of size Δ from the original read. An example of these partitions being computed for example read R_4 is shown in 2.4.

Now that the partition values for each bucket have been computed for read R_4, R_4 can now be added to the appropriate indices of the hashbucket index data structure as seen in Figure 2.5.

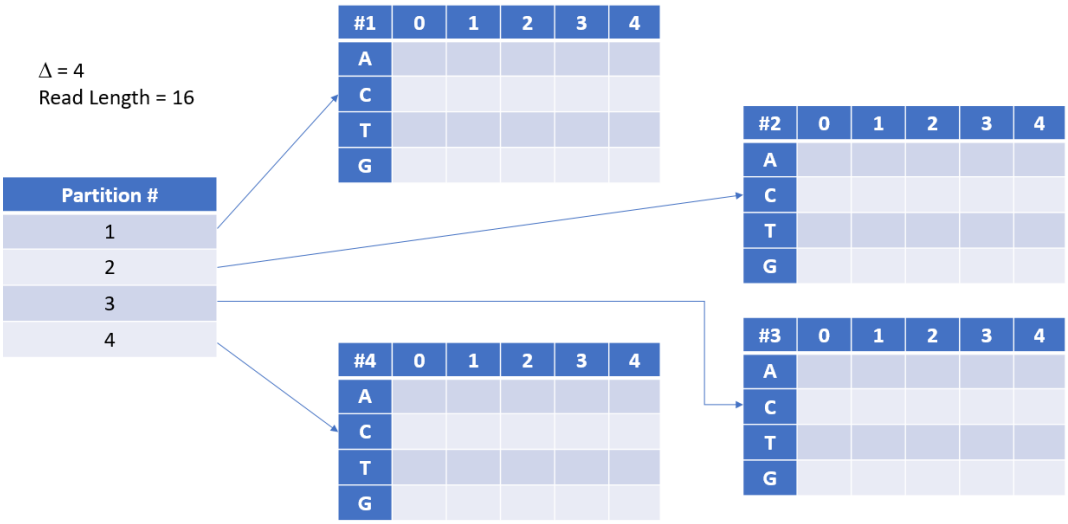

Figure 2.3: An initially empty hash bucket index for reads R_0, R_1, R_2, R_3, and R_4

CCCC

# Occurrences Partition 0	
A	0
C	4
T	0
G	0

CCCC

# Occurrences Partition 1	
A	0
C	4
T	0
G	0

ACTG

# Occurrences Partition 2	
A	1
C	1
T	1
G	1

NNCA

# Occurrences Partition 3	
A	1
C	1
T	0
G	0

Figure 2.4: Computation of Partition Values for read R_4 using Δ of 4

Partition values are computed for all reads in the data set in the same manner that they were computed for R_4. After emplacing all of the reads from the example data set in their appropriate buckets within the hashbucket index, the hashbucket index will be look as it does in Figure 2.6.

Now that we have populated the hashbucket structure with values corresponding to each read, we can now apply the node compartmentalization heuristic for each read to obtain a set of candidate reads to add edges to. Figure 2.7 shows the set of candidate reads for R_4 being found by performing intersections of the buckets. At the end of this process only R_2 remains as a candidate for R_4.

Now that R_2 has been identified as a candidate for adding an edge to from R_4, our custom implementation of Wagner-Fischer can be employed to determine the CIGAR size of read R_4 relative to read R_2, the edge weight. The Wagner-Fischer matrix for this example is shown in Figure 2.8.

By performing Wagner-Fischer to find R_4 relative to R_2, we found that encoding these differences requires a CIGAR string with a CIGAR size of 2. An edge with this weight can now be added to the similarity graph as seen in Figure 2.9.

Performing this process on all reads from the example set R_0 through R_5 results in the similarity graph as seen in Figure 2.10.

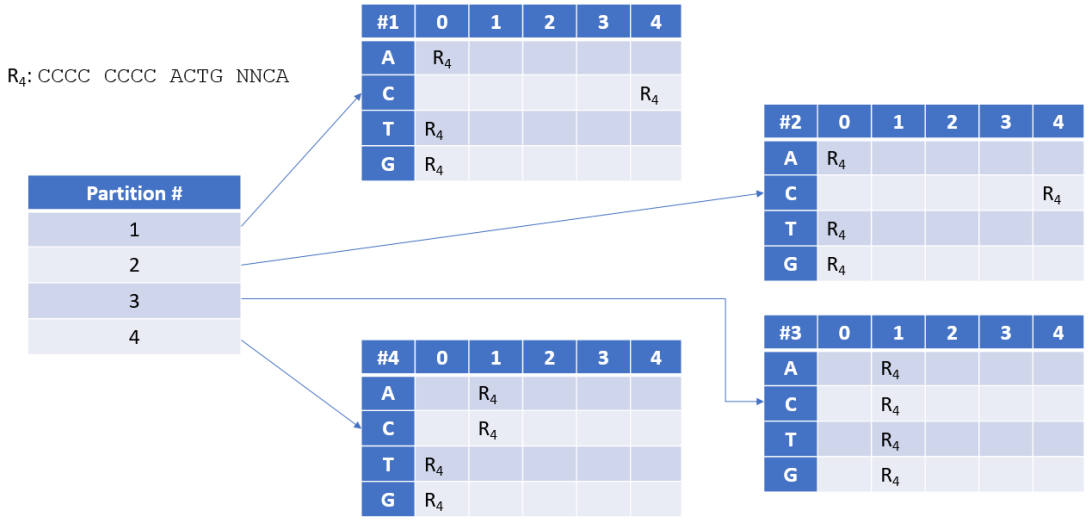

Figure 2.5: Hashbucket Index with only read R_4 added to it

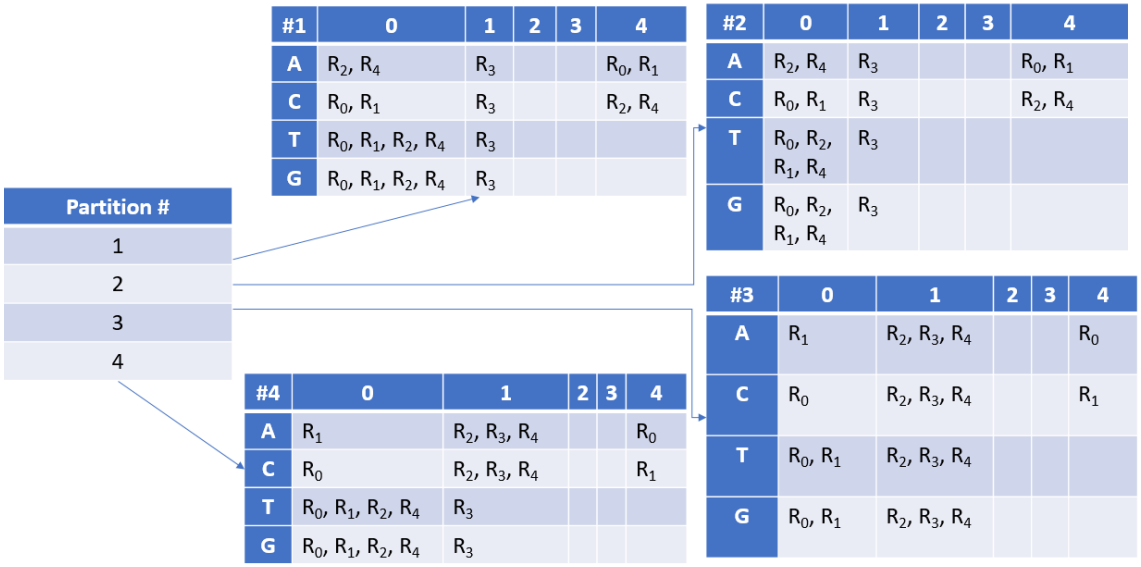

Figure 2.6: Hashbucket index populated with reads R_0, R_1, R_2, R_3, and R_4

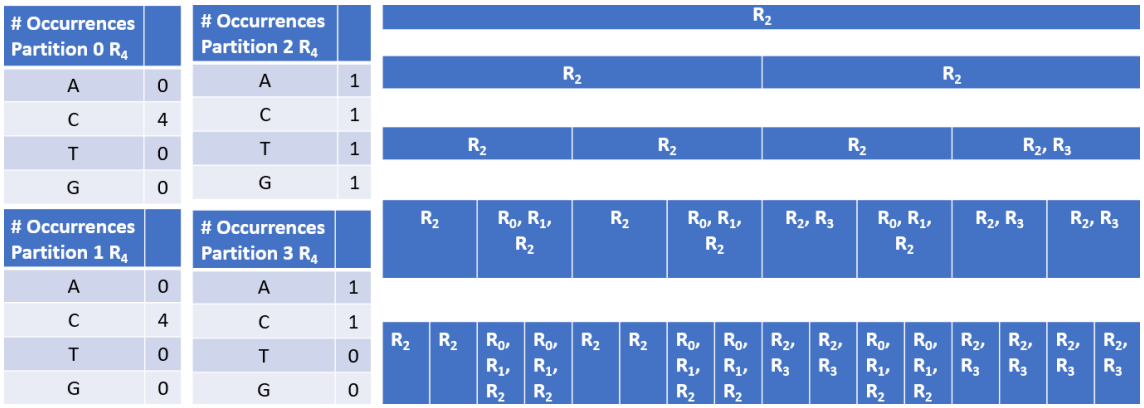

Figure 2.7: Example of node compartmentalization heuristic for read R_4

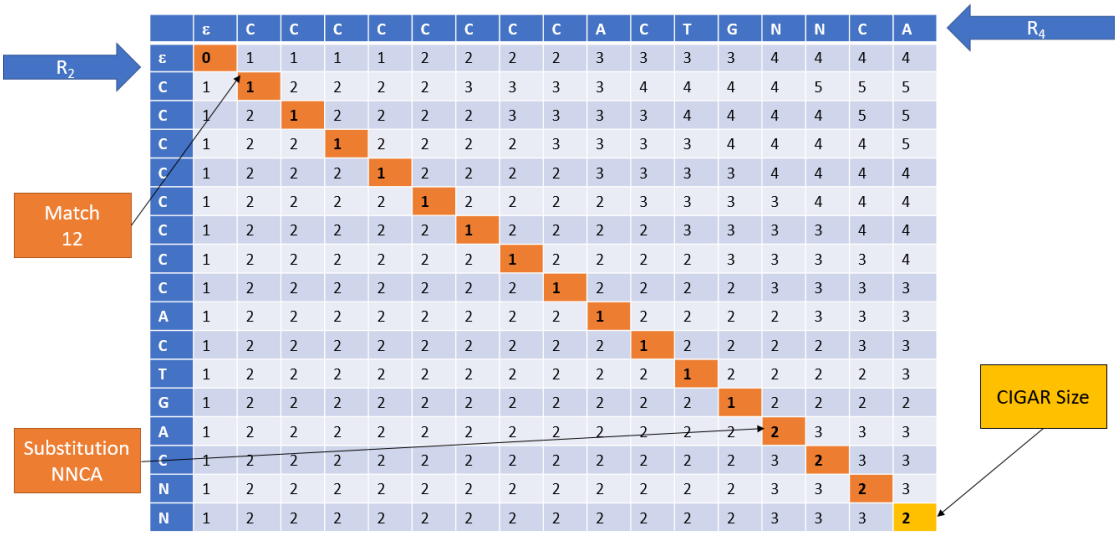

Figure 2.8: CIGARCoil Wagner-Fischer Matrix for read R_4 relative to read R_2

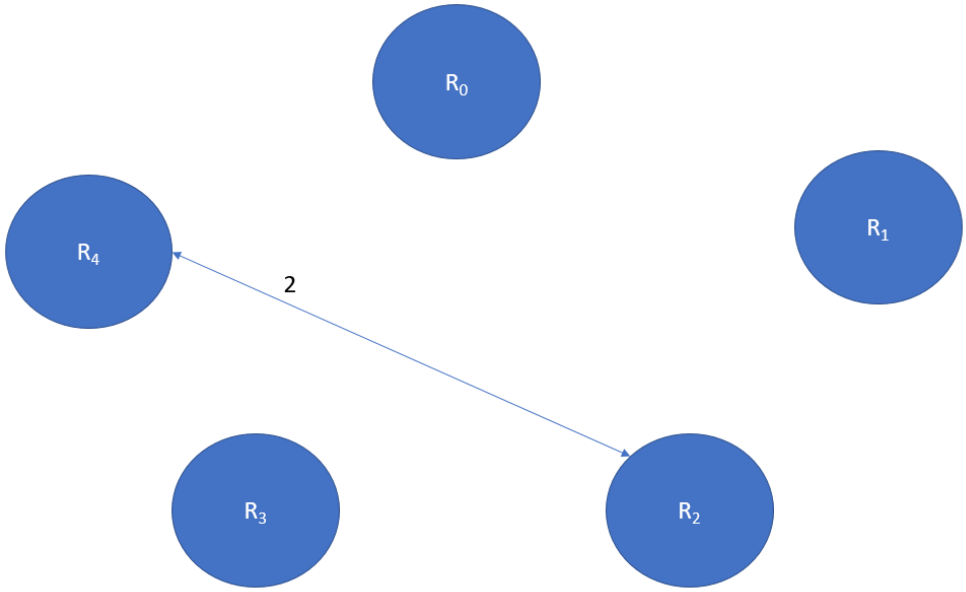

Figure 2.9: Similarity graph with just one edge drawn between R_4 and R_2

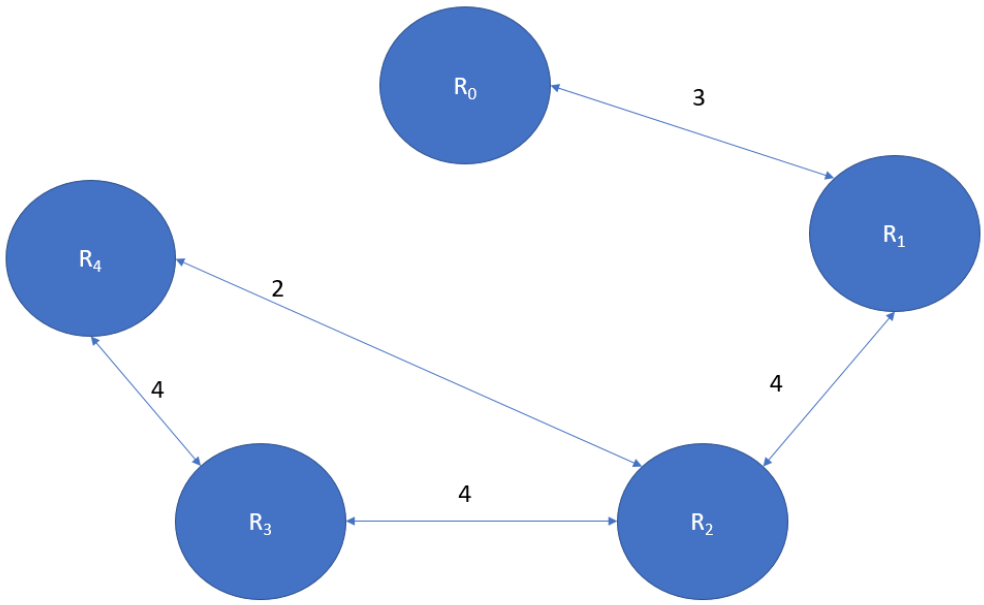

Figure 2.10: Similarity graph for R_0, R_1, R_2, R_3, and R_4 with all edges added

Compute Minimum Spanning Tree After the edges have been added to the tree, then a minimum spanning tree can be computed by applying Prim's algorithm to the similarity graph.

Now that we have finished adding edges to the similarity graph, a MST can be computed for the similarity graph, yielding the tree that requires the fewest number of bytes to encode the differences between the reads. We arbitrarily select the read R_2 as the root of the tree constructed by Prim's algorithm. The only edge that needed to be removed to compute this minimum spanning tree is the edge from read R_3 to read R_4 with weight 4.

Encode Minimum Spanning Tree The parent array for the minimum spanning tree is written to the encoded output file. Then the root's ID, sequence, and quality score (if the original file was FASTQ) are written to the output file. Then for each node in the tree, its id is written to the output file, followed by the cigar string that encodes its sequence relative to its parent, followed by its quality score if the original file was FASTQ.

Encoding the Quality Scores The encoding of quality scores is more challenging than the sequence due to the significantly larger alphabet for quality score characters than sequencing data. Quality scores can potentially range from 0 to 255, and different sequencing machines generate these quality scores differently. The approach taken by other FASTQ compressors such as DSRC [4] and LFQC [16] is to use separate methods for handling sequence and quality data. DSRC uses Huffman encoding on blocks of quality scores, and LFQC uses the zpaq compressor, an open source and open API general compressor, on quality scores. Since zpaq has been shown to be effective at compressing quality scores [2], and zpaq supports various operations that are of interest to future development such as file concatenation and streaming compression, we have also elected to employ the zpaq compressor for compressing quality scores.

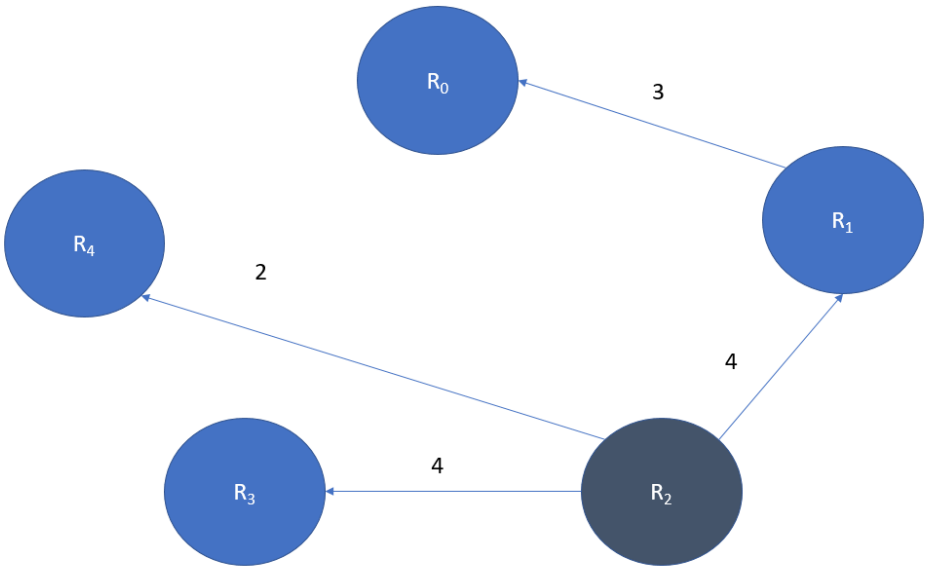

Figure 2.11: MST computed for similarity graph in Figure 2.10

Parent Array:	Index for R_0	Index for R_1	Index for R_2	Index for R_3	Index for R_4
	1	2	2	2	2
Encoded MST:	R_0 relative to R_1 CIGAR R_1 relative to R_2 CIGAR R_2 Explicitly R_3 relative to R_2 CIGAR R_4 relative to R_2 CIGAR				
Compressed Metadata:	R_0 compressed metadata R_1 compressed metadata R_2 compressed metadata R_3 compressed metadata R_4 compressed metadata				

Figure 2.12: Resulting file structure for R_0, R_1, R_2, R_3, and R_4

Writing the CIGARCoil file Now that the MST has been rerooted, The final step is the encoding of the tree. First, the parent array of the MST is written to the file. Note that in our example, R_2 has its parent listed as 2, indicating that it is the root of the tree. Second, the Minimum spanning tree is encoded by writing each read relative to its parent in the tree as a CIGAR string. The root of the tree is written explicitly. Third, the meta-data can be compressed and concatenated to the end of the file. Finally, a general-purpose compressor like bzip is applied to the file, further reducing its size. An example of this file is seen in Figure 2.12.

2.3 Time Complexity

The time complexity of encoding is the worst-case amount of time that will be required to compress a set of data. Let n be the number of reads. In summary the time complexity of encoding the data set is as follows:

$$(n^2 - n) \times O(Wagner - Fischer) + O(Prim's) + O(EncodeTree) \quad (2.1)$$

This means that for each of the n reads in the data set we will create an edge to every other read in the complete directed similarity graph. The cost of computing each edge is the cost of computing a Wagner-Fischer edit distance. Next, a MST is computed for the resulting Similarity Graph using Prim's algorithm. Finally, the tree is encoded using CIGAR strings. Now we will break down the above expression:

Wagner-Fischer Time Complexity For each edge of the complete graph constructed with n nodes, the Wagner-Fischer algorithm is performed to compute the edge weight. As discussed in the preliminaries section, the time complexity of Wagner-Fischer is O(i $\times$ j), where i and j are the lengths of the two nodes' reads, which are dependent on the DNA sequencing machine that can produce reads from a few dozen characters in length to a few hundred characters in length.

Minimum Spanning Tree Time Complexity Once the similarity graph is constructed. A minimum spanning tree of it can be computed using Prim's algorithm, which as discussed in the preliminaries section has a time complexity of $O(E \log(n))$, where E is the number of edges and n is the number of reads.

Encode Tree In order to encode an a node of tree, Wagner-Fischer edit distance must be computed between itself and its parent, which also yields the CIGAR operations required to encode the set of operations. This step is $O(n \times (i \times j))$, where n is the number of nodes, and i and j are the lengths of the parent and child reads.

2.4 Node Compartmentalization Heuristic

Computing the edge weights for all edges within the similarity graph requires a prohibitively large number of operations to be performed. For example, given a set of 100,000,000 reads each with length 100 bases, adding all edges to the undirected graph requires ((100,000,000^2 - 100,000,000) / 2) $\times$ (100 $\times$ 100) operations, which is about fifty quintillion operations that a machine must complete to compute such a set of edges. The following heuristics are used to reduce the number of reads that edges are being added between. This serves three purposes.

- Too many edges would be difficult to store in main memory for a large number of reads.
- The majority of the edges will be pruned immediately after the construction of the graph to make a MST, which will then be encoded using CIGAR strings.
- The Wagner-Fischer edit distance used for the edge weights is $O(i \times j)$ where i and j are the lengths of the two nodes' reads, which would be very costly to compute for every two nodes.

One of the greatest challenges of generating a similarity graph is determining which nodes to create edges between. Although an edge could be drawn between each and node, since there is a CIGAR string for any two strings, doing so would be unwise because this

would result in $(n^2 - n)/2$ edges. This prohibitively large number of edges significantly hinders the creation of the minimum spanning tree using Prim's algorithm ($O(Elog(v))$), although it does guarantee that the tree MST is constructed with the lowest cost edges from the complete graph.

In order to reduce the number of edges that are added to the similarity graph, the following heuristic is employed:

Each read is L bases in length. Each read can be partitioned into b hash buckets with a length of Δ characters.

The following probabilistic calculations are made under the assumption that each read's sequencing data consists of approximately 25 percent Adenine (A), 25 percent Cytosine (C), 25 percent T, 25 percent G, and 0 percent N; however, there will certainly be deviations from this specific to the source of the data.

$$\text{percentageOfReadsInBucket} = \text{choose}(\Delta, i) \times ((1/4)^i) \times (3/4)^{\Delta - i} \tag{2.2}$$

For each Δ characters in the sequence, there are four hash buckets, corresponding to the four characters A, C, T, and G. For read i, if there are j occurrences of A, k occurrences of C, m occurrences of T, and n occurrences of G, then the hash bucket for A will have the value i added at index j, the hash bucket for C will have the value i added at index k, and so are. This is continued for all Δs within the string, populating the $4 \times L$ / Δ hashbuckets. This provides a means for groups of reads with similar characteristics to any given read to be quickly accessed in constant time using a data structure such as a dictionary of vectors for each hash-bucket.

Hashbuckets Data Structure The following data structure as seen in Figure 2.13, a hashbucket index, is used to find all reads with a given number of occurrences of a particular character in a partition of a sequence of length Δ in constant time. This data structure is populated only once when the reads are passed into the file.

Hashbucket Read Insertion Example For example, given a string AAACTTG-GACTGACTG representing the sequencing data of the read i, and a Δ of 8, the first partition of the string is the substring AAACTTGG. This substring contains 3 A, 1 C, 2 T, and 2 G. The value i is emplaced at the end of the sets within the hashbuckets corresponding to 3 As in the first Δ characters, 1 C in the first Δ characters and so on.

Choosing Δ As is illustrated in figures 2.14 and 2.15, reads are most likely to be assigned to the partition corresponding to $\Delta/4$.

The graph in figure 2.16 plots the percentage of total reads in the worst case versus the size of Δ. Although increasing the size of Δ reduces the percentage of reads encountered in this worst-case, this worst-case quickly begins to converge around 12 percent. In our implementation of this, we have elected to use a Δ of 16 because it permits 22 percent of reads in the worst case and is small enough that a read of length 36, the smallest length read of one of our data sets as seen in Table 5.6 as well as the smallest read length supported by a DNA sequencing machine like the Illumina MiSeq sequencer as seen in 1.1 can have still have multiple hash bucket partitions to perform intersections with.

Complete Graph vs. Heuristic Graph A heuristic for adding edges is superior than using a complete graph (G) for constructing a MST(T) if its resulting graph (G') and its

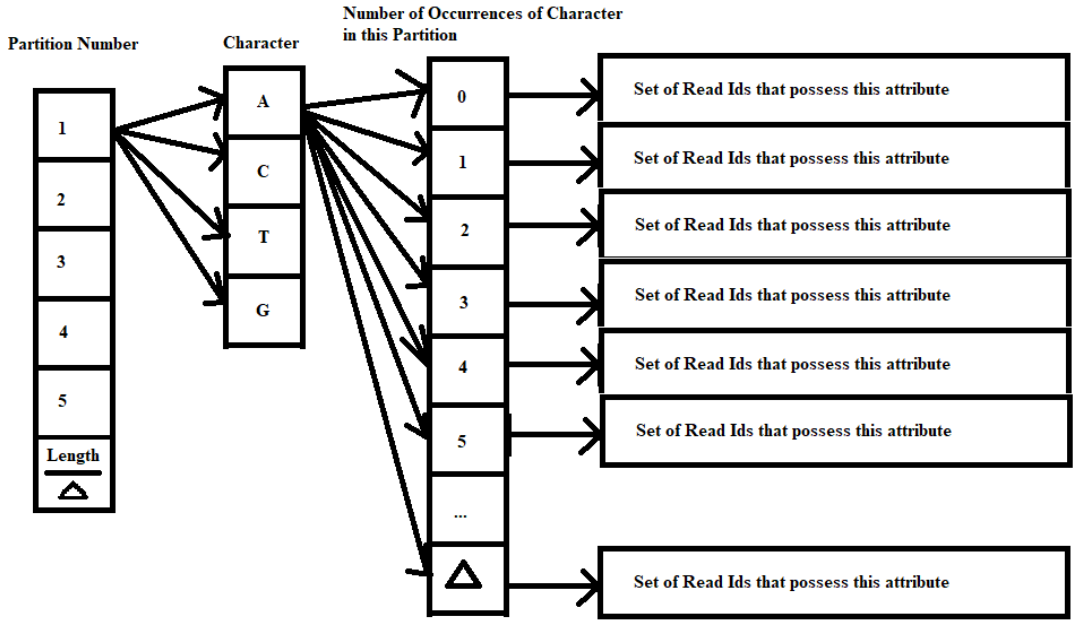

Figure 2.13: The Hash Buckets Index Data Structure that is used to find all reads with a particular attribute in constant time.

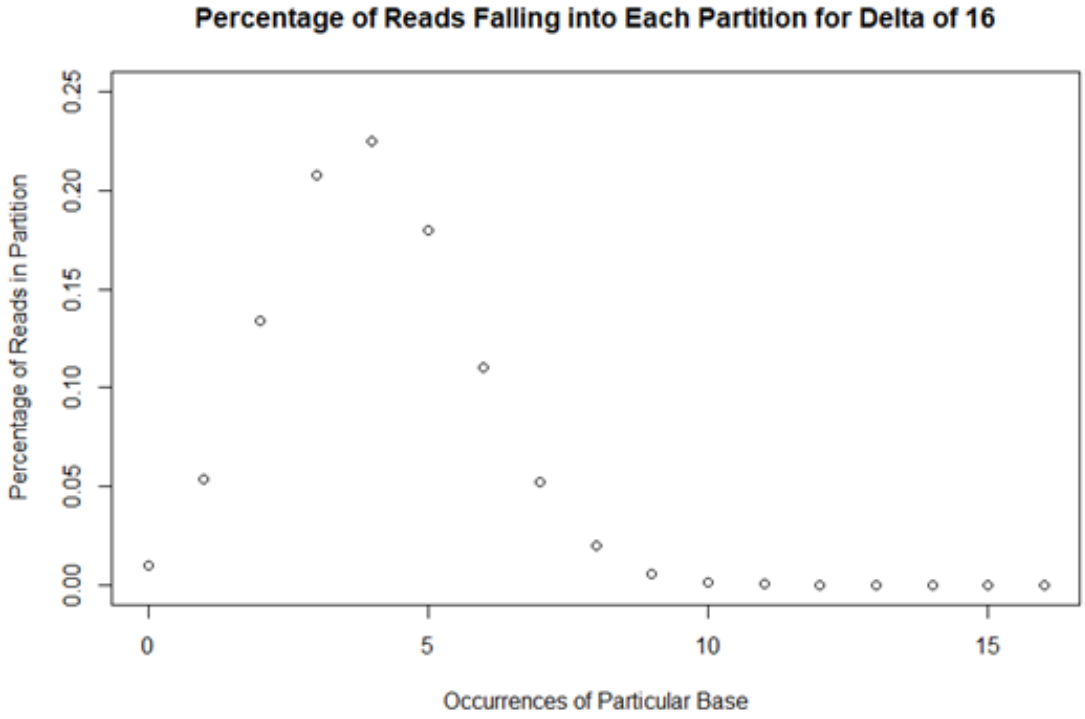

Figure 2.14: Assumed percentage of reads in each bucket for Δ of 16

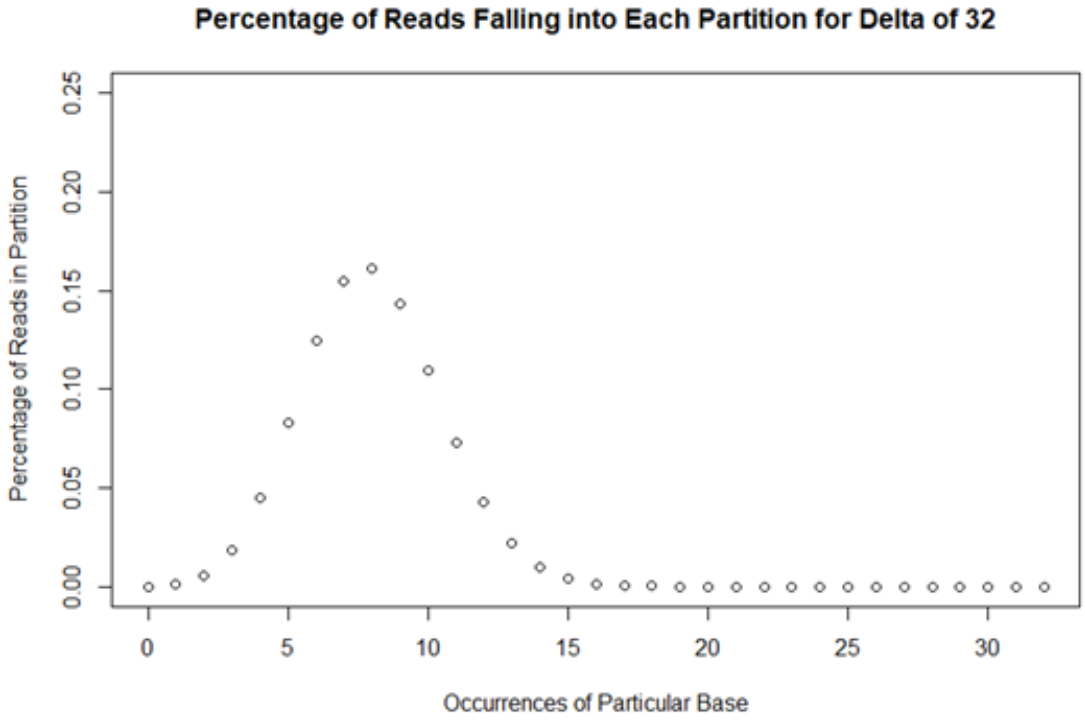

Figure 2.15: Assumed percentage of reads in each bucket for Δ of 32

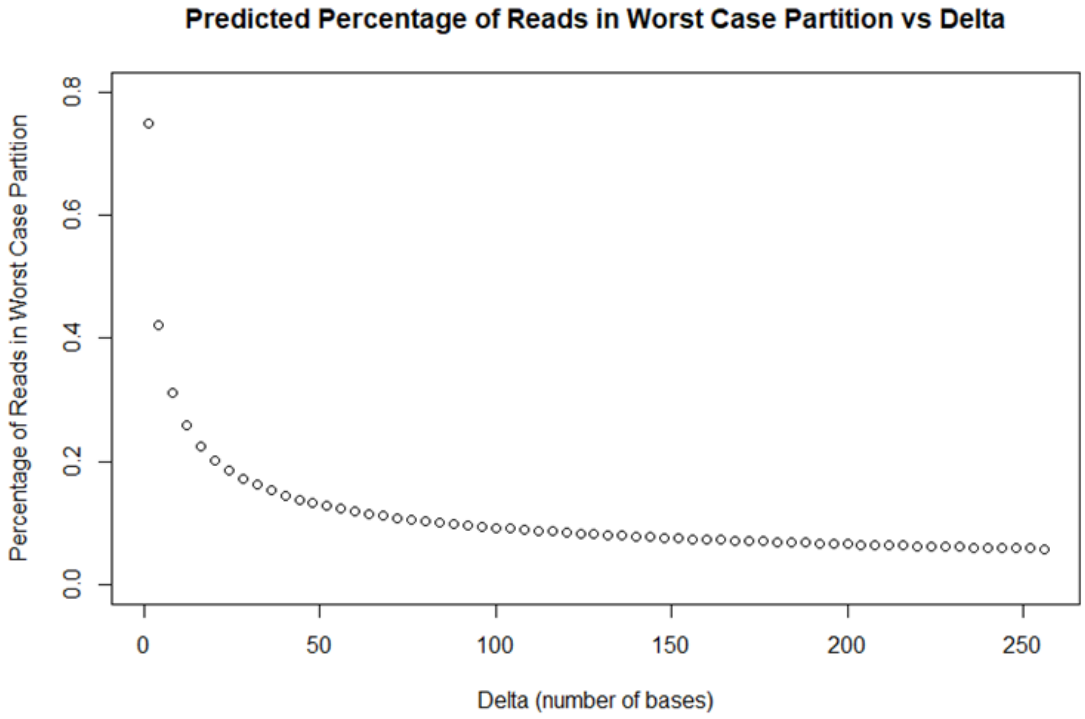

Figure 2.16: Percentage of reads in worst case as Δ increases

MST(T') satisfies the following condition: $\forall(u,v) \in E_G \wedge (u,v) \notin E_{G'}$ and $w(T) \geq w(T')$

Time Complexity With Heuristic The time complexity of encoding is the worst-case amount of time that will be required to compress a set of data. Let n be the number of reads, and let n_h be the number of reads identified by the node compartmentalization heuristic. In summary the time complexity of encoding the data set is as follows:

$$n \times O(heuristic) + O(AddEdges) + O(Prim's) + O(MinTreeHeight) + O(EncodeTree) \tag{2.3}$$

This means that for each of the n reads in the data set, we will first apply the hash bucket heuristic to it. Then for each read's subset of reads to create an edge to, the cost of computing the edge weight is applied and the edge is added. Next, a MST is computed for the resulting Similarity Graph using Prim's algorithm. Then, the height of the tree is minimized to assist with decoding and random access. Finally, the tree is encoded using CIGAR strings. Now we will break down the above expression:

Heuristic Time Complexity The node compartmentalization heuristic identifies a subset of reads n_h for adding edges to between the current read as adding edges between all reads is not feasible and ultimately unnecessary as a minimum spanning tree is immediately generated after the similarity graph is constructed. The time complexity of performing the heuristic is as follows:

$$O(n \times (((L/\Delta) \times 4) - 1)) = O(heuristic) \tag{2.4}$$

The n component corresponds to the cost of performing an intersection on two pre-sorted sets, and the $((L/\Delta) \times 4) - 1$ component corresponds to the number of non-leaf nodes of a binary tree for the hash bucket structure where intersections will be performed. At this point it seems as if this heuristic has resulted in a quadratic time algorithm due to $n \times O(heuristic)$; however, this assumes an exceptionally rare worst case where all reads in the file are the same. The graph shown in Figure 2.16 illustrates the probable number of reads to begin with in each partition, and after each intersection the number of reads remaining to be considered decreases significantly. For example, if there are 25 percent of reads in a partition A and 25 percent of reads in a partition B, then the expected percentage of reads in the intersection of both A and B is 6.25 percent. This percentage of remaining reads continues to become exponentially smaller as more intersections are performed. The number of reads left for consideration after performing all intersections is referred to as n_h.

Adding Edges Time Complexity After a subset of n_h reads has been identified for a given read to generate edges between, the Wagner-Fischer algorithm is performed to compute the edge weight. As discussed in the preliminaries section, the time complexity of Wagner-Fischer is $O(i \times j)$, where i and j are the lengths of the two nodes' reads.

$$O(n \times n_h \times O(i \times j)) = O(AddEdges) \tag{2.5}$$

Minimum Spanning Tree Time Complexity Once the similarity graph is constructed. A minimum spanning tree of it can be computed using Prim's algorithm, which

as discussed in the preliminaries section has a time complexity of $O(E\log(n))$, where E is the number of edges and n is the number of reads.

Root MST such that it has minimal height Once the minimum spanning tree has been constructed, the tree is then re-rooted to minimize the height of tree. Although this does not aid in compression, this step reduces the number of de-coding operations to be performed to randomly access a node later. This is discussed in Chapter 3 takes $O(n)$ time.

Encode Tree In order to encode an a node of tree, Wagner-Fischer edit distance must be computed between itself and its parent, which also yields the CIGAR operations required to encode the set of operations. This step is $O(n \times (i \times j))$, where n is the number of nodes, and i and j are the lengths of the parent and child reads.

2.5 Decoding and Decompression

Decoding and Decompressing a compressed CIGARCoil file is significantly less computationally intensive than compressing the file. A diagram of this work-flow can be seen in Figure 2.17. Algorithm 5 describes the recursive process that is used to decode the entirety of the compressed tree structure.

Algorithm 3 Decode-CIGAR

```
LET c be a cigar string
LET s be the parent string
LET d = ""
LET i = 0
for all Operation o in c do
  if o == MATCH then
    d += s.substr(i,o.length)
    i += o.length
  else if o == DELETION then
    i += o.length
  else if o == SUBSTITUTION then
    u = o.Values
    d += u
    i += u.length
  else if o == INSERTION then
    u = o.Values
    d += u
  end if
end for
return d
```

Algorithm 4 Decode-Child

```
LET c be a cigar string
LET P be the parent array
LET D be the previously decoded reads
if P[c] NOT IN D then
  CALL Decode-Child on P[c]
end if
CALL Decode-CIGAR with c and D[P[c]]
INSERT decoded c in D
```

Algorithm 5 Decode CIGARCoil Sequences

```
LET D be previously decoded reads
LET P = parent array stored at head of file
Insert Root of tree into D
for all CIGAR Strings in File do
  if current string != root then
    CALL Decode-Child on current child
  end if
end for
```

2.6 Special Features

The following two special features, file concatenation and incremental update are provided by the CIGARCoil format. These two features are requested in the MPEG-G DNA compression standard [1].

2.6.1 File Concatenation

Two compressed files can be easily concatenated. Given two CIGARCOIL compressed files, A and B, file B can be concatenated with file A by finding a node on file A's MST to connect the root of file B's MST. Once this node is found, the root of file B can be represented using a CIGAR string relative to the node in A's MST, and the rest of file B can be inserted into file A following the newly modified root of its own tree. The resulting file constructed in this manner likely does not represent a minimum spanning tree; however, concatenating files together in this manner is less computationally expensive as a new similarity graph and mininimum spanning tree is not constructed.

2.6.2 Incremental Update

Incrementally updating the compressed data is made easy by this format. An individual read in this compressed file can be updated by first using the same method as file indexing to arrive at the node of interest. Additionally the children nodes of the chosen read must be decompressed. Next, the chosen read can be modified. After the chosen read is modified, new CIGAR representations for its children reads will need to be assigned to them based upon their modified parent. Then the chosen read will receive its own new CIGAR representation relative to its parent. The resulting file after this update is likely not a minimum spanning tree anymore; however, updating an individual read in this

manner is less computationally expensive than decoding the file, modifying the record, then encoding the modified file.

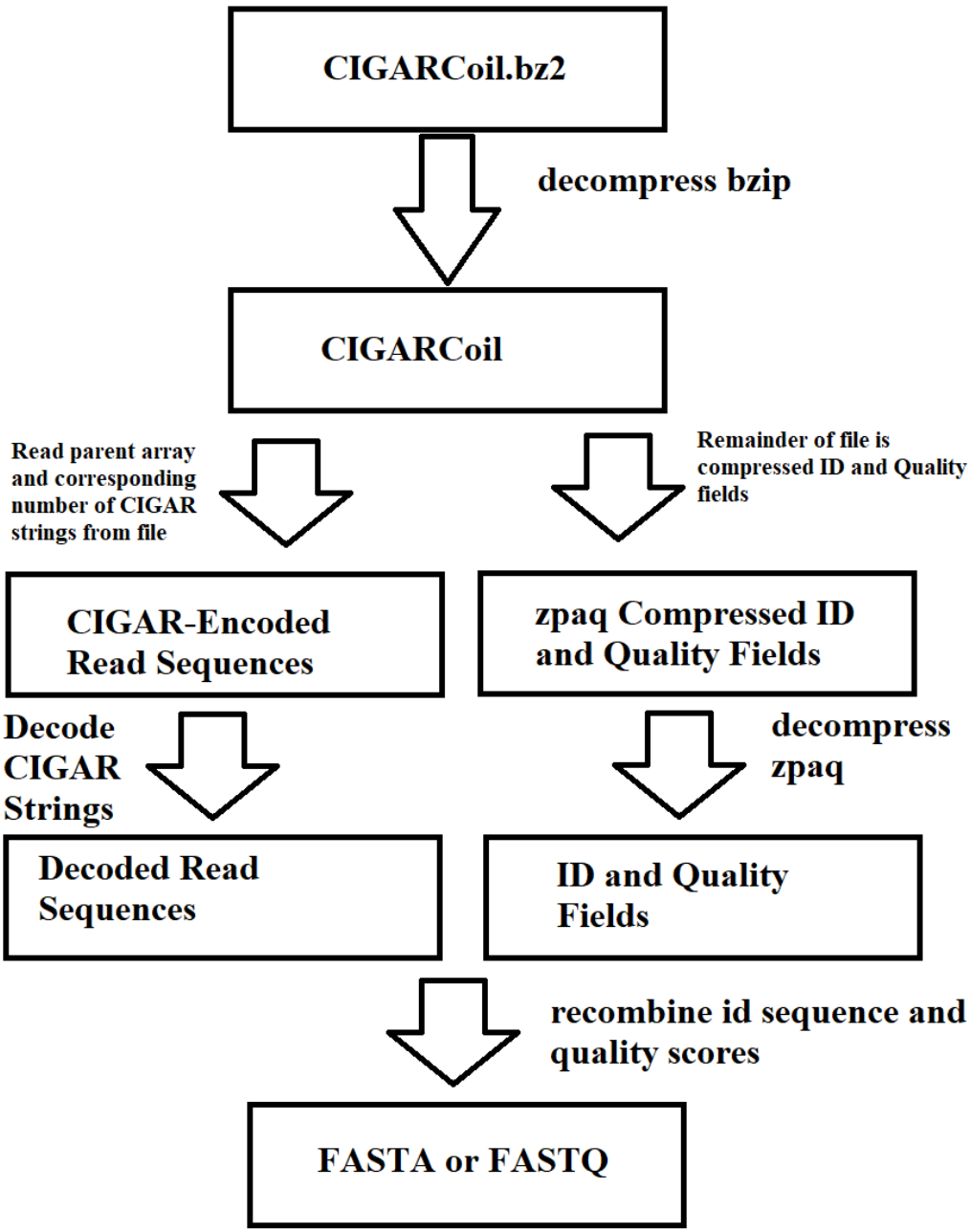

Figure 2.17: Image of CIGARCoil Decoding and Decompression Process that Occurs Once Per Run

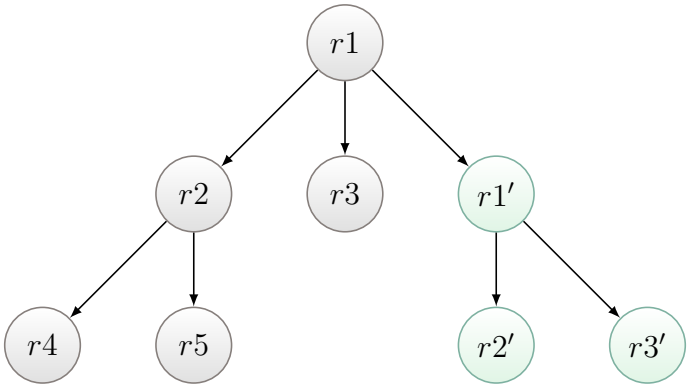

Figure 2.18: File Concatenation Example Tree

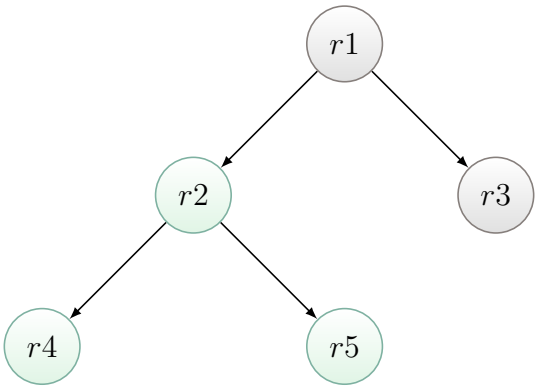

Figure 2.19: Incremental Update Example Tree

Chapter 3

Random Access and Predictive Cache

Random access of a read is a core feature of the CIGARCoil compression format. Random access works by traversing the CIGARCoil file's tree structure, decoding only what is necessary to obtain the read of interest to the user. Random access is made available to an end-user in the form of a square bracket [] operator. Since random access of a CIGARCoil file is not constant time, a predictive cache utility is implemented that prefetches data for the user based on their access patterns.

3.1 Random Access of Compressed File

An individual read of the compressed file can be accessed by decompressing only its parents recursively through the root of the MST. Accessing a read in this manner eliminates the need for decompressing the entire file within memory - providing the end user with a memory efficient means of reading the contents of a particular read in the compressed file. For example as seen in Figure 3.1 if $r4$ is being accessed, then we will decode $r4$'s parent, $r2$, relative to its parent, the root of the minimum spanning tree $r1$, then decode $r4$ relative to the now decoded $r2$. This circumvents the need to decode the other reads of the tree.

The Prim's algorithm provides us with a MST initially rooted arbitrarily at the first node of the graph. The height of the tree provides is a factor when we randomly access a

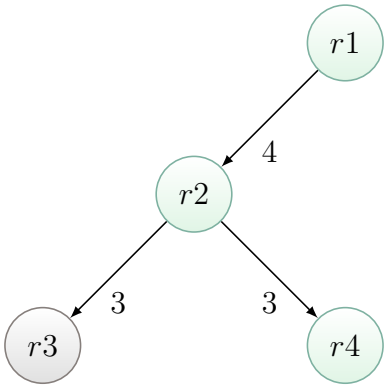

Figure 3.1: File Indexing Example Tree

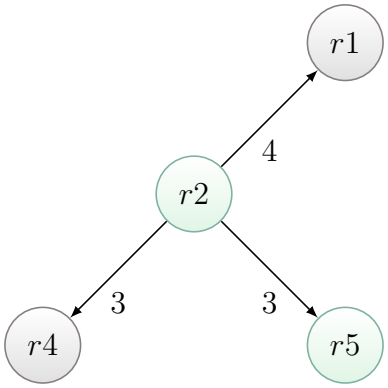

Figure 3.2: New tree after Root Changed to Minimize Height

read and need to decode the string by following the decoding method described previously as we move from the node (corresponding to the read) towards the root. Hence, it is desirable to find a node to be designated as the root that minimizes the height of the tree. Such a node in the graph literature is a called the *center* of the tree.

A tree that has been re-rooted to minimize tree height can be seen in Figure 3.2, which has re-rooted the example tree from Figure 3.1 to minimize the tree's height by changing the root from *r1* to *r2*. An algorithm for minimizing tree height can be seen in Algorithm 6, which takes a parent array and modifies it such that its root is in the center of the tree, reducing the number of traversals needed to reach the root from the average leaf node. The algorithm for centering the tree and minimizing its height works by pruning leaf nodes from the tree until at least two nodes remain. This algorithm is presented in Algorithm 6. These at least two remaining nodes are guaranteed to be at the center of the tree so the first is taken to be the root of the tree. With the new root, the original parent array is modified such that its indices now indicate the new root of the tree. This algorithm is known to run in linear time.

Using a data-set of the first two-million reads from the SRX001540 data set from table 5.6, the cost in time of randomly accessing the base-pair data is compared for both a CIGARCoil compressed file and the original FASTA file inFigure 3.3. On average for a CIGARCoil compressed file, random access requires 0.00112 seconds with a standard deviation of 0.00036. For a FASTA file, random access requires 0.00058 seconds on average with a standard deviation of 0.00010 seconds. As is to be expected, randomly accessing an uncompressed file requires less time than randomly accessing and decoding a record of the compressed file; however, despite the need to decode, random access of the CIGARCoil file only takes about twice as long as randomly accessing a record of an uncompressed file. These results are seen in Figure 3.3.

3.2 Predictive Cache

Since CIGARCoil supports the random access of elements within the file in $O(n)$, CIGARCoil is a candidate for the implementation of a predictive caching strategy where elements that an end-user is likely to request in the future can be fetched in advance. CIGARCoil's predictive cache learns which elements to fetch by using the reinforcement learning strategy, Q-Learning. A reinforcement learning approach has been chosen with the intention

Algorithm 6 Center Tree - Minimizing Tree Height

```
LET P be a parent array representation of the tree
LET n be the number of nodes
LET D be an empty array of size n for the degree of each node
LET A be an empty array of vectors that represents adjacency between nodes
for all i : n do
  LET j be P[i]
  insert i into A[j]
  insert j into A[i]
  D[j] = D[j] + 1
  D[i] = D[i] + 1
end for
LET Q be a FIFO queue
for all i : n do
  if D[i] == 1 then
    Q.push(i)
  end if
end for
LET m = n
while m >2 do
  for q : Q.size() do
    LET f be Q.pop()
    m = m - 1
    for a : A[f] do
      D[a] = D[a] - 1
      if D[a] == 1 then
        Q.push(a)
      end if
    end for
  end for
end while
LET r be Q.pop()
LET R be a parent array of size n
Q.clear()
for a : A[r] do
  R[a] = r
  Q.push(a)
  A[a].remove(r)
end for
while Q.size() >0 do
  LET v = Q.pop()
  for a: A[v] do
    R[a] = v
    Q.push(a)
    A[a].remove(v)
  end for
end while
return R
```

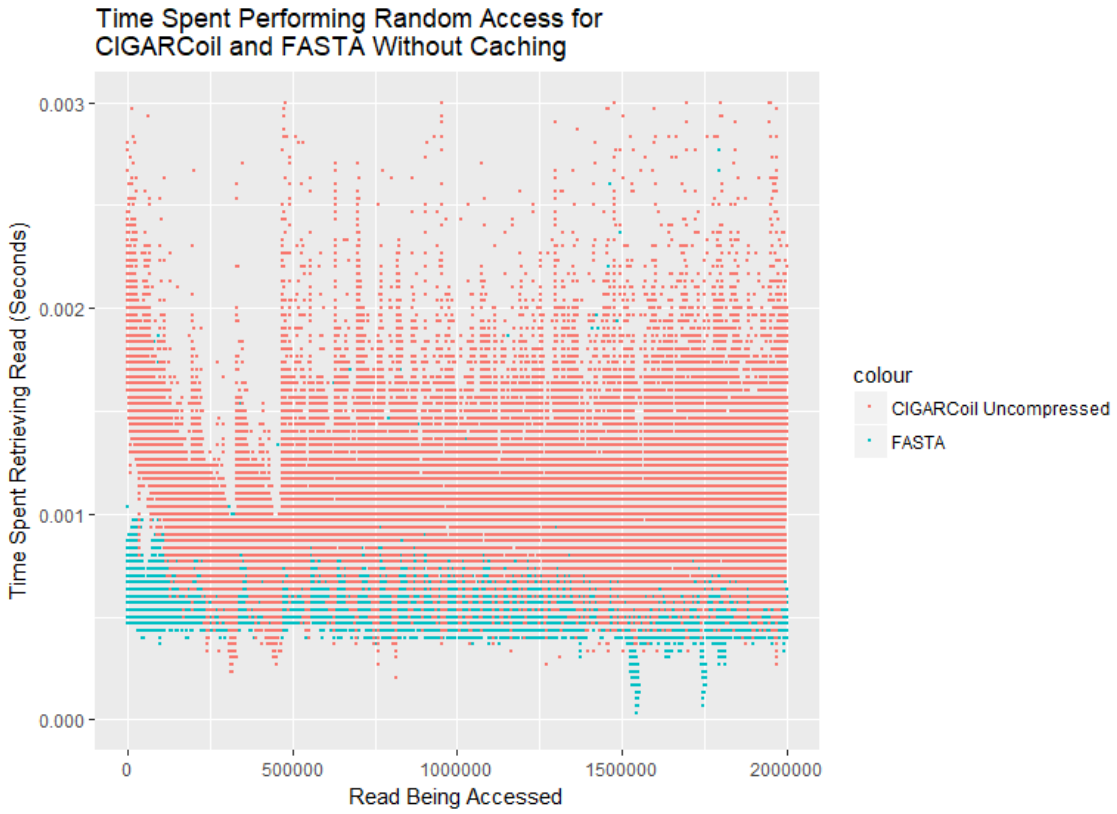

Figure 3.3: Time required to randomly access (and decode) a record from a FASTA and CIGARCoil file

of making this caching feature robust with a variety of different access patterns.

Cache Implementation The cached elements is user-defined sliding window of contiguous elements from the source file. An array of this user-defined size is stored in memory as well as the actual last and first index of the elements being stored. A sliding window of contiguous elements is used because the end-user is assumed to access elements of the file sequentially.

State Representation The state representation used for Q-learning has the following 12 states:

1. User requesting element in [0,10) percent of cache
2. User requesting element in [10,20) percent of cache
3. User requesting element in [20,30) percent of cache
4. User requesting element in [30,40) percent of cache
5. User requesting element in [40,50) percent of cache
6. User requesting element in [50,60) percent of cache
7. User requesting element in [60,70) percent of cache

8. User requesting element in [70,80) percent of cache
9. User requesting element in [80,90) percent of cache
10. User requesting element in [90,100) percent of cache
11. User requesting element before the first element of the cache
12. User requesting element after the last element of the cache

Actions that the Learning Agent can Take The predictive cache, the learning agent, can take one of the following 21 actions from any of the 12 states:

1. Make no change to the window
2. Advance the window 10 percent of the window-size forward
3. Advance the window 20 percent of the window-size forward
4. Advance the window 30 percent of the window-size forward
5. Advance the window 40 percent of the window-size forward
6. Advance the window 50 percent of the window-size forward
7. Advance the window 60 percent of the window-size forward
8. Advance the window 70 percent of the window-size forward
9. Advance the window 80 percent of the window-size forward
10. Advance the window 90 percent of the window-size forward
11. Advance the window 100 percent of the window-size forward
12. Move the window 10 percent of the window-size backward
13. Move the window 20 percent of the window-size backward
14. Move the window 30 percent of the window-size backward
15. Move the window 40 percent of the window-size backward
16. Move the window 50 percent of the window-size backward
17. Move the window 60 percent of the window-size backward
18. Move the window 70 percent of the window-size backward
19. Move the window 80 percent of the window-size backward
20. Move the window 90 percent of the window-size backward
21. Move the window 100 percent of the window-size backward

Rewarding the Learning Agent The learning agent receives a reward based on how close to the center of the window that the user's requested element is in. This reward is chosen because it encourages the predictive cache to keep elements cached in a manner that supports the user iterating forward or backward through the file. If the element requested is outside of the window, then a negative reward is given to the agent, discouraging behavior that led to this state in the future.

Q-Learning Parameters In my implementation a value of 0.01 is chosen for ϵ, indicating that the agent will take a random action 1 percent of the time. This is necessary for the agent to continue to explore different possibilities of actions instead of just choosing an action that maximizes its reward using its state-action table. A value of 0.1 is chosen for α, the learning rate. Choosing this somewhat high value for learning rate is done because it anticipates that the end-user might change their access pattern while accessing the data, allowing it to more quickly learn the new pattern with its high learning rate. A value of 0.01 is chosen for γ, the discount rate. This relatively low value for the discount rate is chosen because the value of a possible future reward is unimportant compared to whether or not the end-user is able to currently access the data that they are requesting.

Time Required to Access A Random Element Using a data-set of the first two-million reads from the SRX001540 data set from table 5.6, the cost in time of randomly accessing the base-pair data is compared for a CIGARCoil compressed file with and without caching in Figure 3.4. In this experiment random access with caching required 0.00038 seconds on average per access with a standard deviation of 0.00025 seconds. Random access without caching required 0.00112 seconds on average per access with a standard deviation of 0.00036 seconds. Caching elements in this manner provides a significant improvement in terms of random access speed. These results are seen in Figure 3.4.

Time Required to Perform Cache Window Adjustment Using a data-set of the first two-million reads from the SRX001540 data set from Table 5.6, the cost in time of adjusting the predictive cache's window, which in this experiment is set to a size of 1000 reads, is shown for both a CIGARCoil compressed file and the original FASTA file in Figure 3.5. On average filling the cache requires 1.053 seconds seconds for a CIGARCoil file with a standard deviation of 0.429 seconds. Filling the cache for a FASTA file required 0.574 seconds on average with a standard deviation of 0.144 seconds. These results are seen in Figure 3.5.

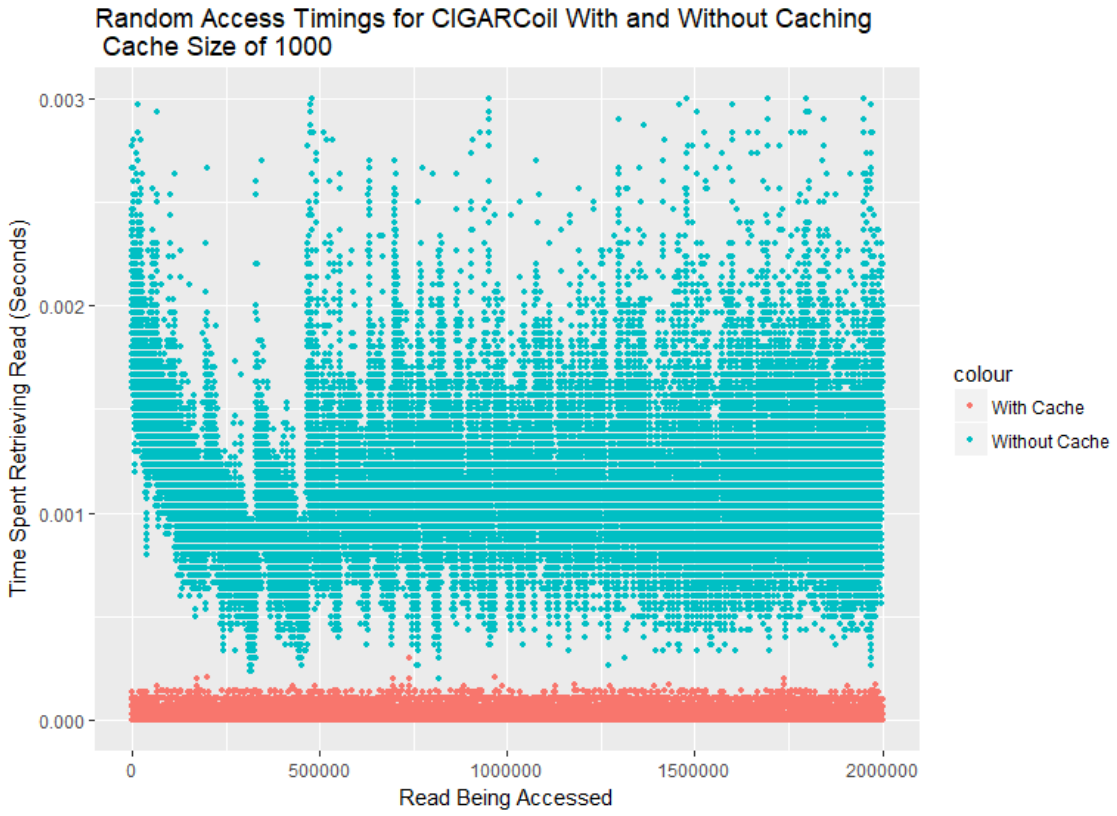

Figure 3.4: Cost of Random Access for a CIGARCoil file with and without caching

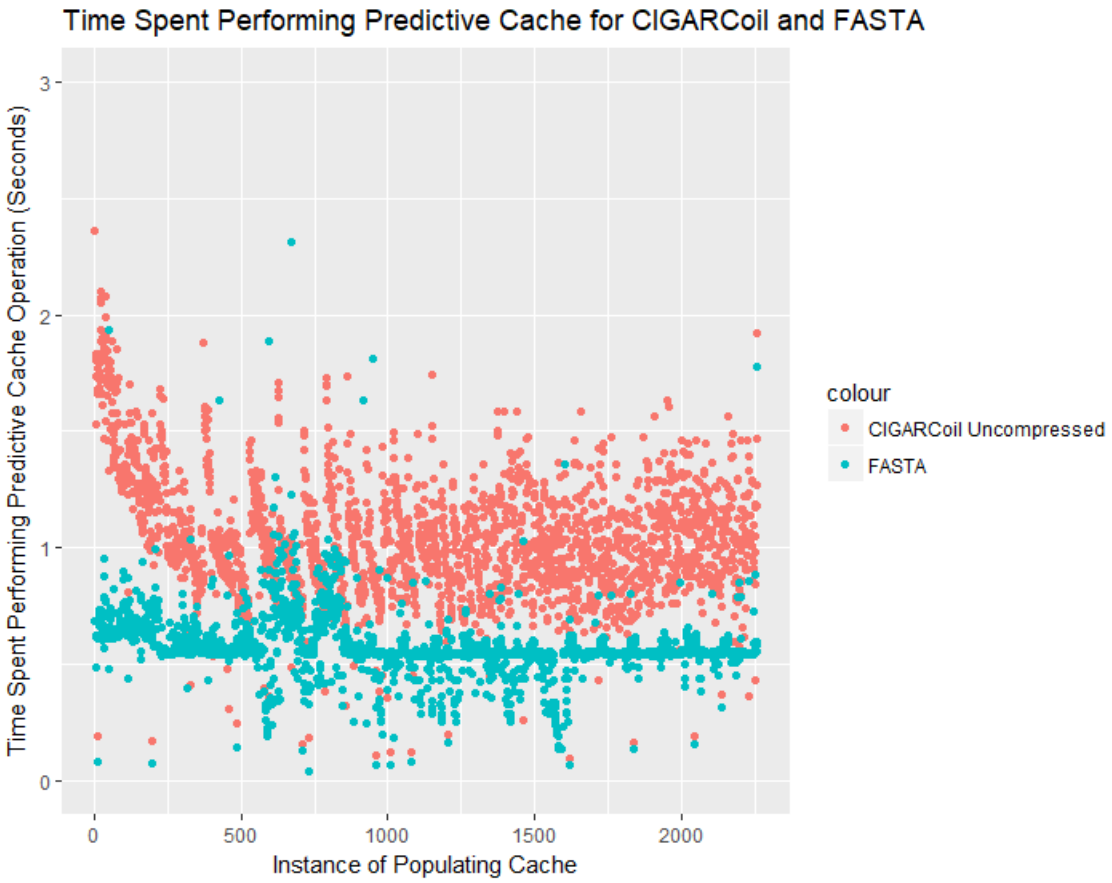

Figure 3.5: Cost Adjusting Predictive Cache for CIGARCoil compressed file and Uncompressed File

Chapter 4

CIGARCoil Clustering

Although the node compartmentalization heuristic that is applied reduces the amount of edit distance computations that need to be performed on the data set, it seemed that it should be possible to use a clustering strategy to offer further improvements to compression speed at the cost of some compression ratio performance. By applying clustering to the data set, similar reads can be clustered together, reducing the number of reads that the read compartmentalizing heuristic has to deal with in the first place. This chapter describes the implementation and results of applying clustering to the CIGARCoil compressor.

After the different parts of the input file have been split into their separate temporary files. A clustering algorithm can then be applied to the sequencing data, which has been shown to drastically reduce compression speeds at the cost of small compression ratio performance decrease as seen in Tables 4.1 and 4.2. K-Means clustering is used as the clustering algorithm because it runs in linear time, and the number of clusters and iterations used in the clustering are specified as command line arguments, allowing the end-user to determine the trade-off in compression that they are willing to incur for faster performance. The k-means implementation in this paper uses a set of k random strings of length equal to the average read length, which was calculated when the file was initially split into sequencing data, id data, and quality streams. Initially we sought to use Wagner-Fischer edit distance for our clustering algorithm's distance metric; however, the cost of applying Wagner-Fischer between the sequences and the centroids took a prohibitively long time. In order to work around this, the distance between a cluster and each of the centroids is instead calculated using Algorithm 7, which provides linear time complexity instead of Wagner-Fischer's quadratic time complexity. After clustering, each cluster becomes its own similarity graph, which are each separately encoded as MST as is done without clustering. Once all clusters have been encoded, they are all combined using the CIGARCoil concatenation idea that is shown in Figure 2.18. An illustration of the CIGARCoil workflow with clustering is shown in Figure 4.1.

4.1 Clustering Hyperparameter Experiments

The following clustering hyperparameters were run using a dataset that consists of the first five-hundred thousand reads of the SRR001540 dataset. These tests were run to provide an idea of the efficacy of the clustering algorithm on reducing compression speeds and how detrimental clustering is to the compression ratio of the resulting file. Each combination was run only once due to the amount of time required for each experiment.

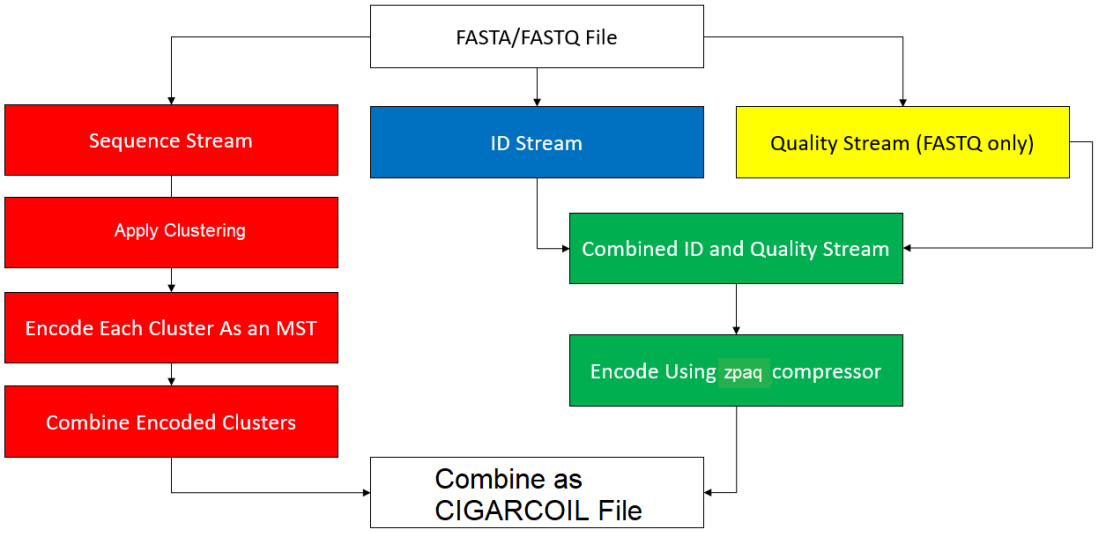

Figure 4.1: CIGARCoil Encoding Work-flow With Clustering

Algorithm 7 Clustering Linear-Time Similarity

```
LET s_1 be the first string
LET s_2 be the second string
LET similarity be 0
LET L be MIN(LENGTH(s_1), LENGTH(s_2))
for i : L do
  if s_1[i] == s_2[i] then
    similarity = similarity + 1
  end if
end for
return similarity
```

Table 4.1: Hyper Parameter Test Compression Ratios

Clusters/Iterations	**1**	**2**	**4**	**8**
1	0.267	-	-	-
2	0.279	0.278	0.279	0.279
4	0.283	0.282	0.282	0.282
8	0.286	0.286	0.286	0.285
16	0.288	0.288	0.287	0.287
32	0.290	0.290	0.290	0.289
64	0.292	0.292	0.292	0.292
128	0.294	0.293	0.293	0.293
256	0.296	0.296	0.295	0.295
512	0.298	0.297	0.297	0.296
1024	0.299	0.299	0.299	0.299
2048	0.301	0.300	0.300	0.300

Table 4.2: Hyper Parameter Test Compression Speed (Hours)

Clusters/Iterations	1	2	4	8
1	10.52	-	-	-
2	6.22	6.16	6.09	6.15
4	3.54	3.97	4.40	4.53
8	2.42	2.55	2.77	3.33
16	1.76	1.93	1.99	2.09
32	1.40	1.52	1.63	1.75
64	1.21	1.33	1.45	1.66
128	1.14	1.27	1.56	1.84
256	1.15	1.35	1.73	2.47
512	1.29	1.64	2.38	3.802
1024	1.62	2.30	3.64	6.32
2048	2.28	3.58	6.19	11.39

Choosing the Number of Iterations and Clusters After performing the clustering hyper parameter tests as seen in Table 4.1 and Table 4.2, we elected to choose 128 clusters with 1 training iteration. This combination was chosen because it reduced the required amount of time for compression of the 500,000 read dataset to 1.14 hours with a compression ratio loss of only 3 percent relative to a run without clustering, which required 10.52 hours. This combination of parameters is used for the other experiments of this section as seen in Table 4.3, Table 4.4, and Table 4.5.

Effect of Clustering As can be seen in Table 4.1 and Table 4.2, clustering improves the compression speed of CIGARCoil with a small reduction in compression ratio performance as the number of clusters increases. The number of iterations improves the compression ratio only slightly as seen in Table 4.1. Additionally, the cost of performing a large number of clustering iterations begins to outweigh the cost of performing the CIGARCoil algorithm as seen in Table 4.2. With the approximate edit distance metric used in our clustering implementation, many reads are assigned to the first centroid because they have nothing in common with any of the centroids. This causes the first cluster to almost always be significantly larger than the other clusters. This relationship between the number of reads in each cluster and the number of training iterations can be seen in Figure 4.2.

Compression Ratio vs. Number of Reads It can be observed in Figure 4.3 that the compression ratio continues to improve for CIGARCoil as the number of reads in the data set increases. CIGARCoil's compression ratio relative to the number of reads continues to improve as bzip and gzip's compression ratios level off.

Compression Speed vs. Number of Reads It can be observed in Figure 4.4 that although compression takes an extremely large amount of time, it is still increasing at an approximately linear rate.

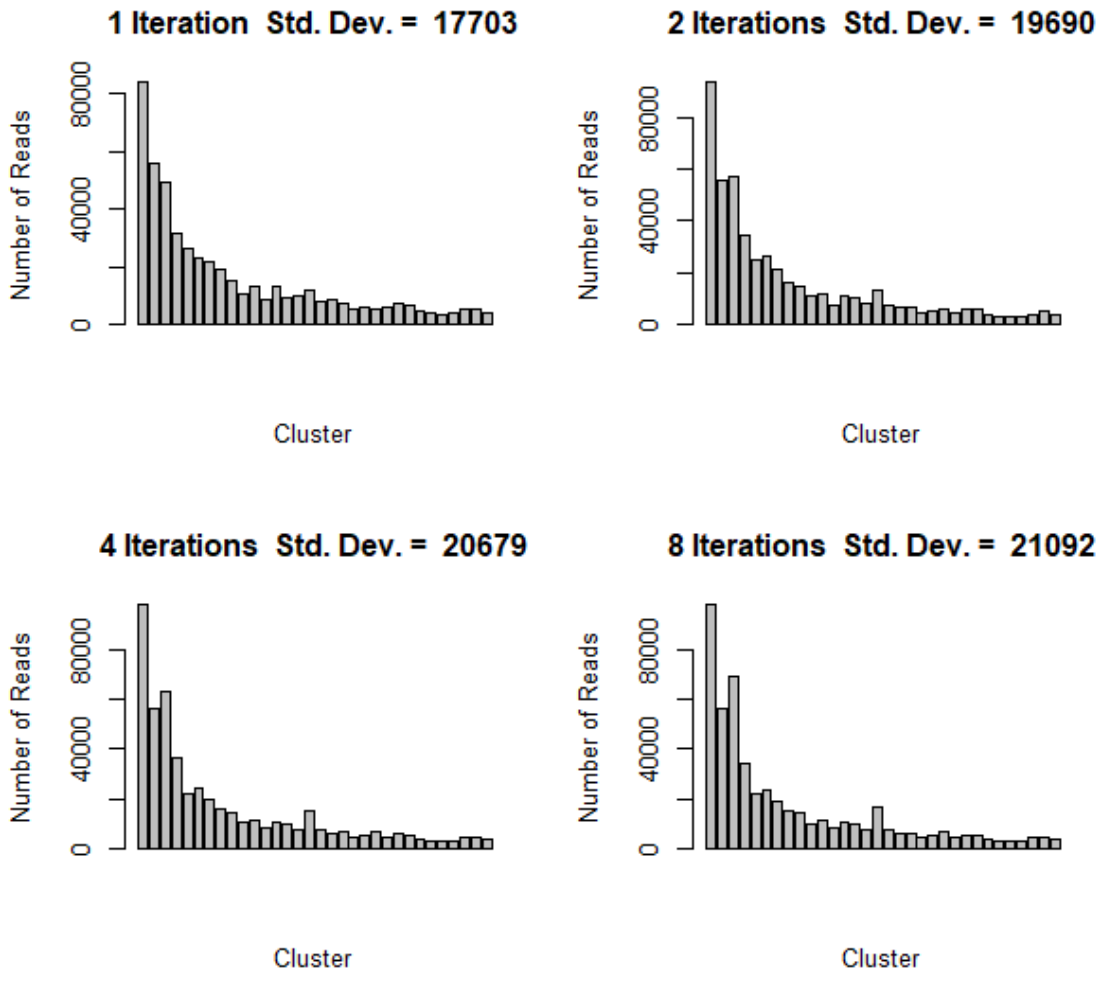

Figure 4.2: Effect of Clustering on Read Distribution With 32 Clusters

Figure 4.3: Compression Ratio vs Number of Reads With Clustering Performed on Reads from Beginning of the SRX001540 Dataset

Decompression Speed vs. Number of Reads It can be observed in Figure 4.5 that the decompression speed increases linearly as as the number of reads in the data set increases. Additionally decompression is significantly faster for CIGARCoil than compression as decompressing eight-million reads required less than an hour and compressing eight-million reads required just over three days.

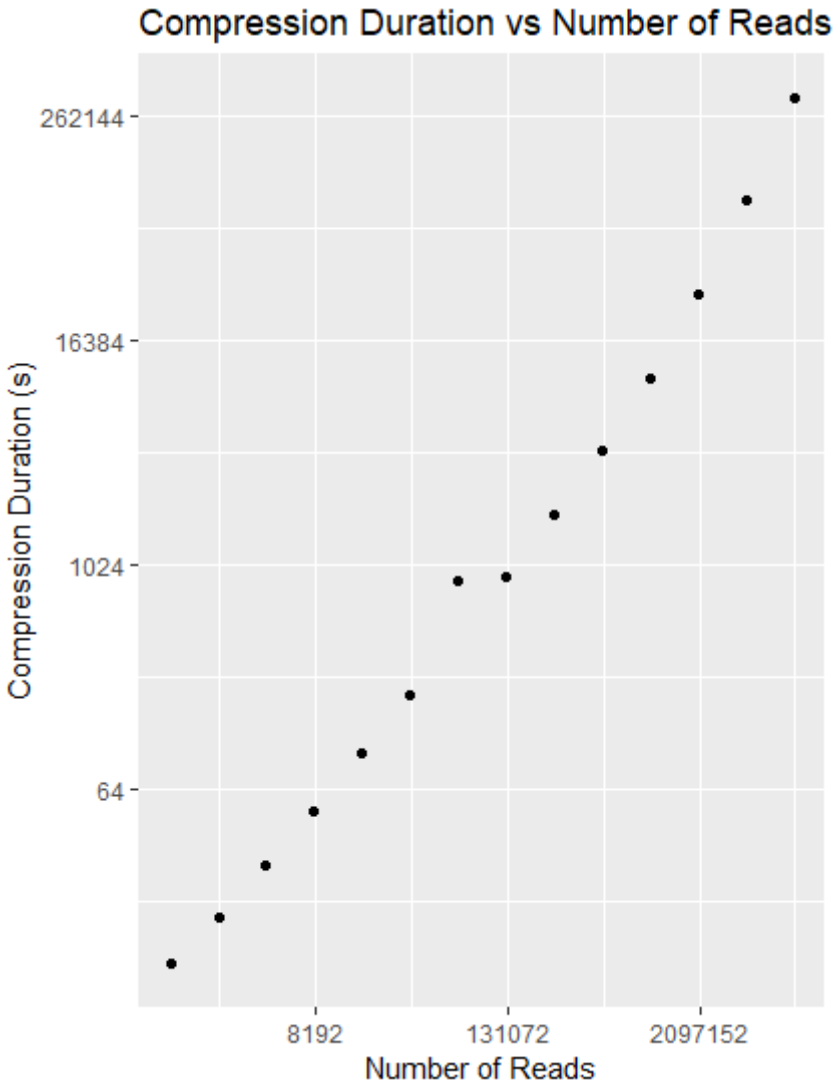

Figure 4.4: Compression Duration vs Number of Reads with Clustering Performed on Reads from Beginning of the SRX001540 Dataset

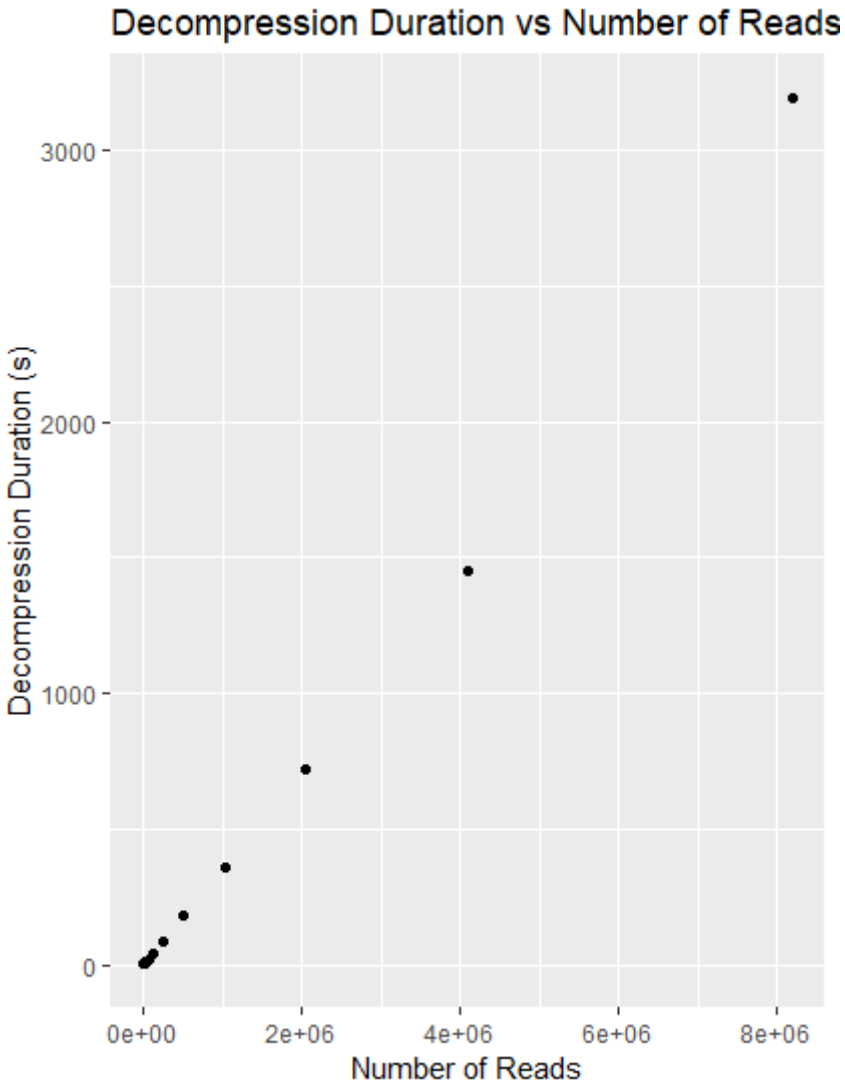

Figure 4.5: Decompression Duration vs Number of Reads With Clustering Performed on Reads from Beginning of the SRX001540 Dataset

Chapter 5

CIGARCoil Implementation

This chapter describes the software components that are used in the implementation of the CIGARCoil compressor. This chapter includes an overview of the source code that has been included in the appendix of this thesis as well as details as to how multi-threading was included. A description of the 2-byte CIGAR string struct is included in this section as well as a description of the modified Wagner-Fischer algorithm that yields edit distances in terms of CIGAR size is included in this chapter. Finally, a set of results of the CIGARCoil compressor in comparison to other compressors is included at the end of this chapter.

5.1 Source Code Overview

This section describes the source code that can be found in the appendix of this thesis.

CigarOperation This class defines the two-byte object that is described in table 5.2. This class has constructors that support the four different types of operations: match, substitution, deletion, and insertion. Additionally this class has methods that converts its four three bit fields into either a match or deletion length, or up to four characters. The source code for this class can be found in Appendix C.

Read This class represents a single read obtained from the DNA sequencing FASTA or FASTQ input file. This class stores statistical information for the read and the location of the read in the original file rather than the entirety of the read - taking a cue from the ReCoil compressor's usage of external memory algorithms. By doing this, only a 64-bit unsigned integer is used for each read rather than what is likely at least 300 bytes of id,quality, and sequence information. A key strength of storing a read's data like this is that the read's explicit id, quality, and base-pair information are only all used at once at the very beginning and very end of the encoding and compression step - making storing them in memory extremely wasteful. The source code for this class can be found in Appendix B.

Similarity Graph This class defines the similarity graph. The similarity graph is implemented as an adjacency list similar to 1.3. This implementation is chosen because the graph being constructed will be sparse and it allows for edges to be added in constant time. The head of each read's adjacency list contains a read object, which are accessed

by the order which they were read from the file in constant time. The source code for this class can be found in Appendix D.

Hash Bucket Index This class creates the data structure that is described inFigure 2.13. The source code for this class can be found in Appendix E.

Min Heap This class is used in my implementation of Prim's algorithm for finding the MST of the similarity graph. The source code for this class can be found in Appendix F.

Wagner-Fischer Matrix This class is used for finding the CIGAR-size edit distance between two strings. Additionally this class can return the set of cigar operations that must be performed to transform the first string into the next. This implementation of Wagner-Fischer makes use of modified cost functions that reflect the two-byte structure of table 5.2. The source code for this class can be found in Appendix G.

Decoded Reads This class stores the set of decoded reads for the DNA File Wrapper objects. A fixed number of reads is stored in this structure that stores the elements that are most frequently accessed. The source code for this class can be found in Appendix I.

CigarCoil Utilities This class provides a set of static helper methods that are used by the various other classes of this program. Methods include performing zpaq compression and decompression, concatenating files together, generating temporary files, and computing edit distances. The source code for this class can be found in Appendix A.

DNA File Wrapper This class provides the bulk of the functionality for this compressor. This class is instantiated by passing it a path to the file for the wrapper to be created around. Supported file types are FASTA, FASTQ, and CIGARCoil. This object provides a square bracket operator that an end-user can use to access a desired element of their choosing. Additionally, if the file is a FASTA or FASTQ file, the file can be compressed into a CIGARCoil file, and if the the file is a CIGARCoil file, then it can be decoded back into a FASTA or FASTQ file, whichever it was originally. The source code for this class can be found in Appendix H.

5.2 Adding Edges with Multiple Threads

Adding edges and performing the node compartmentalization heuristic is the most computationally expensive part of the CIGARCoil compressor. By splitting the number of nodes to add edges from by the umber of available cores on the machine, the tasks of querying the hashbucket index structure and adding edges can be performed by multiple threads simultaneously as seen in algorithm 8.

Algorithm 8 Adding Edges With Multiple Threads

```
LET c be the number of available cores
LET n be the number of reads to be compressed
for i : c do
  LET s be (c / n) × i
  if i != c then
    LET e be (c / n) × (i + 1)
  else
    LET e be n
  end if
  Call Add Edge Function for reads [s ... e]
end for
```

5.3 CIGAR Operation Struct

Information Theoretic Lower Bound In order to determine the minimum number of bits to use to represent each each character in the original file, an information theoretic lower bound can be used. The IUPAC (International Union of Pure and Applied Chemistry) specifies that the following six items can be expressed by a character in a read's sequencing data:

1. Adenine (A)
2. Cytosine (C)
3. Thymine (T)
4. Guanine (G)
5. Unknown (N)
6. Space (-)

Since we have six symbols that need to be represented, the theoretical minimum number of bits required can be calculated using $\lceil \log_2 6 \rceil$, which is 3.

Encoding of Each Base From the previously calculated theoretic lower bound, it is clearly seen that 3 bits are required to represent each base for the six different symbols. Table 5.1 illustrates the binary strings that have been given to the different base codes. A couple of extra values for uracil and emptiness have been added as these two three bit combinations would otherwise be wasted. Uracil, although not used by DNA, is used instead of T in RNA. Including this symbol allows CIGARCoil to support RNA FASTA and FASTQ files. The empty position code is used by CIGARCoil when there is an insertion or substitution with fewer than four characters. Having this empty position helps to facilitate using a fixed-length two-byte struct for each cigar operation as is seen in table 5.2.

Table 5.1: Base Encoding

Base	Binary Representation
Empty Position	000
Adenine	001
Cytosine	010
Thymine	011
Guanine	100
Unknown Base	101
Uracil	110
Space	111

CIGAR String Encoding In order to express our CIGAR strings in an efficient manner, a special CigarOperation structure is defined. This two-byte structure contains the following three components.

1. End of sequence bit 'E' that indicates the end of this read and the beginning of the next
2. Three bits that can represent up to eight different operations
3. 12 bits that either represent four base encodings or an integer value from [0, 4095].

Considering that not all FASTA/FASTQ files consist of reads that are of the same length, and that this compression scheme ought to support the concatenation of multiple compressed FASTA/FASTQ files, which certainly have no guarenteed read length among themselves. The following set of operations is instead used.

Table 5.2: Supporting up to Eight Operations with Unfixed Read Length

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**0**	**1**	**2**	**3**	**4**	**5**
E	0			Match Length											
E	1			mismatch0			mismatch1			mismatch2			mismatch3		
E	2			insert0			insert1			insert2			insert3		
E	3			Deletion Length											

5.4 Customized Wagner-Fischer Algorithm

Although Wagner-Fischer is an excellent algorithm for calculating an edit distance between two strings, its notion of an operation doesn't coincide with the 2-byte struct that we are using that favors the match and deletion operations over the insertion and substitution operations. CIGARCoil makes use of a modified Wagner-Fischer Matrix that allows subsequent match and deletion operations to only cost one operation, as well as groups of up to 4 insertions/substitutions also only costing 1 operation. By using a customized Wagner-Fischer matrix, we can find an edit distance that corresponds directly

with the amount of resources that will be used to encode the child relative to its parent. Additionally, this custom Wagner-Fischer Matrix allows an optimal CIGAR string using our struct to be generated, whereas the standard wagner-fischer matrix would frequently lead to CIGAR strings that contain many match 1 operations, which are inefficient compared to a substitute multiple characters operation. For example when finding an edit distance for the two strings ACTG and GCTA, the traditional Wagner-Fischer matrix would indicate to substitute the first character, match the next two characters, and then substitute the final character, but the custom wagner-fischer matrix based on our struct would instead opt for one substitution operation that covers all 4 of the characters. We have decided to refer to the edit distance from this matrix as the CIGAR size.

Table 5.3: Custom Wagner-Fischer Matrix Example

	ϵ	**A**	**A**	**G**	**G**	**T**	**C**	**C**	**C**
ϵ	0	1	1	1	1	2	2	2	2
G	1	1	2	2	2	2	3	3	3
A	1	2	1	2	2	3	2	3	3
A	1	2	2	1	2	3	3	2	3
A	1	2	2	2	1	2	2	2	2
A	1	2	2	2	2	2	3	3	3
C	1	2	2	2	2	3	2	3	3
C	1	2	2	2	2	3	3	2	3
C	1	2	2	2	2	3	3	3	2
C	1	2	2	2	2	3	3	3	3

Side-effect of Modifying Wagner-Fischer algorithm In the traditional Wagner-Fischer algorithm, all operations are weighted the same, making the algorithm symmetrical (*i.e.*, $WF(s_1, s_2) = WF(s_2, s_1)$); however, since we are using different weights for the operations to reflect our structure in table 5.2, our implementation of Wagner-Fischer loses its symmetry. A simple example of two strings yielding asymmetric CIGAR sizes is as follows for s_1 of AAAAACCCC and s_2 of CCCC:

Table 5.4: String s_1 relative to s_2

	ϵ	**A**	**A**	**A**	**A**	**A**	**C**	**C**	**C**	**C**
ϵ	0	1	1	1	1	2	2	2	2	3
C	1	1	2	2	2	2	3	3	3	3
C	1	2	1	2	2	2	2	3	3	3
C	1	2	2	1	2	2	2	2	3	3
C	1	2	2	2	1	2	2	2	2	3

As can be seen in tables 5.4 and 5.5, the modified version of Wagner-Fischer is not symmetrical as proven by counter-example. This means that an edge from s_1 to s_2 should should more accurately be a directed edge rather than an undirected edge. We have

Table 5.5: s_2 relative to s_1

	ϵ	**C**	**C**	**C**	**C**
ϵ	0	1	1	1	1
A	1	1	2	2	2
A	1	2	1	2	2
A	1	2	2	1	2
A	1	2	2	2	1
A	1	2	2	2	2
C	1	2	2	2	2
C	1	2	2	2	2
C	1	2	2	2	2
C	1	2	2	2	2

elected to continue to use undirected edges in this thesis because we make an assumption that edges representing greatest similarity will consist of mostly substitution and match operations. A CIGAR string that consists of only match and substitution operations is symmetrical under our implementation.

$$MATCH \implies A[i] = B[i]$$

$$SUB \implies A[i] \neq B[i]$$

If A relative to B is the CIGAR specifying a match of size x, followed by substitution of y characters, followed by match of z characters, then A must be equal to: $B_0B_1...B_{x-1}A_xA_{x+1}...A_{x+y-1}B_{x+y}B_{x+y+1}...B_{x+y+z+-1}$. With A equivalent to B in those terms, it is obvious that B relative to A can also be represented with a CIGAR specifying match of size x, followed by a substitution of y characters, followed by a match of size z characters. In terms of A, B must be equal to: $B_0B_1...B_{x-1}B_xB_{x+1}...B_{x+y-1}B_{x+y}B_{x+y+1}...B_{x+y+z+-1}$, showing the symmetry of match and substitution operations.

5.5 Results

The following results represent CIGARCoil as compared to a few other reference-string-free compression algorithms (SeqSqueeze1, DSRC2, and LFQC) as well as bzip. The results for these other compressors are obtained from [16], and the CIGARCoil algorithm is run on the same set of publicly available FASTQ data sets.

CIGARCoil vs. Other Compression Schemes In order to show the performance of CIGARCoil in contrast to other existing algorithms. A set of four benchmark data sets have been chosen as seen in table 5.6. The first three FASTQ data sets are data sets used in the benchmarking of the LFQC and DSRC2 compressors in [16]. Like CIGARCoil, LFQC and DSRC2 are reference-genome free lossless compressors that can process FASTQ files. The other data sets used in [16] could either not be located or used a color-space representation of the base-pairs, which is supported by CIGARCoil presently. The

fourth data set is the only real data set that the ReCoil compressor was applied against in [28].

bzip2 This compressor, bzip2, is a general purpose compression algorithm that compresses files using Burrows-Wheeler transforms. Its inclusion in these results is to highlight the differences in performance between these specialized DNA compressors and general purpose compressors. Bzip does not provide any special access to the compressed file, such as a square bracket operator, and it has been shown to be inferior to specialized DNA sequence compressors such as LFQC, DSRC2 in previous work [16].

LFQC LFQC is an algorithm for the compression of DNA sequences that was first proposed in 2014 in [16]. This algorithm uses lossless and non-reference based compression on FASTQ files. This compression scheme compresses the FASTQ file's identification, sequence, and quality score information separately, each using a different algorithm that performs run-length encoding. Although this algorithm achieves impressive results in terms of compression ratio and speed, it does not provide for special access to the compressed file.

DSRC2 DSRC2 is an algorithm for the compression of DNA sequences that was first proposed in 2010 in [4]. Similarly to LFQC, this algorithm also treats IDs, sequences, and quality scores as separate streams during compression, making use of different forms of run-length encoding. This algorithm

ReCoil The ReCoil algorithm, from which CIGARCoil has adapted the similarity graph idea is also used for comparison.

Interpretation of Results Table 5.7 illustrates the performance of CIGARCoil versus the general-purpose compressor, Bzip2, as well as the specialized compressors DSRC2 and LFQC. CIGARCoil encoding alone shows that the file can be reduced to less than a third in the case of the SRR001471 data set, with CIGARCoil in its zipped state outperforming or rivaling other specialized compressors in terms of compression ratio. In terms of compression speed, as seen in table 5.8, CIGARCoil requires significantly more time to run than any of the other compressors being compared to, which can be contributed to the amount of time that is required to generate the CIGARCoil similarity graph in comparison to the block-compression approaches to bzip2, DSRC2, and LFQC. In table 5.9, CIGARCoil is shown to be slower at decompression than the other approaches as well. This can be attributed to uncompressing the zpaq compression that is applied to the entirety of the file's meta-data. Random access of the base-pair data of the file is still fast and possible.

Comparing against ReCoil Although ReCoil and CIGARCoil are relatively similar due to their usage of a similarity graph while encoding base-pair data, ReCoil is heavily specialized in that it only processes FASTA data with fixed-read lengths without supporting meta-data, which is why it can not be applied to many of the data sets in 5.6. In the ReCoil paper, Yanovsky runs ReCoil on a variety of synthetic data sets, which were generated by making random samples of fixed length from Human Chromosome 14, where each nucleotide had a 2 percent change of being modified and each substring had

Table 5.6: Benchmark Datasets Used

Dataset	**Type**	**Size [GB]**	# of Reads	Avg. Read Length
SRR001471	LS454-FASTQ	0.393210	629071	275.210
SRR003186	LS454-FASTQ	1.575455	1280292	584.362
SRR003177	LS454-FASTQ	1.802988	1504571	568.203
SRR001540	Illumina-FASTA	17.684191	192132427	36

Datasets Downloaded from the National Center for Biotechnology Information

Table 5.7: Compression Percent (File Size of Compressed / File Size of Original)

	SRR001471	**SRR003186**	**SRR003177**
bz2	24.248	26.034	26.680
Dsrc2 *	20.665	23.026	21.739
LFQC *	19.062	21.538	19.585
ReCoil	-	-	-
CIGARCoil	29.081	34.302 **	33.656 **
CIGARCoil.bz2	19.933	23.581 **	23.048 **
CigarCoil clustered	31.036	34.654 **	34.021 **
CIGARCoil.bz2 clustered	20.471	23.752 **	22.953 **

* Results taken from [16]

** Results Created With 100,000 Read Sample

Table 5.8: Compression Speeds in MiB/s (Millions of Bytes per Second)

	SRR001471	**SRR003186**	**SRR003177**
bz2	2.604	2.783	2.056
Dsrc2 *	13.102	25.217	30.369
LFQC *	0.759	1.080	1.053
ReCoil	-	-	-
CIGARCoil	0.0012047	0.0010671 **	0.0010964 **
CIGARCoil.bz2	0.0012046	0.0010671 **	0.0010963 **
CigarCoil clustered	0.00173	0.0013925 **	0.0014363 **
CIGARCoil.bz2 clustered	0.00171	0.0013923 **	0.0014362 **

* Results taken from [16]

** Results Created With 100,000 Read Sample

Table 5.9: Decompression Speeds in MiB/s (Millions of Bytes per Second)

	SRR001471	**SRR003186**	**SRR003177**
bz2	3.818	5.293	3.920
Dsrc2 *	6.907	9.608	9.670
LFQC *	0.717	1.217	1.173
ReCoil	-	-	-
CIGARCoil	0.0388	0.358 **	0.357 **
CIGARCoil.bz2	0.0386	0.358 **	0.357 **
CigarCoil clustered	0.0382	0.346 **	0.353 **
CIGARCoil.bz2 clustered	0.0381	0.346 **	0.353**

* Results taken from [16]

** Results Created With 100,000 Read Sample

a 5 percent change of being reverse complimented (*i.e.*, replacing As with Ts, Gs with Cs, and vice-versa) [28]. We generated our own synthetic data with the same parameters from Human Chromosome 14; however, in the interest in running the data set in a reasonable amount of time we generated a smaller data set of two million reads, whereas the other compressors were run on a data set of 50 million reads. Due to the nature of how CIGARCoil achieves better compression ratio performance with larger numbers of reads, it is plausible to anticipate that CIGARCoil would further improve upon its result on this synthetic data if it were also run on a set of 50 million reads. Table 5.10 shows how CIGARCoil stacks up against other compressors on this synthetic data. CIGARCoil and ReCoil are clearly superior to general-purpose compressors like 7zip and bzip on this data. ReCoil outperforms CIGARCoil on this data set with a compression ratio that is 2 percent of the original file size better, while being significantly faster than CIGARCoil. We believe that this difference can be attributed to the highly specialized nature of ReCoil towards files with fixed read lengths without meta data, which this synthetic file conformed to; additionally, the ReCoil compressor does not store a parent array for the tree but instead stores the encoded reads in pre-order traversal order, which saves 200 megabytes in this particular instance where the initial uncompressed file was 1800 megabytes. We use a parent array in our CIGARCoil implementation, which is utilized for random access of the compressed reads within the file.

Table 5.10: Comparison Against ReCoil on Synthetic Data Set

	Time (Min)	**Compression Ratio**
7zip *	300	0.27
bzip *	45	0.36
ReCoil *	290	0.18
CIGARCoil	1190	0.20

* Results taken from [28]

Chapter 6

Conclusion

Building off of the work done on the Coil and ReCoil DNA compression algorithms, I have incorporated the reference-based compression ideas presented in [7] and [28] to provide a new compression algorithm, CIGARCoil, which seeks to improve upon these two approaches by supporting a wider variety of input data, while providing compression results that are approximately the same if not better than those of other compressors. Furthermore, the CIGARCoil compression scheme also supports several operations that are of use to end-users such as random access of the compressed file, using a square-bracket operator, updating a read within the compressed file, and concatenating multiple compressed files together. Additionally, CIGARCoil can utilize a predictive caching mechanism that pre-fetches records for an end-user, further increasing the utility of the CIGARCoil format. Ultimately, CIGARCoil provides a compression scheme that provides end-users with compression ratios that are on-par or better than other compressors as seen in tables 5.7 and 5.10, while offering end-users additional usability of the compressed file, reducing the number of reasons that an end-user would have for decompressing and recompressing a file, and making it possible to store and randomly access large numbers in main memory on commodity machines. The primary drawback of CIGARCoil is that it takes a significant amount of time to encode and compress data sets as seen in table 5.8.

CIGARCoil and MPEG-G The CIGARCoil format addresses three areas of the MPEG-G white paper standard, which are enabled with the underlying tree structure of the encoded sequencing data:

1. random access of the file
2. concatenation of compressed files
3. update of the content of the compressed files

Democratization of Research By providing a DNA sequencing read compression format that supports the random access of the compressed file, end-users are provided with a compress-once format that enables them to perform work on compressed files without the need for decompressing the file as they can iterate over the compressed file with a square bracket operator. This reduces the amount of resources required for processing the compressed DNA reads, enabling more researchers to contribute to this area.

6.1 Future Work

We foresee the following areas of future work for the CIGARCoil compressor, which will improve its compression speed performance as well as increase its utility by supporting color-space encoded reads as well as supporting streaming compression.

Support Color-Space Encoded Reads The datasets used in this paper consist of sequencing data that consists of A, C, T, G, and N; however, there are some types of Datasets, such as SOLid, which use color-space encoded reads where instead of A, C, T, G, and N, numerical values are used that represent a base's value relative to the base in the sequence. The encoding algorithm could be modified to support color-space encoded reads.

Distributed Computing Approach It seems possible to perform some components of the CIGARCoil compression, notably the construction of the hashbucket inedex structure by using a tool like Hadoop's map-reduce.

GPGPU Computing The largest weakness of this approach currently is the tremendous length of the compression time. Fortunately, the most time-intensive parts of the compression algorithm, the node compartmentalizing heuristic and the edit distance computation can be enhanced by integrating GPGPU computing, reducing the time required to compress the file by utilizing more potentially already available resources.

Streaming Compression We believe that this algorithm could be adapted to work with streaming compression. Such an approach could work during the edge adding part of the algorithm, where sequence data are streamed in and immediately added to the similarity graph. Once the data are finished being read, then the minimum spanning tree and CIGAR encoding steps can occur, finishing the compression of the file.

Bibliography

[1] Claudio Alberti, Tom Paridaens, Jan Voges, Daniel Naro, Junaid J Ahmad, Massimo Ravasi, Daniele Renzi, Paolo Ribeca, Giorgio Zoia, Idoia Ochoa, et al. An introduction to mpeg-g, the new iso standard for genomic information representation. *bioRxiv*, page 426353, 2018.

[2] James K Bonfield and Matthew V Mahoney. Compression of fastq and sam format sequencing data. *PloS one*, 8(3):e59190, 2013.

[3] Cornel Constantinescu and Gero Schmidt. Fast and efficient compression of next generation sequencing data. In *2018 Data Compression Conference*, pages 402–402. IEEE, 2018.

[4] Sebastian Deorowicz and Szymon Grabowski. Compression of dna sequence reads in fastq format. *Bioinformatics*, 27(6):860–862, 2011.

[5] Sebastian Deorowicz and Szymon Grabowski. Data compression for sequencing data. *Algorithms for Molecular Biology*, 8(1):25, 2013.

[6] Jack Edmonds. Optimum branchings. *Journal of Research of the national Bureau of Standards B*, 71(4):233–240, 1967.

[7] Markus Hsi-Yang Fritz, Rasko Leinonen, Guy Cochrane, and Ewan Birney. Efficient storage of high throughput dna sequencing data using reference-based compression. *Genome research*, 2011.

[8] Gary Gordon and Elizabeth McMahon. A greedoid polynomial which distinguishes rooted arborescences. *Proceedings of the American Mathematical Society*, 107(2):287–298, 1989.

[9] Ananth Grama, Vipin Kumar, Anshul Gupta, and George Karypis. *Introduction to parallel computing*. Pearson Education, 2003.

[10] Elena Grassi, Federico Di Gregorio, and Ivan Molineris. Kungfq: A simple and powerful approach to compress fastq files. *IEEE/ACM transactions on computational biology and bioinformatics*, 9(6):1837–1842, 2012.

[11] Eric D Green, Mark S Guyer, and National Human Genome Research Institute. Charting a course for genomic medicine from base pairs to bedside. *Nature*, 470(7333):204, 2011.

[12] Joseph B Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50, 1956.

[13] Heng Li, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis, and Richard Durbin. The sequence alignment/map format and samtools. *Bioinformatics*, 25(16):2078–2079, 2009.

[14] Stuart Lloyd. Least squares quantization in pcm. *IEEE transactions on information theory*, 28(2):129–137, 1982.

[15] Michael Nelson, Sridhar Radhakrishnan, Amlan Chatterjee, and Chandra N Sekharan. Queryable compression on streaming social networks. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 988–993. IEEE, 2017.

[16] Marius Nicolae, Sudipta Pathak, and Sanguthevar Rajasekaran. Lfqc: a lossless compression algorithm for fastq files. *Bioinformatics*, 31(20):3276–3281, 2015.

[17] Robert Clay Prim. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401, 1957.

[18] Michael J Quinn and Narsingh Deo. Parallel graph algorithms. *ACM Computing Surveys (CSUR)*, 16(3):319–348, 1984.

[19] David Salomon and Giovanni Motta. *Handbook of data compression*. Springer Science & Business Media, 2010.

[20] Claude Elwood Shannon. A mathematical theory of communication. *Bell system technical journal*, 27(3):379–423, 1948.

[21] Jay Shendure and Hanlee Ji. Next-generation dna sequencing. *Nature biotechnology*, 26(10):1135, 2008.

[22] Zachary D Stephens, Skylar Y Lee, Faraz Faghri, Roy H Campbell, Chengxiang Zhai, Miles J Efron, Ravishankar Iyer, Michael C Schatz, Saurabh Sinha, and Gene E Robinson. Big data: astronomical or genomical? *PLoS biology*, 13(7):e1002195, 2015.

[23] Robert Endre Tarjan. Finding optimum branchings. *Networks*, 7(1):25–35, 1977.

[24] J Craig Venter, Mark D Adams, Eugene W Myers, Peter W Li, Richard J Mural, Granger G Sutton, Hamilton O Smith, Mark Yandell, Cheryl A Evans, Robert A Holt, et al. The sequence of the human genome. *science*, 291(5507):1304–1351, 2001.

[25] Robert A Wagner and Michael J Fischer. The string-to-string correction problem. *Journal of the ACM (JACM)*, 21(1):168–173, 1974.

[26] Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards*. PhD thesis, King's College, Cambridge, 1989.

[27] W Timothy J White and Michael D Hendy. Compressing dna sequence databases with coil. *BMC bioinformatics*, 9(1):242, 2008.

[28] Vladimir Yanovsky. Recoil-an algorithm for compression of extremely large datasets of dna data. *Algorithms for Molecular Biology*, 6(1):23, 2011.

Appendices

Appendix A

CIGARCoil Utilities

A.1 Header File

```
#ifndef CIGAR_COIL_UTIL_H
#define CIGAR_COIL_UTIL_H

#include <string>
#include "WagnerFischerMatrix.h"
#include "SimilarityGraph.h"
#include "MinHeap.h"
#include <vector>
#include <queue>
#include<fstream>
#include "libzpaq.h"

// Provides a set of static methods to be called by the various methods of this compressor
class CigarCoilUtilities
{
public:

  // returns the wagner fischer edit distance in terms of CIGAR operations
  static unsigned short getWagnerFischerEditDistance(const std::string *originString, const std::string *targetString);

  // returns a cheap linear time similarity distance based on the diagonal of a Wagner Fischer Edit Distance matrix only
  static unsigned short getCheapSimilarityDistanceMetric(const std::string *originString, const std::string *targetString, unsigned short currentBest);

  // returns the first index of a vector that is greater than a target value
  static unsigned int findFirstIndexGreaterThanTarget(const vector<unsigned int> *v, unsigned int target);

  // returns the first index of a vector that is greater than a target value
  static unsigned int findFirstIndexGreaterThanTarget(const vector<double> *v, double target);

  // creates a temporary file and returns the string of the path to the temporary file
```

```
31    static string createTemporaryFile();
32
33    // returns data at the given position of a file
34    static string getDataAtFilePosition(unsigned long long pos, unsigned
       short length, ifstream *fileStream);
35
36    // returns a parent array of the provided graph object
37    static unsigned int* PrimMST(SimilarityGraph* graph);
38
39    // re-centers a tree to minimize its height
40    static unsigned int* getMinimumHeightTree(const unsigned int* parentArray
       , size_t numberOfElements, unsigned int *root);
41
42    // returns which element of 4 is larger in value
43    static char greatestOfFour(double val1, double val2, double val3, double
       val4);
44
45    // encodes file of quality values and returns name of encoded file
46    static string encodeZpaq(const char *qualityFile);
47
48    // decodes zpaq compressed file
49    static string decodeZpaq(const char *compressedZpaqFile);
50
51  private:
52    // Disallow creating an instance of this class
53    CigarCoilUtilities() {}
54  };
55
56  // Implement a zpaq reader object for zpaq calls
57  class ZpaqReader : public libzpaq::Reader {
58  private:
59    std::ifstream *_fileStream;
60  public:
61    ZpaqReader(std::ifstream *fileName);
62    // should return 0..255, or -1 at EOF
63    int get();
64    // read to buf[n], return no. read
65    int read(char* buf, int n);
66    ~ZpaqReader() {};
67
68  };
69
70  // Implement a zpaq writer object for zpaq calls
71  class ZpaqWriter : public libzpaq::Writer {
72  private:
73    std::ofstream *_fileStream;
74  public:
75    ZpaqWriter(std::ofstream *fileStream);
76    // should output low 8 bits of c
77    void put(int c);
78    // write buf[n]
79    void write(const char* buf, int n);
80    ~ZpaqWriter() {};
81  };
82
83  #endif // !CIGAR_COIL_UTIL_H
```

A.2 Definitions

```
#include "CigarCoilUtilities.h"

// uses a heuristic for the similarity distance between two reads or returns early if it has no chance of beating the current best edit distance
unsigned short CigarCoilUtilities::getCheapSimilarityDistanceMetric(const std::string *originString, const std::string *targetString, unsigned short currentBest) {

  size_t smallerLength = min(originString->length(), targetString->length());

  unsigned short similarity = 0;

  short checkpoint = smallerLength - currentBest;

  for (size_t c = 0; c < smallerLength; c++) {
    if (originString->at(c) == targetString->at(c)) {
      similarity++;
    }

    // quit early
    if (c == checkpoint && similarity < currentBest) {
      return similarity;
    }

  }

  return similarity;

}

// compute wagner fischer edit distance (CIGAR size) of two strings
unsigned short CigarCoilUtilities::getWagnerFischerEditDistance(const std::string *originString, const std::string *targetString) {

  return WagnerFischerMatrix(originString, targetString).getEditDistance();

}

// uses binary search to find the first vertex of a vector that is greater than the given target
unsigned int CigarCoilUtilities::findFirstIndexGreaterThanTarget(const vector<unsigned int> * preSortedVector, unsigned int target) {

  size_t low = 0;
  size_t high = preSortedVector->size();
  while (low != high) {
    size_t lowHighSum = low + high;
    unsigned int mid = (lowHighSum) / 2;
    if (preSortedVector->at(mid) <= target) {
      low = mid + 1;
    }
    else {
      high = mid;
    }
```

```
49    }
50    return high;
51  }
52
53  // returns the first index greater than the requested target
54  unsigned int CigarCoilUtilities::findFirstIndexGreaterThanTarget(const
        vector<double> * preSortedVector, double target) {
55
56    // don't even bother searching
57    if (preSortedVector->at(preSortedVector->size() - 1) < target) {
58      return preSortedVector->size();
59    }
60
61    size_t low = 0;
62    size_t high = preSortedVector->size();
63    while (low != high) {
64      size_t lowHighSum = low + high;
65      unsigned int mid = (lowHighSum) / 2;
66      if (preSortedVector->at(mid) <= target) {
67        low = mid + 1;
68      }
69      else {
70        high = mid;
71      }
72    }
73    return high;
74  }
75
76  // creates a new temporary file and returns the path to it
77  string CigarCoilUtilities::createTemporaryFile() {
78  #ifdef unix
79    string temporaryCompressedFileName = tmpnam(nullptr);
80
81    ofstream file;
82    file.open(temporaryCompressedFileName.c_str(), ios::out);
83    file.close();
84    return temporaryCompressedFileName;
85  #else
86    char temporaryCompressedFileName[L_tmpnam_s];
87    tmpnam_s(temporaryCompressedFileName, L_tmpnam_s);
88
89    ofstream file;
90    file.open(temporaryCompressedFileName, ios::out);
91    file.close();
92    return string(temporaryCompressedFileName);
93  #endif // unix
94  }
95
96  string CigarCoilUtilities::getDataAtFilePosition(unsigned long long pos,
        unsigned short length, ifstream *fileStream) {
97
98
99
100   char *buffer = new char[length];
101
102   fileStream->seekg(pos, fileStream->beg);
103   fileStream->read(buffer, length);
104
```

```
105   string result = string(buffer, length);
106
107   delete[] buffer;
108
109   return result;
110 }
111
112 // constructs a minimum spanning tree with Prim's algorithm and returns the
      parent array
113 unsigned int* CigarCoilUtilities::PrimMST(SimilarityGraph* graph)
114 {
115   // Get the number of vertices in graph
116   unsigned int numberOfVertices = graph->getVectorSize();
117   // create parent array to store constructed MST
118   unsigned int *parent = new unsigned int[numberOfVertices];
119   // Key values used to pick minimum weight edge in cut
120   unsigned int *key = new unsigned int[numberOfVertices];
121
122   // minHeap represents set of edges
123   MinHeap minHeap = MinHeap(numberOfVertices);
124
125   // Initialize min heap with all vertices. Key value of
126   // all vertices (except 0th vertex) is initially infinite
127   for (unsigned int v = 1; v < numberOfVertices; ++v) {
128     parent[v] = -1;
129     key[v] = UINT_MAX;
130     minHeap.addNewMinHeapNode(v, key[v]);
131     minHeap.pos[v] = v;
132   }
133
134   // Make key value of 0th vertex as 0 so that it
135   // is extracted first
136   key[0] = 0;
137   minHeap.addNewMinHeapNode(0, key[0]);
138   minHeap.pos[0] = 0;
139
140   // Initially size of min heap is equal to V
141   minHeap.size = numberOfVertices;
142
143   // In the following loop, min heap contains all nodes
144   // not yet added to MST.
145   while (!minHeap.isEmpty()) {
146     // Extract the vertex with minimum key value
147     struct MinHeapNode* minHeapNode = minHeap.extractMin();
148
149     // Store the extracted vertex number
150     int u = minHeapNode->v;
151
152     // Traverse through all adjacent vertices of u (the extracted
153     // vertex) and update their key values
154     struct AdjListNode* pCrawl = graph->adjList[u].head;
155     while (pCrawl != NULL) {
156       int v = pCrawl->dest;
157
158       // If v is not yet included in MST and weight of u-v is
159       // less than key value of v, then update key value and
160       // parent of v
161       if (minHeap.isInMinHeap(v) && pCrawl->weight < key[v]) {
```

```
162           key[v] = pCrawl->weight;
163           parent[v] = u;
164           minHeap.decreaseKey(v, key[v]);
165         }
166         pCrawl = pCrawl->next;
167       }
168     }
169
170     return parent;
171 }
172
173 // returns a new parent array that minimizes the height of the provided
      parent array
174 unsigned int* CigarCoilUtilities::getMinimumHeightTree(const unsigned int*
      parentArray, size_t numberOfElements, unsigned int *root)
175 {
176     // create FIFO queue
177     queue<unsigned int> q;
178
179     unsigned int *degrees = new unsigned int[numberOfElements];
180     unsigned int *result = new unsigned int[numberOfElements];
181
182     // track which nodes are adjacent to each other
183     vector <vector<unsigned int>> adj = vector<vector<unsigned int>>();
184
185     // initialize degrees to 1
186     for (size_t i = 0; i < numberOfElements; i++) {
187       adj.push_back(vector<unsigned int>());
188       degrees[i] = 1;
189       adj[i] = vector<unsigned int>();
190     }
191
192     // determine degrees by adding up number of children for each node and
        populate adjacency vector for trees
193     for (size_t i = 1; i < numberOfElements; i++) {
194       degrees[parentArray[i]] += 1;
195       adj[i].push_back(parentArray[i]);
196       adj[parentArray[i]].push_back(i);
197     }
198
199     //  enqueue leaf nodes
200     for (unsigned int i = 0; i < numberOfElements; i++) {
201       if (degrees[i] == 1) {
202         q.push(i);
203       }
204     }
205
206     unsigned int numberOfVerticesRemaining = numberOfElements;
207
208     //  loop until total vertex remains less than 2
209     while (numberOfVerticesRemaining > 2)
210     {
211       for (size_t i = 0; i < q.size(); i++)
212       {
213         unsigned int t = q.front();
214         q.pop();
215         numberOfVerticesRemaining--;
216
```

```
217       // for each neighbour, decrease its degree and
218       // if it become leaf, insert into queue
219       for (auto j = adj[t].begin(); j != adj[t].end(); j++)
220       {
221         degrees[*j]--;
222         if (degrees[*j] == 1)
223           q.push(*j);
224       }
225     }
226   }
227
228   delete[] degrees;
229
230   //  copying the result from queue to result vector
231   vector<unsigned int> res;
232   while (!q.empty())
233   {
234     res.push_back(q.front());
235     q.pop();
236   }
237
238   // get minimum height root for the tree and set its parent in the new
      parent array to itself
239   unsigned minimumHeightRoot = res[0];
240   result[minimumHeightRoot] = minimumHeightRoot;
241   *root = minimumHeightRoot;
242
243   // empty queue
244   while (!q.empty()) {
245     q.pop();
246   }
247
248   q.push(minimumHeightRoot);
249
250   while (!q.empty()) {
251     unsigned int currentParentVertex = q.front();
252     q.pop();
253
254     for (size_t c = 0; c < adj[currentParentVertex].size(); c++) {
255       unsigned int currentChild = adj[currentParentVertex][c];
256       if (!adj[currentChild].empty()) { // checks if this child has already
      been added
257         // updates resulting parent array and pushes this child into the
      queue
258         result[currentChild] = currentParentVertex;
259         q.push(currentChild);
260       }
261     }
262
263     // no longer need this set of adjacent vertices - also marks that this
      vertex has been handled
264     adj[currentParentVertex].clear();
265
266   }
267
268   // returns updated parent array
269   return result;
270 }
```

```
271
272
273 char CigarCoilUtilities::greatestOfFour(double val1, double val2, double
    val3, double val4) {
274   if (val1 >= val2) {
275     if (val1 >= val3) {
276       if (val1 >= val4) {
277         return 1;
278       }
279       else {
280         return 4;
281       }
282     }
283     else {
284       if (val3 >= val4) {
285         return 3;
286       }
287       else {
288         return 4;
289       }
290     }
291   }
292   else {
293     if (val2 >= val3) {
294       if (val2 >= val4) {
295         return 2;
296       }
297       else {
298         return 4;
299       }
300     }
301     else {
302       if (val3 >= val4) {
303         return 3;
304       }
305       else {
306         return 4;
307       }
308     }
309   }
310 }
311
312 // encode file using zpaq
313 string CigarCoilUtilities::encodeZpaq(const char *sourceFile) {
314   string encodedFileName = CigarCoilUtilities::createTemporaryFile();
315   ifstream inputFileStream(sourceFile, ios_base::binary);
316   ofstream outputFileStream(encodedFileName.c_str(), ios_base::binary);
317   ZpaqReader reader = ZpaqReader(&inputFileStream);
318   ZpaqWriter writer = ZpaqWriter(&outputFileStream);
319
320   // call zpaq compressor with method 5 (slower with best compression)
321   libzpaq::compress(&reader, &writer, "5");
322
323   inputFileStream.close();
324   outputFileStream.close();
325
326   return encodedFileName;
327 }
```

```
328
329 // decompresses zpaq-compressed file
330 string CigarCoilUtilities::decodeZpaq(const char *compressedZpaqFile) {
331   string decompressedFile = CigarCoilUtilities::createTemporaryFile();
332   ifstream inputFileStream(compressedZpaqFile, ios_base::binary);
333   ofstream outputFileStream(decompressedFile.c_str(), ios_base::binary);
334   ZpaqReader reader = ZpaqReader(&inputFileStream);
335   ZpaqWriter writer = ZpaqWriter(&outputFileStream);
336
337   libzpaq::decompress(&reader, &writer);
338
339   inputFileStream.close();
340   outputFileStream.close();
341
342   return decompressedFile;
343 }
344
345 // provide location of pre-existing file
346 ZpaqReader::ZpaqReader(std::ifstream* fileStream) {
347   _fileStream = fileStream;
348 }
349
350 int ZpaqReader::get() {
351
352   if (_fileStream->eof()) {
353     return -1;
354   }
355   else {
356     return _fileStream->get();
357   }
358 }
359
360 int ZpaqReader::read(char *buf, int n) {
361   _fileStream->read(buf, n);
362   return _fileStream->gcount();
363 }
364
365 // provide location of pre-existing file
366 ZpaqWriter::ZpaqWriter(std::ofstream *fileStream) {
367   _fileStream = fileStream;
368 }
369
370 void ZpaqWriter::put(int c) {
371   _fileStream->put(c);
372 }
373
374 void ZpaqWriter::write(const char* buf, int c) {
375   _fileStream->write(buf, c);
376 }
```

Appendix B

Read

B.1 Header File

```
//
//  Author:  Addison Womack
//  Class: Read
//
//  Purpose: This class represents
//  a read's sequencing data
//
////////////////////////////////////////////////

#ifndef READ_H
#define READ_H

#include <iostream>
#include <string>
#include <set>
#include <vector>
#include <algorithm>
#include <iterator>
#include <stack>
#include <math.h>
#include "CigarOperation.h"

using namespace std;

class Read {

private:

  // location in origin file of this sequence
  unsigned long long _filePos;
  // length of the sequence
  unsigned short _sequenceLength;
  // returns the magnitude of the partitions object, treating its indices
    as dimensions of a euclidean vector
  double getMagnitudeOfBasesVector() const;


public:

  vector<unsigned char> partitions;
```

```
40
41    // Default Constructor
42    Read();
43
44    // Initializer Constructor
45    Read(string sequence, unsigned long long filePos, unsigned short
        readLengthForPartitionsCap, unsigned short partitionsCap);
46
47    // Copy Constructor
48    Read(const Read & read);
49
50    // Destructor
51    ~Read();
52
53    // Getter that returns the value of this Read
54    unsigned long long getSequencePos();
55
56    unsigned short getSequenceLength();
57
58    // gets the difference between this read and another by comparing
        magnitude of partition vector
59    double getMagnitudeOfDifferenceOfTwoReads(Read *comparisonRead);
60
61    // gets the difference between this read and another by comparing the
        angle between partition vectors
62    double getAngleBetweenTwoReads(const Read *comparisonRead);
63
64    // finds partition values for a given string
65    static vector<unsigned char> populatePartitionValues(string *sequence,
        size_t readLengthForPartitionsCap, unsigned short partitionsCap);
66
67    // Overloaded assignment operator
68    Read & operator= (const Read & read);
69 };
70
71
72 #endif
```

B.2 Definitions

```
1  #include "Read.h"
2
3  /* Default Constructor for Read intitalizes private fields to NULL */
4  Read::Read() {
5    partitions = vector<unsigned char>();
6    _filePos = 0;
7    _sequenceLength = 0;
8  }
9
10 // Initializer Constructor
11 Read::Read(string sequence, unsigned long long filePos, unsigned short
       readLengthForPartitionsCap, unsigned short partitionsCap) {
12   partitions = populatePartitionValues(&sequence,
       readLengthForPartitionsCap, partitionsCap);
13   _filePos = filePos;
14   _sequenceLength = sequence.length();
15 }
```

```
16
17 /* Copy Constructor that creates a Read from another read */
18 Read::Read(const Read& userRead) {
19   _filePos = userRead._filePos;
20   partitions = userRead.partitions;
21   _sequenceLength = userRead._sequenceLength;
22 }
23
24 // populates the partitions of the current read based on the provided
     sequence
25 vector<unsigned char> Read::populatePartitionValues(string *sequence,
     size_t readLengthForPartitionsCap, unsigned short partitionSize) {
26   vector<unsigned char> partitions = vector<unsigned char>();
27
28   unsigned char currentPartitionValueA = (sequence->at(0) == 'A') ? 1 : 0;
29   unsigned char currentPartitionValueC = (sequence->at(0) == 'C') ? 1 : 0;
30   unsigned char currentPartitionValueG = (sequence->at(0) == 'G') ? 1 : 0;
31   unsigned char currentPartitionValueT = (sequence->at(0) == 'T') ? 1 : 0;
32
33   size_t stoppingPoint = min(sequence->size(), readLengthForPartitionsCap);
34
35   for (size_t i = 1; i < stoppingPoint; i++) {
36     if (i % partitionSize == 0) {
37       partitions.push_back(currentPartitionValueA);
38       partitions.push_back(currentPartitionValueC);
39       partitions.push_back(currentPartitionValueG);
40       partitions.push_back(currentPartitionValueT);
41       currentPartitionValueA = 0;
42       currentPartitionValueC = 0;
43       currentPartitionValueG = 0;
44       currentPartitionValueT = 0;
45     }
46     switch (sequence->at(i)) {
47     case 'A':
48       currentPartitionValueA++;
49       break;
50     case 'C':
51       currentPartitionValueC++;
52       break;
53     case 'G':
54       currentPartitionValueG++;
55       break;
56     case 'T':
57       currentPartitionValueT++;
58     }
59   }
60
61   return partitions;
62 }
63
64 // Gets the value of this Read
65 unsigned long long Read::getSequencePos() {
66   return _filePos;
67 }
68
69 unsigned short Read::getSequenceLength() {
70   return _sequenceLength;
71 }
```

```
72
73 double Read::getMagnitudeOfBasesVector() const {
74
75   unsigned int valueToBeSquareRooted = 0;
76   for (size_t i = 0; i < partitions.size(); i++) {
77     valueToBeSquareRooted += partitions[i] * partitions[i];
78   }
79
80   return sqrt(valueToBeSquareRooted);
81 }
82
83 double Read::getMagnitudeOfDifferenceOfTwoReads(Read *comparisonRead) {
84
85   unsigned int valueToBeSquareRooted = 0;
86   size_t smallerNumberOfDimensions =
87     (partitions.size() < comparisonRead->partitions.size()) ?
88     partitions.size() :
89     comparisonRead->partitions.size();
90
91   for (size_t i = 0; i < smallerNumberOfDimensions; i++) {
92     short difference = partitions[i] - comparisonRead->partitions[i];
93     valueToBeSquareRooted += difference * difference;
94   }
95
96   return sqrt(valueToBeSquareRooted);
97
98 }
99
100 // returns angle between two reads in degrees
101 double Read::getAngleBetweenTwoReads(const Read *comparisonRead) {
102
103   unsigned int innerProduct = 0;
104   size_t smallerNumberOfDimensions =
105     (partitions.size() < comparisonRead->partitions.size()) ?
106     partitions.size() :
107     comparisonRead->partitions.size();
108
109   for (size_t i = 0; i < smallerNumberOfDimensions; i++) {
110     innerProduct += partitions[i] * comparisonRead->partitions[i];
111   }
112
113   double magnitudeOfThisVector = getMagnitudeOfBasesVector();
114   double magnitudeOfComparisonVector = comparisonRead->
      getMagnitudeOfBasesVector();
115
116   // min is used to bound error due to precision (e.g. two equivalent reads
        could end up with acos(1.000000002) leading to NAN)
117   return acos(min(innerProduct / (magnitudeOfThisVector *
      magnitudeOfComparisonVector),1.0));
118
119 }
120
121 /* Destructor for Read */
122 Read::~Read() {
123   partitions.clear();
124 }
125
126 /* Overloaded assignment operator */
```

```
127 Read& Read:: operator= (const Read& read) {
128   _filePos = read._filePos;
129   _sequenceLength = read._sequenceLength;
130   partitions = read.partitions;
131   return *this;
132 }
```

Appendix C

CIGAR Operation

C.1 Header File

```
#ifndef CIGAR_OPERATION_H
#define CIGAR_OPERATION_H

#include <string>
#include <stdio.h>

#define MatchOperation 0;
#define SubstitutionOperation 1;
#define InsertionOperation 2;
#define DeletionOperation 3;

#define MAXIMUM_MATCH_OR_DELETION_SIZE = 4095;

const unsigned char PositionEmpty = 0;
const unsigned char ValueAdenine  = 1;
const unsigned char ValueCytosine = 2;
const unsigned char ValueThymine  = 3;
const unsigned char ValueGuanine  = 4;
const unsigned char ValueNotKnown = 5;
const unsigned char ValueUracil = 6;
const unsigned char ValueSpace = 7;

static const unsigned int maxNumberOfCharactersPerInsertion = 4;

// define 2-byte struct
struct Operation {
  unsigned short reserved : 1;
  unsigned short operationType : 3;
  unsigned short value3 : 3;
  unsigned short value2 : 3;
  unsigned short value1 : 3;
  unsigned short value0 : 3;

  unsigned short getNumericValue() {

    return value3 * 512 +
      value2 * 64 +
      value1 * 8 +
      value0;
  }
```

```
};

// This class represents a single CIGAR operation
class CigarOperation {
private:
  // 2-byte structure
  Operation operation;
public:
  // default constructor
  CigarOperation();

  // substitution/insertion constructor
  CigarOperation(std::string s, bool isSub);
  // match/deletion constructor
  CigarOperation(unsigned int length, bool isMatch);

  // construct from struct
  CigarOperation(Operation operation);

  // construct from two bytes
  CigarOperation(unsigned char byteArr[2]);

  // returns the two-bytes that represent this struct
  char* GetBytes();

  // returns the internal 2-byte struct
  Operation getOperation();

  // returns whether or not this is a match operation
  bool isMatch();

  // returns whether or not this is an insertion operation
  bool isInsertion();

  // returns whether or not this is a substitution operation
  bool isSubstitution();

  // returns whether or not this is a deletion operation
  bool isDeletion();

  // returns the insertion or substitution values
  std::string getValueString();
  unsigned short getValueNumeric();

  // decodes a 3-bit value
  static char mapValueToCharacter(unsigned char val) {
    switch (val) {
    case PositionEmpty:
      return '\0';
    case ValueAdenine:
      return 'A';
    case ValueCytosine:
      return 'C';
    case ValueThymine:
      return 'T';
    case ValueGuanine:
      return 'G';
    case ValueNotKnown:
```

```
99        return 'N';
100     case ValueUracil:
101       return 'U';
102     case ValueSpace:
103       return '-';
104     }
105   }
106
107   // encodes a 3-bit value
108   static unsigned char mapCharacterToValue(char character) {
109     switch (character) {
110     case '\0':
111       return PositionEmpty;
112     case 'A':
113       return ValueAdenine;
114     case 'C':
115       return ValueCytosine;
116     case 'T':
117       return ValueThymine;
118     case 'G':
119       return ValueGuanine;
120     case 'U':
121       return ValueUracil;
122     case '.':
123     case '-':
124       return ValueSpace;
125     default:
126       // normally N
127       return ValueNotKnown;
128     }
129   }
130 };
131
132
133
134
135 #endif // !CIGAR_OPERATION_H
```

C.2 Definitions

```
1  #include "CigarOperation.h"
2
3  // substitution/insertion constructor
4  CigarOperation::CigarOperation(std::string s, bool isSub) {
5    Operation op;
6
7    if (isSub) {
8      op.operationType = SubstitutionOperation;
9    }
10   else {
11     op.operationType = InsertionOperation;
12   }
13
14   // only supports up to length 4 per mismatch operation
15
16   op.value0 = 0;
17   op.value1 = 0;
```

```
18    op.value2 = 0;
19    op.value3 = 0;
20    op.reserved = 1;
21
22    switch (s.length()) {
23    case 4:
24      op.value3 = CigarOperation::mapCharacterToValue(s.at(3));
25    case 3:
26      op.value2 = CigarOperation::mapCharacterToValue(s.at(2));
27    case 2:
28      op.value1 = CigarOperation::mapCharacterToValue(s.at(1));
29    case 1:
30      op.value0 = CigarOperation::mapCharacterToValue(s.at(0));
31      break;
32    default:
33      printf("undefined behavior...\n");
34    }
35
36    operation = op;
37
38  }
39
40  CigarOperation::CigarOperation() {
41    Operation op;
42    op.operationType = MatchOperation;
43    op.reserved = 0;
44    op.value0 = 0;
45    op.value1 = 0;
46    op.value2 = 0;
47    op.value3 = 0;
48    operation = op;
49  }
50
51  // match/deletion constructor
52  CigarOperation::CigarOperation(unsigned int length, bool isMatch) {
53    Operation op;
54    op.reserved = 1;
55    if (isMatch) {
56      op.operationType = MatchOperation;
57    }
58    else {
59      op.operationType = DeletionOperation;
60    }
61    op.value3 = length / (512);
62    unsigned short remainder = length % (512);
63
64    op.value2 = remainder / (64);
65    remainder = remainder % (64);
66
67    op.value1 = remainder / (8);
68
69    remainder = remainder % (8);
70
71    op.value0 = remainder;
72
73    operation = op;
74  }
75
```

```
76 // construct from two bytes
77 CigarOperation::CigarOperation(unsigned char byteArr[2]) {
78   Operation* op = (Operation*) byteArr;
79   operation = *op;
80 }
81
82 CigarOperation::CigarOperation(Operation op) {
83   operation = op;
84 }
85
86 // return underlying 2-byte structure
87 Operation CigarOperation::getOperation() {
88   return operation;
89 }
90
91 // is this a match operation
92 bool CigarOperation::isMatch() {
93   return operation.operationType == MatchOperation;
94 }
95
96 // is this a insertion opeartion
97 bool CigarOperation::isInsertion() {
98   return operation.operationType == InsertionOperation;
99 }
100
101 // is this a substitution operation
102 bool CigarOperation::isSubstitution() {
103   return operation.operationType == SubstitutionOperation;
104 }
105
106 // is this a deletion operation
107 bool CigarOperation::isDeletion() {
108   return operation.operationType == DeletionOperation;
109 }
110
111 // get string corresponding to  last 12 bits of the 2-byte struct
112 std::string CigarOperation::getValueString() {
113   std::string result = "";
114   char currentCharacter = mapValueToCharacter(operation.value0);
115   result += currentCharacter;
116   if (currentCharacter == '\0')
117     return result; // this should never happen
118   currentCharacter = mapValueToCharacter(operation.value1);
119   if (currentCharacter == '\0')
120     return result;
121   result += currentCharacter;
122   currentCharacter = mapValueToCharacter(operation.value2);
123   if (currentCharacter == '\0')
124     return result;
125   result += currentCharacter;
126   currentCharacter = mapValueToCharacter(operation.value3);
127   if (currentCharacter == '\0')
128     return result;
129   result += currentCharacter;
130   return result;
131 }
132
133 // returns the value corresponding to the last 12 bits of the 2-byte struct
```

```
134 unsigned short CigarOperation::getValueNumeric() {
135   return operation.getNumericValue();
136 }
137
138 // converts the underlying two-byte struct to an array of characters
139 char* CigarOperation::GetBytes() {
140   return reinterpret_cast<char *>(&operation);
141 }
```

Appendix D

Similarity Graph

D.1 Header File

```
#ifndef SIMILARITY_GRAPH_H
#define SIMILARITY_GRAPH_H

#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include "Read.h"
#include <vector>

// A structure to represent a node in adjacency list
struct AdjListNode {
  unsigned int dest;
  unsigned char weight;
  struct AdjListNode* next;
};

// A structure to represent an adjacency list
struct AdjList {
  struct AdjListNode* head; // pointer to head node of list
  Read read;
};

class SimilarityGraph {
private:
  AdjListNode* addNewAdjListNode(unsigned int dest, unsigned char weight);
public:
  // size of array
  unsigned int V;

  unsigned int getVectorSize();

  // Destructor
  ~SimilarityGraph();

  // formerly called array
  //struct AdjList* adjList;
  vector<AdjList> adjList;
  void addRead(Read read);
  void addEdge(unsigned int src, unsigned int dest, unsigned char weight);
  Read getReadAt(unsigned int i);
```

```
41
42    void clearReadPartitionInfo();
43
44    SimilarityGraph();
45 };
46
47 #endif
```

D.2 Definitions

```
1  #include "SimilarityGraph.h"
2
3  // initializes graph
4  SimilarityGraph::SimilarityGraph() {
5    adjList = vector<AdjList>();
6  }
7
8  unsigned int SimilarityGraph::getVectorSize() {
9    return adjList.size();
10 }
11
12
13 Read SimilarityGraph::getReadAt(unsigned int i) {
14   return adjList.at(i).read;
15 }
16
17
18 void SimilarityGraph::addRead(Read read) {
19   AdjList adjList = AdjList();
20   adjList.head = NULL;
21   adjList.read = read;
22   this->adjList.push_back(adjList);
23 }
24
25 // A utility function to create a new adjacency list node
26 struct AdjListNode* SimilarityGraph::addNewAdjListNode(unsigned int dest,
       unsigned char weight)
27 {
28   struct AdjListNode* newNode = (struct AdjListNode*)malloc(sizeof(struct
       AdjListNode));
29   newNode->dest = dest;
30   newNode->weight = weight;
31   newNode->next = NULL;
32   return newNode;
33 }
34
35 // Adds an edge to an undirected graph
36 void SimilarityGraph::addEdge(unsigned int src, unsigned int dest, unsigned
       char weight)
37 {
38   // Add an edge from src to dest.  A new node is added to the adjacency
39   // list of src.  The node is added at the begining
40   struct AdjListNode* newNode = addNewAdjListNode(dest, weight);
41   newNode->next = adjList[src].head;
42   adjList[src].head = newNode;
43
44   // Since graph is undirected, add an edge from dest to src also
```

```
45    newNode = addNewAdjListNode(src, weight);
46    newNode->next = adjList[dest].head;
47    adjList[dest].head = newNode;
48  }
49
50  void SimilarityGraph::clearReadPartitionInfo() {
51    for (size_t i = 0; i < adjList.size(); i++) {
52      adjList[i].read.partitions.clear();
53    }
54  }
55
56  /* Destructor for Similarity Graph */
57  SimilarityGraph::~SimilarityGraph() {
58    for (size_t i = 0; i < adjList.size(); i++) {
59      /* deref head_ref to get the real head */
60      struct AdjListNode* current = adjList[i].head;
61      struct AdjListNode* next;
62
63      while (current != NULL)
64      {
65        next = current->next;
66        free(current);
67        current = next;
68      }
69
70      /* deref head_ref to affect the real head back
71      in the caller. */
72      adjList[i].head = NULL;
73
74    }
75  }
```

Appendix E

Hash Bucket Index

E.1 Header File

```
#ifndef HASH_BUCKET_INDEX_H
#define HASH_BUCKET_INDEX_H

#include <vector>

using namespace std;

// structure used for read compartmentalization heuristic
class HashBucketIndex {
private:
  // size of partitions (Delta)
  unsigned char partitionSize;

  // read lengths to base partition size on
  unsigned short readLength;

  // number of hash buckets to create
  unsigned short numberOfHashBuckets;

  // underlying structure is a 3-dimensional matrix of values
  vector<vector<vector<unsigned int>>> hashBuckets;

public:

  HashBucketIndex(unsigned char partitionSize, unsigned short readLength);
  void insert(unsigned short hashBucketNumber, unsigned char partitionValue
    , unsigned int readNumber);
  const vector<unsigned int>* at(unsigned short hashBucketNumber, unsigned
    char partitionValue) const;
  ~HashBucketIndex();
};

#endif
```

E.2 Definitions

```
#include "HashBucketIndex.h"

```

```
3  HashBucketIndex::HashBucketIndex(unsigned char partitionSize, unsigned
       short readLength) {
4    this->partitionSize;
5    this->readLength;
6
7    // 1 bucket for each A C T G
8    numberOfHashBuckets = (readLength / partitionSize) * 4;
9
10   hashBuckets = vector<vector<vector<unsigned int>>>();
11
12   for (size_t c = 0; c < numberOfHashBuckets; c++) {
13     hashBuckets.push_back(vector<vector<unsigned int>>());
14     for (size_t p = 0; p < (partitionSize + 1); p++) {
15       hashBuckets[c].push_back(vector<unsigned int>());
16     }
17   }
18
19 }
20
21 const vector<unsigned int>* HashBucketIndex::at(unsigned short
       hashBucketNumber, unsigned char partitionValue) const {
22   return &hashBuckets.at(hashBucketNumber).at(partitionValue);
23 }
24
25 void HashBucketIndex::insert(unsigned short hashBucketNumber, unsigned char
       partitionValue, unsigned int readNumber) {
26   hashBuckets[hashBucketNumber][partitionValue].push_back(readNumber);
27 }
28
29 HashBucketIndex::~HashBucketIndex() {
30   hashBuckets.clear();
31 }
```

Appendix F

Min Heap

F.1 Header File

```
////////////////////////////////////////////////////////////////
// This class provides a min heap data structure that
// is used primarily in computation of a minimum spanning tree

#ifndef MIN_HEAP_H
#define MIN_HEAP_H

#include <limits.h>
#include <stdio.h>
#include <stdlib.h>

// Structure to represent a min heap node
struct MinHeapNode {
  unsigned int v;
  unsigned int key;
};

class MinHeap {
private:
public:
  // Number of heap nodes stored in this structure
  unsigned int size;
  // maximum capacity of this structure
  unsigned int capacity;
  // This is needed for decreaseKey()
  unsigned int* pos;
  // 2-d array of nodes stored in this min-heap structure
  struct MinHeapNode** array;

  MinHeap(unsigned int capacity);
  void swapMinHeapNode(struct MinHeapNode** a, struct MinHeapNode** b);
  void addNewMinHeapNode(unsigned int v, unsigned int key);
  void minHeapify(unsigned int idx);
  bool isEmpty();
  struct MinHeapNode* extractMin();

  void decreaseKey(unsigned int v, unsigned int key);

  bool isInMinHeap(unsigned int v);
};
```

```
41
42 #endif
```

F.2 Definitions

```
1  #include "MinHeap.h"
2
3  MinHeap::MinHeap(unsigned int capacity) {
4    this->pos = (unsigned int*)malloc(capacity * sizeof(unsigned int));
5    this->size = 0;
6    this->capacity = capacity;
7    this->array = (struct MinHeapNode**)malloc(capacity * sizeof(struct
       MinHeapNode*));
8  }
9
10 // A utility function to swap two nodes of min heap. Needed for min heapify
11 void MinHeap::swapMinHeapNode(struct MinHeapNode** a, struct MinHeapNode**
       b)
12 {
13   struct MinHeapNode* t = *a;
14   *a = *b;
15   *b = t;
16 }
17
18 // A utility function to create a new Min Heap Node
19 void MinHeap::addNewMinHeapNode(unsigned int v, unsigned int key)
20 {
21   struct MinHeapNode* minHeapNode = (struct MinHeapNode*)malloc(sizeof(
       struct MinHeapNode));
22   minHeapNode->v = v;
23   minHeapNode->key = key;
24
25   this->array[v] = minHeapNode;
26 }
27
28 // A utility function to check if the given minHeap is empty or not
29 bool MinHeap::isEmpty()
30 {
31   return this->size == 0;
32 }
33
34 // A utility function to check if a given vertex
35 // 'v' is in min heap or not
36 bool MinHeap::isInMinHeap(unsigned int v)
37 {
38   if (this->pos[v] < this->size)
39     return true;
40   return false;
41 }
42
43 // Function to decreasy key value of a given vertex v. This function
44 // uses pos[] of min heap to get the current index of node in min heap
45 void MinHeap::decreaseKey(unsigned int v, unsigned int key)
46 {
47   // Get the index of v in  heap array
48   unsigned int i = this->pos[v];
49
```

```
50    // Get the node and update its key value
51    this->array[i]->key = key;
52
53    // Travel up while the complete tree is not heapified.
54    // This is a O(Logn) loop
55    while (i && this->array[i]->key < this->array[(i - 1) / 2]->key) {
56      // Swap this node with its parent
57      this->pos[this->array[i]->v] = (i - 1) / 2;
58      this->pos[this->array[(i - 1) / 2]->v] = i;
59      swapMinHeapNode(&this->array[i], &this->array[(i - 1) / 2]);
60
61      // move to parent index
62      i = (i - 1) / 2;
63    }
64  }
65
66
67  // Standard function to extract minimum node from heap
68  struct MinHeapNode* MinHeap::extractMin()
69  {
70    if (isEmpty())
71      return NULL;
72
73    // Store the root node
74    struct MinHeapNode* root = this->array[0];
75
76    // Replace root node with last node
77    struct MinHeapNode* lastNode = this->array[this->size - 1];
78    this->array[0] = lastNode;
79
80    // Update position of last node
81    this->pos[root->v] = this->size - 1;
82    this->pos[lastNode->v] = 0;
83
84    // Reduce heap size and heapify root
85    --this->size;
86    minHeapify(0);
87
88    return root;
89  }
90
91  // A standard function to heapify at given idx
92  // This function also updates position of nodes when they are swapped.
93  // Position is needed for decreaseKey()
94  void MinHeap::minHeapify(unsigned int idx)
95  {
96    unsigned int smallest, left, right;
97    smallest = idx;
98    left = 2 * idx + 1;
99    right = 2 * idx + 2;
100
101   if (left < this->size && this->array[left]->key < this->array[smallest]->
      key)
102     smallest = left;
103
104   if (right < this->size && this->array[right]->key < this->array[smallest
      ]->key)
105     smallest = right;
```

```
106
107   if (smallest != idx) {
108     // The nodes to be swapped in min heap
109     MinHeapNode* smallestNode = this->array[smallest];
110     MinHeapNode* idxNode = this->array[idx];
111
112     // Swap positions
113     this->pos[smallestNode->v] = idx;
114     this->pos[idxNode->v] = smallest;
115
116     // Swap nodes
117     swapMinHeapNode(&this->array[smallest], &this->array[idx]);
118
119     minHeapify(smallest);
120   }
121 }
```

Appendix G

Wagner Fischer Matrix

G.1 Header File

```
#ifndef WAGNER_FISCHER_H
#define WAGNER_FISCHER_H

#include <string>
#include <stack>
#include <vector>
#include "CigarOperation.h"
#include "DNAFileWrapper.h"

using namespace std;

// tracks dominant action for the given cell
enum cellType{
  notSetYet,
  match,
  deletion,
  insertion,
  substitution
};

// repreents single cell of matrix
struct cell {
  unsigned short value;
  unsigned short numberOfConsecutiveOperations;
  cellType type;
};

class WagnerFischerMatrix {
private:
  cell *arr;
  size_t width;
  size_t height;
  // string being transformed into target
  string origin;
  // string that the source is transforming into
  string target;

  void addMatchOperation(vector<CigarOperation> * operations, size_t size);
  void addDeletionOperation(vector<CigarOperation> * operations, size_t
    size);
```

```
40   void addInsertionOperations(vector<CigarOperation> *operations, string
       toBeInserted);
41   void addSubstitutionOperations(vector<CigarOperation> *operations, string
       toInsert);
42
43 public:
44
45   WagnerFischerMatrix(const string *originString, const string *
       targetString);
46   cell* at(size_t rowIndex, size_t columnIndex);
47
48   void set(size_t rowIndex, size_t columnIndex, unsigned short value,
       cellType type, unsigned short numberOfConsecutiveOperations);
49
50   string getCigar();
51
52   unsigned short getEditDistance();
53
54   // Destructor
55   ~WagnerFischerMatrix();
56 };
57
58 #endif // !WAGNER_FISCHER_H
```

G.2 Definitions

```
1  #include "WagnerFischerMatrix.h"
2
3  WagnerFischerMatrix::WagnerFischerMatrix(const string *originString, const
       string *targetString) {
4    // For all i and j, d[i,j] will hold the CIGAR size.
5
6    //let d be a 2 - d array with dimensions[0..m, 0..n]
7    width = originString->length() + 1;
8    height = targetString->length() + 1;
9
10   arr = new cell[width * height];
11   for (size_t i = 0; i < width * height; i++) {
12     arr[i].type = notSetYet;
13     arr[i].numberOfConsecutiveOperations = 0;
14     arr[i].value = 0;
15   }
16
17   origin = *originString;
18   target = *targetString;
19
20   // the top left corner corresponds to two empty strings
21   set(0, 0, 0, notSetYet, 0);
22
23   // the distance of any first string to an empty second string
24   // (transforming the string of the first i characters of s into
25   // the empty string requires i deletions)
26   for (size_t i = 1; i <= originString->length(); i++) {
27     // with the struct being used, reducing any string to an empty string
       requires only 1 operation
28     set(i, 0, 1, deletion, i);
29   }
```

```
30
31  // top row --- insertions only
32  for (size_t j = 1; j <= targetString->length(); j++) {
33    // with the struct being used, each insertion can insert up to 4
     characters at once
34    size_t value = ((j - 1) / 4) + 1;
35    set(0, j, value, insertion, j);
36  }
37
38  // apply dynamic algorithm to fill in the wagner-fischer matrix
39  for (size_t i = 1; i <= originString->length(); i++) {
40    for (size_t j = 1; j <= targetString->length(); j++) {
41      if ((originString->at(i - 1) == targetString->at(j - 1)) && (at(i -
     1, j - 1)->type == match)) {
42        // no new operation required: match already in progress
43        set(i, j, at(i - 1, j - 1)->value, match, at(i - 1, j - 1)->
     numberOfConsecutiveOperations + 1);
44
45      }
46      else if ((i >= 4) && (j >= 4) && (originString->at(i - 1) ==
     targetString->at(j - 1)) && (originString->substr(i - 4, 4) ==
     targetString->substr(j - 4, 4))) {
47        // create a new match operation on previous diagonal values
48        size_t value = at(i - 3, j - 3)->value + 1;
49        set(i, j, value, match, 4);
50        set(i - 1, j - 1, value, match, 3);
51        set(i - 2, j - 2, value, match, 2);
52        set(i - 3, j - 3, value - 1, match, 1);
53      }
54      else {
55        // operation is not a match operation
56
57        cellType previousType;
58
59        // get adjacent three cells
60        cell *above = at(i - 1, j);
61        cell *leftSide = at(i, j - 1);
62        cell *topLeftCorner = at(i - 1, j - 1);
63
64        // minimum operation
65        size_t deletionValue = (above->type == deletion) ? above->value :
     above->value + 1;
66        size_t insertionValue;
67        if (leftSide->type == insertion) {
68          bool needsANewStruct = leftSide->numberOfConsecutiveOperations %
     4 == 0;
69          insertionValue = needsANewStruct ? leftSide->value + 1 : leftSide
     ->value;
70        }
71        else {
72          insertionValue = leftSide->value + 1;
73        }
74        size_t substitutionValue;
75
76        if (topLeftCorner->type == substitution) {
77          bool needsANewStruct = topLeftCorner->
     numberOfConsecutiveOperations % 4 == 0;
78          substitutionValue = needsANewStruct ? topLeftCorner->value + 1 :
```

```
    topLeftCorner->value;
79        }
80        else {
81          substitutionValue = topLeftCorner->value + 1;
82        }
83
84        if (deletionValue <= substitutionValue) {
85          if (deletionValue <= insertionValue) {
86            size_t numberOfConsecutiveOperations = above->type == deletion
    ? above->numberOfConsecutiveOperations + 1 : 1;
87            set(i, j, deletionValue, deletion,
    numberOfConsecutiveOperations);
88          }
89          else {
90            size_t numberOfConsecutiveOperations = leftSide->type ==
    insertion ? leftSide->numberOfConsecutiveOperations + 1 : 1;
91            set(i, j, insertionValue, insertion,
    numberOfConsecutiveOperations);
92          }
93        }
94        else {
95          if (substitutionValue <= insertionValue) {
96            size_t numberOfConsecutiveOperations = topLeftCorner->type ==
    substitution ? topLeftCorner->numberOfConsecutiveOperations + 1 : 1;
97            set(i, j, substitutionValue, substitution,
    numberOfConsecutiveOperations);
98          }
99          else {
100           size_t numberOfConsecutiveOperations = leftSide->type ==
    insertion ? leftSide->numberOfConsecutiveOperations + 1 : 1;
101           set(i, j, insertionValue, insertion,
    numberOfConsecutiveOperations);
102         }
103       }
104     }
105   }
106 }
107 }
108
109 // get value at specified row and column
110 cell* WagnerFischerMatrix::at(size_t rowIndex, size_t columnIndex) {
111   return &arr[rowIndex + width * columnIndex];
112 }
113
114 // set value at specified row and column
115 void WagnerFischerMatrix::set(size_t rowIndex, size_t columnIndex, unsigned
     short value, cellType type, unsigned short
    numberOfConsecutiveOperations) {
116   arr[rowIndex + width * columnIndex].value = value;
117   arr[rowIndex + width * columnIndex].type = type;
118   arr[rowIndex + width * columnIndex].numberOfConsecutiveOperations =
    numberOfConsecutiveOperations;
119 }
120
121 string WagnerFischerMatrix::getCigar() {
122
123   size_t currentRow = origin.length();
124   size_t currentColumn = target.length();
```

```
125
126 stack<char> operationStack = stack<char>();
127 vector<CigarOperation> *operations = &vector<CigarOperation>();
128
129 string cigar = "";
130
131 bool isSub = false;
132 bool isInsert = false;
133
134 while ((currentColumn > 0) || (currentRow > 0)) {
135   // case if on left border of matrix
136   if (currentColumn < 1) {
137     operationStack.push('D');
138     currentRow--;
139   }
140   // case if on top border of matrix
141   else if (currentRow < 1) {
142     operationStack.push(target.at(currentColumn-- - 1));
143     operationStack.push('I');
144   }
145   else {
146     size_t numberOfConsecutiveOperations = at(currentRow, currentColumn)
 ->numberOfConsecutiveOperations;
147     switch (at(currentRow, currentColumn)->type) {
148     case match:
149
150       // if the match has less than 4, then it's more efficient to use
 substitution
151       if (numberOfConsecutiveOperations < 4) {
152         for (size_t i = 0; i < numberOfConsecutiveOperations; i++) {
153           operationStack.push(target.at(currentColumn-- - 1));
154           operationStack.push('S');
155         }
156         currentRow -= numberOfConsecutiveOperations;
157       }
158       else {
159
160         for (size_t i = 0; i < numberOfConsecutiveOperations; i++) {
161           operationStack.push('M');
162         }
163         currentRow -= numberOfConsecutiveOperations;
164         currentColumn -= numberOfConsecutiveOperations;
165       }
166       break;
167     case deletion:
168       for (size_t i = 0; i < numberOfConsecutiveOperations; i++) {
169         operationStack.push('D');
170       }
171       currentRow -= numberOfConsecutiveOperations;
172       break;
173     case substitution:
174       for (size_t i = 0; i < numberOfConsecutiveOperations; i++) {
175         operationStack.push(target.at(currentColumn-- - 1));
176         operationStack.push('S');
177       }
178       currentRow -= numberOfConsecutiveOperations;
179       break;
180     case insertion:
```

```
181         for (size_t i = 0; i < numberOfConsecutiveOperations; i++) {
182           operationStack.push(target.at(currentColumn-- - 1));
183           operationStack.push('I');
184         }
185         break;
186       }
187     }
188   }
189
190   bool isMatch = false;
191   bool isDelete = false;
192   bool isInsertion = false;
193   bool isSubstitution = false;
194   string currentString = "";
195   size_t matchOrDeletionLength = 0;
196
197   while (operationStack.size() > 0) {
198     char currentSymbol = operationStack.top();
199     operationStack.pop();
200
201     switch (currentSymbol) {
202     case 'M':
203       if (isMatch) {
204         matchOrDeletionLength++;
205       }
206       else if (isDelete) {
207         // add delete operation
208         addDeletionOperation(operations, matchOrDeletionLength);
209         // reset length
210         matchOrDeletionLength = 1;
211         isMatch = true;
212         isDelete = false;
213       }
214       else if (isSubstitution) {
215         // add sub with current string
216         if (currentString.length() > 0)
217           addSubstitutionOperations(operations, currentString);
218         currentString = "";
219         isMatch = true;
220         isSubstitution = false;
221         matchOrDeletionLength = 1;
222       }
223       else if (isInsertion) {
224         // add insertion with current string
225         if (currentString.length() > 0) addInsertionOperations(operations,
    currentString);
226         currentString = "";
227         isMatch = true;
228         isInsertion = false;
229         matchOrDeletionLength = 1;
230       }
231       else {
232         isMatch = true;
233         matchOrDeletionLength = 1;
234       }
235       break;
236     case 'D':
237
```

```
238       if (isMatch) {
239         // add match operation
240         addMatchOperation(operations, matchOrDeletionLength);
241         // reset length
242         matchOrDeletionLength = 1;
243         isDelete = true;
244         isMatch = false;
245       }
246       else if (isDelete) {
247         matchOrDeletionLength++;
248       }
249       else if (isSubstitution) {
250         // add sub with current string
251         if (currentString.length() > 0)
252           addSubstitutionOperations(operations, currentString);
253 
254         currentString = "";
255         isDelete = true;
256         isSubstitution = false;
257         matchOrDeletionLength = 1;
258       }
259       else if (isInsertion) {
260         // add insertion with current string
261         if (currentString.length() > 0) addInsertionOperations(operations,
    currentString);
262         currentString = "";
263         isDelete = true;
264         isInsertion = false;
265         matchOrDeletionLength = 1;
266       }
267       else {
268         isDelete = true;
269         matchOrDeletionLength = 1;
270       }
271       break;
272     default: // insertion/substitution
273 
274         // take off next operation since it tells us what the current
    symbol is for
275       bool isThisInsertion = 'I' == currentSymbol; // otherwise
    substitution
276       bool isThisSubstitution = 'S' == currentSymbol;
277 
278       if (isMatch) {
279         addMatchOperation(operations, matchOrDeletionLength);
280         matchOrDeletionLength = 0;
281         isMatch = false;
282       }
283       else if (isDelete) {
284         addDeletionOperation(operations, matchOrDeletionLength);
285         matchOrDeletionLength = 0;
286         isDelete = false;
287       }
288       else if (isThisInsertion && isSubstitution) {
289         // add sub with current string
290         if (currentString.length() > 0)
291           addSubstitutionOperations(operations, currentString);
292         isSubstitution = false;
```

```
293         currentString = "";
294       }
295       else if (isThisSubstitution && isInsertion) {
296         // add insertion with current string
297         if (currentString.length() > 0) addInsertionOperations(operations,
    currentString);
298         isInsertion = false;
299         currentString = "";
300       }
301
302       isInsertion = isThisInsertion;
303       isSubstitution = isThisSubstitution;
304
305       currentString += operationStack.top();
306       operationStack.pop();
307
308       isMatch = false;
309       isDelete = false;
310       matchOrDeletionLength = 0;
311     }
312
313   }
314
315   if (isMatch) {
316     addMatchOperation(operations, matchOrDeletionLength);
317
318   }
319   else if (isDelete) {
320     addDeletionOperation(operations, matchOrDeletionLength);
321   }
322
323   if (isSubstitution) {
324     // add sub with current string
325     if (currentString.length() > 0)
326       addSubstitutionOperations(operations, currentString);
327     currentString = "";
328   }
329   else if (isInsertion) {
330     // add insertion with current string
331     if (currentString.length() > 0) addInsertionOperations(operations,
    currentString);
332     currentString = "";
333   }
334
335   char * myBytes = new char[operations->size() * 2];
336
337   for (unsigned int j = 0; j < operations->size(); j++) {
338     char * currentOpBytes = operations->at(j).GetBytes();
339     myBytes[(2 * j)] = currentOpBytes[0];
340     myBytes[(2 * j) + 1] = currentOpBytes[1];
341   }
342
343   cigar = string(myBytes, operations->size() * 2);
344   delete[] myBytes;
345 #ifdef DEBUG
346   if (DNAFileWrapper::decodeChildSequenceRelativeToParent(&cigar, &origin)
    != target) {
347     printf("something is wrong...\n");
```

```
348   }
349 #endif
350
351   return cigar;
352 }
353
354 void WagnerFischerMatrix::addMatchOperation(vector<CigarOperation> *
      operations, size_t equivalentRegionSize) {
355   operations->push_back(CigarOperation(equivalentRegionSize, true));
356 }
357
358 void WagnerFischerMatrix::addDeletionOperation(vector<CigarOperation> *
      operations, size_t equivalentRegionSize) {
359   operations->push_back(CigarOperation(equivalentRegionSize, false));
360 }
361
362 void WagnerFischerMatrix::addSubstitutionOperations(vector<CigarOperation>
      *operations, string toInsert) {
363   size_t insertionSize = toInsert.size();
364   size_t insertionRemainder = insertionSize % 4;
365   unsigned short numberOfCompleteInsertions = insertionSize / 4; // integer
       division intended
366   for (unsigned short t = 0; t < numberOfCompleteInsertions; t++) {
367     string segment = toInsert.substr(t * 4, 4);
368     CigarOperation operation = CigarOperation(segment, true);
369     operations->push_back(operation);
370   }
371   if (insertionRemainder > 0)
372     operations->push_back(CigarOperation(toInsert.substr(
      numberOfCompleteInsertions * 4, insertionRemainder), true));
373 }
374
375 void WagnerFischerMatrix::addInsertionOperations(vector<CigarOperation> *
      operations, string toInsert) {
376   size_t insertionSize = toInsert.size();
377   size_t insertionRemainder = insertionSize % 4;
378   unsigned short numberOfCompleteInsertions = insertionSize / 4; // integer
       division intended
379   for (unsigned short t = 0; t < numberOfCompleteInsertions; t++) {
380     string segment = toInsert.substr(t * 4, 4);
381     CigarOperation operation = CigarOperation(segment, false);
382     operations->push_back(operation);
383   }
384   if (insertionRemainder > 0)
385     operations->push_back(CigarOperation(toInsert.substr(
      numberOfCompleteInsertions * 4, insertionRemainder), false));
386 }
387
388 unsigned short WagnerFischerMatrix::getEditDistance() {
389   return at(origin.length(), target.length())->value;
390 }
391
392 /* Destructor for matrix */
393 WagnerFischerMatrix::~WagnerFischerMatrix() {
394   delete[] arr;
395 }
```

Appendix H

DNA File Wrapper

H.1 Header File

```
#ifndef DNAFileWrapper_H
#define DNAFileWrapper_H

#include<fstream>
#include<string>
#include "Read.h"
#include<stdio.h>
#include<vector>
#include<stack>
#include<list>
#include "SimilarityGraph.h"
#include "CigarOperation.h"

#ifndef UINT32_MAX
#define UINT32_MAX __UINT32_MAX__
#endif

#ifdef unix
#include<pthread.h>
#include<map>
#include<sys/sysinfo.h>

struct argumentStruct {
  SimilarityGraph* graph;
  HashBucketIndex* indices;
  unsigned int startingPosition;
  unsigned int stoppingPosition;
  string fileName;
};

#else
#include <thread>
#define USING_THREAD
#include<unordered_map>
#define USING_UNORDERED_MAP
#endif


#include <algorithm>
#include <iterator>
```

```
#include <ctime>
#include <sstream>
#include "WagnerFischerMatrix.h"
#include "CigarCoilUtilities.h"
#include "HashBucketIndex.h"
#include "DecodedReads.h"

enum DNAFileType {
  FASTQ,
  FASTA,
  SAM,
  CIGARCOIL
};

static const char* cigarFileMarker = "CGRC";
static const size_t maximumNumberOfReadsToApplyWagnerFischerTo = 2;
static const size_t halfMaximumNumberOfReadsToApplyWagnerFischerTo =
    maximumNumberOfReadsToApplyWagnerFischerTo / 2;
static const int numberOfStates = 13;
static const int numberOfActions = 21;

// learning rate
static const double ALPHA = 0.05;

// discount rate
static const double GAMMA = 0.01;

// probability
static const double EPSILON = 0.05;

enum predictiveCacheAccessPatterns {
  NEXT_SEQUENTIAL,
  NEXT_DELTA,
  PREV_SEQUENTIAL,
  PREV_DELTA,
  RANDOM_IN_DELTA
};

// Class that permits operations to be performed on FASTA/FASTQ/CIGARCoil
    files
class DNAFileWrapper {
private:
  // path to the file
  string myFileName;
  // type of file that it is
  DNAFileType fileType;
  // typical read length for this file
  unsigned short readLength;
  // position of the parent array in a CIGARCoil file
  streampos parentArrayLength;
  // element that is the root of the tree of a CIGARCoil file
  unsigned int root;

  // is this wrapped around a file with an underlying FASTQ structure?
  bool isFASTQ;

  // track positions of reads within a file
  vector<std::streampos> readPositions;
```

```
97
98  // track position in CIGARCoil file where zpaq file begins
99  unsigned long long idQualityStart;
100
101 // methdos for accessing the ith element of a particular type of file
102 string fastQFileAccess(size_t i);
103 string fastAFileAccess(size_t i);
104 string cigarCoilFileAccess(size_t i);
105 string cigarCoilFileAccess(size_t readNumber, ifstream *fileStream);
106 string cigarCoilFileAccess(size_t i, string childSequence);
107 string cigarCoilFileAccess(size_t i, string childSequence, ifstream *
     fileStream);
108
109 // encodes the given MST
110 static void encodeMSTAndWriteToFile(unsigned int root, unsigned int *
     parents, SimilarityGraph *similarityGraph, string outputFileName, string
      inputFileName, ifstream *fileStream);
111
112 // encodes file of sequence values and returns name of encoded file
113 static string encodeSequenceFields(const char *sequenceFile, unsigned
     short averageReadLength, bool isUsingWagnerFischerForEdgeWeigths);
114
115 // concatenates a set of files together
116 void concatenateFilesTogether(const string *files, size_t numberOfFiles,
     string resultFileName);
117
118 // concatenates a set of CIGAR encoded sequences together
119 static void concatenateCompressedSequencesTogether(const string *files,
     size_t numberOfFiles, string resultFileName);
120
121 // returns a parent array for the given CIGAR object
122 static vector<unsigned int> findParentArray(const char *fileName, size_t
     offset, streampos *finalPosition);
123
124 // computes K-means in a memory conservative manner
125 vector<unsigned int>* kmeans(const char * sequenceFileName, size_t
     numberOfClusters, size_t maximumNumberOfIterations, unsigned short
     sequenceLength);
126
127 // computes K-means with all sequences stored in memory
128 vector<unsigned int>* kmeans(const vector<string> * sequences, size_t
     numberOfClusters, size_t maximumNumberOfIterations, unsigned short
     sequenceLength);
129
130 // initialize a random set of strings
131 string* initializeCentroids(size_t numberOfCentroids, unsigned short
     sequenceLength);
132
133 // updates the centroids based on average of clustered elements
134 void recomputeCentroids(double ***runningAverageForEachReadPosition,
     size_t numberOfCentroids, string *centroids, unsigned short
     sequenceLength);
135
136 ////////////////////////////////////////
137 // Predictive Cache Private variables
138 bool isInitialized;
139 // The number of elements for the predictive cache to store
140 size_t numberOfElementsToCache;
```

```
141   // the first id of the cache window
142   unsigned int idOfFirstElementCached;
143   // the last id of the cache window
144   unsigned int idOfLastElementCached;
145   // set of cached elements
146   vector<string> cachedElements;
147
148   // Q-table of state action pairings
149   float stateActionPairs[numberOfStates][numberOfActions];
150
151   // make learning agent perform an action based on requested i
152   void qLearningPrediction(size_t requestedId);
153
154   // returns the best action for the given state
155   int getBestActionForAState(size_t state);
156
157   // fetches up to the specified number of elements into the cache
158   void fetchElementsForward(size_t start, size_t numberOfElements);
159
160   // fetches up to the specified number of elements into the cache
161   void fetchElementsBackward(size_t start, size_t numberOfElements);
162
163   // figures out which state corresponds to the element i
164   unsigned int determineState(unsigned int i);
165
166   // initializes the predictive cache
167   void initialize();
168
169   // returns the sequence at the specified element
170   std::string getElement(size_t element);
171
172 public:
173
174   // array for parentArray of reads
175   vector<unsigned int> parentArray;
176
177   // used for accessing encoded file
178   DecodedReads decodedReads;
179
180   // undoes a set of CIGAR operations based on the parent sequence
181   static string decodeChildSequenceRelativeToParent(const string *
      childCigar, const string *parentSequence);
182
183   // initializes this object with the file path to the file
184   DNAFileWrapper(const char *fileName);
185
186   // encodes the wrapped FASTA or FASTQ file
187   void encode(const char *encodedFileName, size_t numberOfClusters, size_t
      maximumNumberOfIterations, bool isBeingConservativeWithMainMemory, bool
      isUsingWagnerFischerForEdgeWeigths);
188
189   // decodes the wrapped CIGARCoil file
190   void decode(const char *decodedFileName);
191
192   // updates the read at the given sequence
193   void updateReadSequence(size_t i, string sequence);
194
195   // concatenates two CIGARCoil files together
```

```
196   DNAFileWrapper concatenate(DNAFileWrapper *childFile, string
        concatenatedFileName);
197
198   // static method for adding edges that can be run by multiple threads
199   static void parallelAddEdges(SimilarityGraph* graph,
200     const HashBucketIndex* indices,
201     unsigned int startingPosition, unsigned int stoppingPosition, string
        fileName, bool isUsingWagnerFischerForEdgeWeigths);
202
203   // construct a CIGARCoil file with a given parent array, and the original
        FASTA/FASTQ file
204   void reconstructCompressedFile(const char *uncompressedFileName, const
        char *reconstructedCompressedFileName, const char *
        compressedIdAndQualityFileName, bool isBeingMemoryConservative);
205
206   ~DNAFileWrapper();
207
208   // Returns string at the specified element i
209   std::string at(size_t i);
210
211   // square bracket operator for accessing the ith sequence
212   std::string operator[](size_t i);
213
214
215 };
216
217
218 #endif // !DNAFileParser_H
```

H.2 Definitions

```
1  #include "DNAFileWrapper.h"
2
3  DNAFileWrapper::DNAFileWrapper(const char *fileName) {
4
5    ifstream fileStream;
6
7    fileStream.open(fileName);
8
9    if (fileStream.bad()) {
10     printf("file stream is bad\n");
11   }
12
13   myFileName = fileName;
14   parentArray = vector<unsigned int>();
15
16   isInitialized = false;
17   numberOfElementsToCache = 1000;
18
19   unsigned int idOfFirstElementCached = 0;
20   unsigned int idOfLastElementCached = 0;
21   vector<string> cachedElements = vector<string>();
22
23   for (size_t i = 0; i < numberOfStates; i++) {
24     for (size_t j = 0; j < numberOfActions; j++) {
25       stateActionPairs[i][j] = 0;
26     }
```

```
27  }
28
29
30  readPositions = vector<std::streampos>();
31
32  string firstLine;
33  string secondLine;
34  string thirdLine;
35
36  getline(fileStream, firstLine);
37
38  parentArrayLength = 0;
39  root = 0;
40  idQualityStart = 0;
41
42  if (firstLine.length() > 4 && firstLine.substr(0, 4) == cigarFileMarker)
     {
43    // this file is a CIGARCoil file
44    fileType = DNAFileType::CIGARCOIL;
45
46    fileStream.close();
47
48    // extract parent array from CIGARCoil file
49    parentArray = findParentArray(fileName, 4, &parentArrayLength);
50    for (size_t i = 0; i < parentArray.size(); i++) {
51      if (parentArray[i] == i) {
52        root = i;
53        break;
54      }
55    }
56
57    fileStream.close();
58    fileStream.open(fileName, ios::binary);
59    fileStream.seekg(parentArrayLength, ios_base::beg);
60    readPositions.push_back(fileStream.tellg());
61    unsigned long long previousInsertionPosition = parentArrayLength;
62    string temp = "";
63    string bigTemp = "";
64    size_t tempMax = 0;
65    size_t positionOfTempMax;
66    double averageCig = 0.0;
67    unsigned int rNum = 0;
68    for (size_t i = 1; i < parentArray.size(); i++) {
69      getline(fileStream, temp);
70      if (temp == "") {
71        printf("hit.\n");
72      }
73      else if (fileStream.eof()) {
74        printf("end of file reached early.\n");
75      }
76      else if (!fileStream.is_open()) {
77        printf("not open.\n");
78      }
79      else if (fileStream.bad()) {
80        printf("bad\n");
81      }
82      else {
83        if (temp.size() > tempMax) {
```

```
84            tempMax = temp.size();
85            bigTemp = temp;
86            positionOfTempMax = rNum;
87          }
88          averageCig = (temp.size() + averageCig) / ++rNum;
89
90        }
91        std::streampos sPos = fileStream.tellg();
92
93
94
95        readPositions.push_back(fileStream.tellg());
96        previousInsertionPosition = readPositions[i];
97 #ifdef DEBUG
98        ifstream testStream(myFileName.c_str());
99        testStream.seekg(readPositions[i - 1], ios_base::beg);
100       string testString = "";
101       getline(testStream, testString);
102       if (temp != testString) {
103         printf("something is wrong\n");
104       }
105 #endif
106     }
107
108     decodedReads = DecodedReads(parentArray.size());
109
110     numberOfElementsToCache = (numberOfElementsToCache > parentArray.size()
      ) ? parentArray.size() : numberOfElementsToCache;
111
112     getline(fileStream, temp);
113     idQualityStart = fileStream.tellg();//temp.size() +
      previousInsertionPosition + 2;
114
115   }
116   else {
117
118     getline(fileStream, secondLine);
119     getline(fileStream, thirdLine);
120
121     readLength = secondLine.length();
122
123     char firstLineFirstChar = firstLine.at(0);
124     char thirdLineFirstChar = thirdLine.at(0);
125     if (firstLineFirstChar == '@' && thirdLineFirstChar == '+') {
126       fileType = DNAFileType::FASTQ;
127       isFASTQ = true;
128     }
129     else if (firstLineFirstChar == '>' || firstLineFirstChar == '@') {
130       fileType = DNAFileType::FASTA;
131       isFASTQ = false;
132     }
133     else if (false) {
134       // TODO: Condition if SAM format
135       isFASTQ = false;
136     }
137   }
138
139   fileStream.close();
```

```
140
141 }
142
143 #ifndef USING_THREAD
144
145 #endif // !USING_THREAD
146
147 // adds part of the edges to a similarity graph
148 void DNAFileWrapper::parallelAddEdges(SimilarityGraph* graph,
149   const HashBucketIndex* indices,
150   unsigned int startingPosition, unsigned int stoppingPosition, string
       fileName, bool isUsingWagnerFischerForEdgeWeigths) {
151
152   ifstream fileStream;
153   fileStream.open(fileName.c_str());
154
155   string rootSequence = CigarCoilUtilities::getDataAtFilePosition(graph->
       getReadAt(0).getSequencePos(), graph->getReadAt(0).getSequenceLength(),
       &fileStream);
156
157   for (unsigned int i = startingPosition; i < stoppingPosition; i++) {
158
159     double bestMatchSoFar = 10000;
160     Read currentRead = graph->getReadAt(i);
161     clock_t intersectionStart = clock();
162
163     vector<vector<unsigned int>> vectors = vector<vector<unsigned int>>();
164
165     // populates vector with all queries for the current read's partitions
166     for (size_t p = 0; p < currentRead.partitions.size(); p++) {
167       const vector<unsigned int> * currentVector = indices->at(p,
       currentRead.partitions.at(p));
168       unsigned int firstIndex = CigarCoilUtilities::
       findFirstIndexGreaterThanTarget(currentVector, i);
169
170       vector<unsigned int>::const_iterator first = currentVector->begin() +
       firstIndex;
171       vector<unsigned int>::const_iterator last = currentVector->end();
172       vector<unsigned int> entriesGreaterThanThisRead(first, last);
173
174       vectors.push_back(entriesGreaterThanThisRead);
175     }
176
177     vector<unsigned int> bestIntersectionVector = vector<unsigned int>();
178     size_t bestSizeSoFar = 2000000;
179     bool intersectionsOfAllSetsFound = true;
180
181     vector<vector<unsigned int>> intersections = vector<vector<unsigned int
       >>();
182     do {
183
184       intersections.clear();
185
186       for (size_t q = 0; q < vectors.size(); q += 2) {
187         // if odd number of vectors, last-most vector skips to next round
188         if (q == vectors.size() - 1) {
189           intersections.push_back(vectors.at(q));
190         }
```

```
191         else {
192           vector<unsigned int> intersectionVector = vector<unsigned int>();
193           vector<unsigned int> *v1 = &vectors.at(q);
194           vector<unsigned int> *v2 = &vectors.at(q + 1);
195 
196           // takes intersection of two vectors
197           set_intersection(v1->begin(), v1->end(),
198             v2->begin(), v2->end(),
199             back_inserter(intersectionVector));
200 
201           if (intersectionVector.size() > 0) {
202             intersections.push_back(intersectionVector);
203             if (intersectionVector.size() < bestSizeSoFar) {
204               bestIntersectionVector = intersectionVector;
205               bestSizeSoFar = intersectionVector.size();
206             }
207           }
208           else {
209             intersectionsOfAllSetsFound = false;
210           }
211         }
212       }
213 
214       // next round will have about half as many vectors left
215       vectors = intersections;
216 
217     } while (intersections.size() > 1);
218 
219     // if only one vector remains, then it is assumed to be the best
220     if (intersections.size() == 1) {
221       bestIntersectionVector = intersections.at(0);
222     }
223 
224     if (!fileStream.is_open()) {
225       fileStream.open(fileName);
226     }
227     string parentSequence = CigarCoilUtilities::getDataAtFilePosition(
    currentRead.getSequencePos(), currentRead.getSequenceLength(), &
    fileStream);
228 
229     Read *childRead;
230 
231     // additional step for reducing set size that is useless to enter if
    all intersections were successful
232     if (bestIntersectionVector.size() >
    maximumNumberOfReadsToApplyWagnerFischerTo && !
    intersectionsOfAllSetsFound) {
233       vector<unsigned int> correspondingIndices = vector<unsigned int>();
234       vector<double> intersectionsApproximateValues = vector<double>();
235 
236       // initialize vector of best values
237       unsigned int currentCandidate = bestIntersectionVector[0];
238       double approximateEditDistance = currentRead.getAngleBetweenTwoReads
    (&graph->getReadAt(currentCandidate));
239       for (size_t j = 0; j < maximumNumberOfReadsToApplyWagnerFischerTo; j
    ++) {
240         correspondingIndices.push_back(currentCandidate);
241         intersectionsApproximateValues.push_back(approximateEditDistance);
```

```
242       }
243
244       // populate vectors with best values
245       for (size_t k = 1; k < bestIntersectionVector.size(); k++) {
246         currentCandidate = bestIntersectionVector[k];
247
248         approximateEditDistance = currentRead.getAngleBetweenTwoReads(&
    graph->getReadAt(currentCandidate));
249
250         unsigned int firstIndexGreaterThanTarget = CigarCoilUtilities::
    findFirstIndexGreaterThanTarget(&intersectionsApproximateValues,
    approximateEditDistance);
251
252         if (firstIndexGreaterThanTarget < intersectionsApproximateValues.
    size()) {
253           // transition sorted elements of vectors up by 1 to make room for
     new value
254           for (unsigned int m = maximumNumberOfReadsToApplyWagnerFischerTo
    - 1; m > firstIndexGreaterThanTarget; m--) {
255             intersectionsApproximateValues[m] =
    intersectionsApproximateValues[m - 1];
256             correspondingIndices[m] = correspondingIndices[m - 1];
257           }
258
259           intersectionsApproximateValues[firstIndexGreaterThanTarget] =
    approximateEditDistance;
260           correspondingIndices[firstIndexGreaterThanTarget] =
    currentCandidate;
261
262         }
263
264       }
265
266       bestIntersectionVector.clear();
267       unsigned int previousInsertion = 2000000000;
268       for (size_t n = 0; n < maximumNumberOfReadsToApplyWagnerFischerTo; n
    ++) {
269         if (correspondingIndices[n] == previousInsertion) {
270           n = maximumNumberOfReadsToApplyWagnerFischerTo;
271         }
272         else {
273           bestIntersectionVector.push_back(correspondingIndices[n]);
274           previousInsertion = correspondingIndices[n];
275         }
276       }
277
278     }
279
280     unsigned char currentBest = UCHAR_MAX;
281     for (size_t k = 0; k < bestIntersectionVector.size(); k++) {
282
283       unsigned int bestChild = bestIntersectionVector.at(k);
284       childRead = &graph->getReadAt(bestChild);
285
286       string childSequence = CigarCoilUtilities::getDataAtFilePosition(
    childRead->getSequencePos(), childRead->getSequenceLength(), &fileStream
    );
287
```

```
288       unsigned char numberOfOperationsRequired = 0;
289
290       if (isUsingWagnerFischerForEdgeWeigths) {
291         WagnerFischerMatrix matrix = WagnerFischerMatrix(&parentSequence, &
    childSequence);
292         numberOfOperationsRequired = matrix.getEditDistance();
293       }
294       else {
295         numberOfOperationsRequired = parentSequence.size() -
    CigarCoilUtilities::getCheapSimilarityDistanceMetric(&parentSequence, &
    childSequence, 0) + 1;
296       }
297
298       // caps the weight of an edge at 255 - this should rarely if ever
    occur
299       unsigned char cappedNumberOfRequirements = (
    numberOfOperationsRequired > 255) ? 255 : numberOfOperationsRequired;
300
301       if (cappedNumberOfRequirements < currentBest) {
302         // adds edge to this undirected graph
303         graph->addEdge(i, bestChild, cappedNumberOfRequirements);
304         currentBest = cappedNumberOfRequirements;
305       }
306
307     }
308
309   }
310   fileStream.close();
311 }
312
313 // concatenates a set of files together
314 void DNAFileWrapper::concatenateFilesTogether(const string *files, size_t
    numberOfFiles, string resultFileName) {
315   ofstream appenderStream(resultFileName.c_str(), std::ios_base::app | std
    ::ios_base::binary);
316
317   for (size_t n = 0; n < numberOfFiles; n++) {
318     std::ifstream fileStream(files[n].c_str(), std::ios_base::binary);
319
320     appenderStream.seekp(0, std::ios_base::end);
321     appenderStream << fileStream.rdbuf();
322
323     fileStream.close();
324   }
325
326   appenderStream.close();
327 }
328
329 vector<unsigned int> DNAFileWrapper::findParentArray(const char *fileName,
    size_t offset, streampos *finalPosition) {
330
331   ifstream fileStream;
332
333   fileStream.open(fileName, ios::binary);
334
335   // populate parent array
336   fileStream.seekg(offset, ios_base::beg);
337
```

```
338   const size_t bufferSize = sizeof(int);
339   char verificationBuffer[bufferSize];
340   char buffer[bufferSize];
341
342   streamsize amountRead = 0;
343   unsigned int root = 0;
344   unsigned int numberOfReads = 0;
345
346   vector<unsigned int> resultArray = vector<unsigned int>();
347
348   while (true) {
349
350     if (!fileStream) {
351       printf("throw error for not finding end sequence\n");
352       return  resultArray;
353     }
354
355     unsigned int positionInBuffer = 0;
356     unsigned int amountToRead = bufferSize;
357     unsigned int makeupCharacterCount = 0;
358     unsigned int numberOfConsecutiveReturnCharacters = 0;
359
360     do {
361       makeupCharacterCount = 0;
362       char *verificationBuffer = new char[amountToRead];
363       fileStream.read(verificationBuffer, amountToRead);
364
365       for (size_t i = 0; i < amountToRead; i++) {
366
367         if (verificationBuffer[i] == '\r') {
368           numberOfConsecutiveReturnCharacters++;
369         }
370         else {
371           numberOfConsecutiveReturnCharacters = 0;
372         }
373
374         if ((i != (amountToRead - 1)) &&
    numberOfConsecutiveReturnCharacters % 2 == 1 && verificationBuffer[i +
    1] == '\n') {
375           makeupCharacterCount++;
376           buffer[positionInBuffer++] = verificationBuffer[++i];
377         }
378         else if ((positionInBuffer > 0) && buffer[positionInBuffer - 1] ==
    '\r' && verificationBuffer[i] == '\n') {
379           buffer[positionInBuffer - 1] = '\n';
380           makeupCharacterCount++;
381         }
382         else {
383           buffer[positionInBuffer++] = verificationBuffer[i];
384         }
385       }
386       amountToRead = makeupCharacterCount;
387       delete[] verificationBuffer;
388     } while (makeupCharacterCount > 0);
389
390     if (buffer[3] == '\r') { // I hate windows
391       streampos p = fileStream.tellg();
392       char checkBuffer[1];
```

```
393       fileStream.read(checkBuffer, 1);
394       if (checkBuffer[0] == '\n') {
395         buffer[3] = '\n';
396       }
397       else {
398         fileStream.seekg(p, ios_base::beg);
399       }
400     }
401
402     unsigned int currentValue = *reinterpret_cast<unsigned int*>(buffer);
403
404     if (currentValue == UINT32_MAX)
405       break;
406
407     if (resultArray.size() == currentValue)
408       root = resultArray.size();
409     resultArray.push_back(currentValue);
410     numberOfReads++;
411   }
412
413   *finalPosition = fileStream.tellg();
414   fileStream.close();
415
416   return resultArray;
417 }
418
419 // This method concatenates a set of cigarcoil compressed sequences
    together, encoding the root of each cluster
420 // relative to the root of the first file and merging their parent arrays
    together.
421 void DNAFileWrapper::concatenateCompressedSequencesTogether(const string *
    files, size_t numberOfFiles, string resultFileName) {
422
423   // the concatenated files will be written to the result file
424   ofstream outputFile(resultFileName.c_str(), std::ofstream::out);
425
426   // this vector will hold the combined parent arrays
427   vector<unsigned int> combinedParentArray = vector<unsigned int>();
428   streampos * parentArrayLengths = new streampos[numberOfFiles];
429   size_t *numberOfElements = new size_t[numberOfFiles];
430   unsigned int *roots = new unsigned int[numberOfFiles];
431
432   vector<string> rootSequences = vector<string>();
433
434   // combines the parent arrays of all files
435
436   for (size_t n = 0; n < numberOfFiles; n++) {
437     vector<unsigned int> currentParentArray = findParentArray(files[n].
     c_str(), 4, &parentArrayLengths[n]);
438     size_t sizePreConcatenation = combinedParentArray.size();
439     numberOfElements[n] = currentParentArray.size();
440     for (unsigned int i = 0; i < currentParentArray.size(); i++) {
441
442       if (i == currentParentArray.at(i)) {
443         roots[n] = i;
444         currentParentArray[i] = roots[0];
445       }
446       else {
```

```
447         currentParentArray[i] += sizePreConcatenation;
448       }
449
450       combinedParentArray.push_back(currentParentArray[i]);
451
452     }
453   }
454
455   // writes special identifying 4 bytes to signal that this file is a
      cigarcoil file
456   outputFile.write(cigarFileMarker, 4);
457
458   unsigned int* parents = &combinedParentArray[0];
459
460   outputFile.write(reinterpret_cast<char *>(parents), sizeof(int) *
      combinedParentArray.size());
461
462   unsigned int arrayEnding = UINT32_MAX;
463   outputFile.write(reinterpret_cast<char *>(&arrayEnding), sizeof(int));
464
465   ifstream firstFileStream(files[0].c_str());
466
467   string parentSequence = "";
468
469   string currentLine = "";
470
471   size_t readNumber = 0;
472
473   firstFileStream.seekg(parentArrayLengths[0], ios_base::beg);
474
475   while (firstFileStream) {
476     getline(firstFileStream, currentLine);
477     if (readNumber == roots[0])
478       parentSequence = currentLine;
479     outputFile << currentLine;
480
481     if (readNumber != (numberOfElements[0] - 1))
482       outputFile << "\n";
483
484     readNumber++;
485   }
486
487   for (size_t n = 1; n < numberOfFiles; n++) {
488     ifstream currentFileStream(files[n].c_str());
489     currentFileStream.seekg(parentArrayLengths[n], ios_base::beg);
490     readNumber = 0;
491
492     while (currentFileStream) {
493       getline(currentFileStream, currentLine);
494
495       if (readNumber == roots[n]) {
496         string cigar = WagnerFischerMatrix(&parentSequence, &currentLine).
      getCigar();
497         outputFile << cigar;
498       }
499       else {
500         outputFile << currentLine;
501       }
```

```
502
503       if (readNumber != (numberOfElements[n] - 1))
504         outputFile << "\n";
505
506       readNumber++;
507
508     }
509   }
510
511   delete[] numberOfElements;
512   delete[] roots;
513   delete[] parentArrayLengths;
514   outputFile.close();
515 }
516
517 // creates a set of random strings of the specified length
518 string *DNAFileWrapper::initializeCentroids(size_t numberOfCentroids,
      unsigned short sequenceLength) {
519
520   string *centroids = new string[numberOfCentroids];
521
522   // initialize centroids
523   for (size_t i = 0; i < numberOfCentroids; i++) {
524     string centroid = "";
525     // populate centroid with random sequence
526     for (size_t stringPos = 0; stringPos < sequenceLength; stringPos++) {
527       int randomValue = rand() % 4;
528       switch (randomValue) {
529       case 0:
530         centroid += 'A';
531         break;
532       case 1:
533         centroid += 'C';
534         break;
535       case 2:
536         centroid += 'T';
537         break;
538       case 3:
539         centroid += 'G';
540         break;
541       }
542     }
543     centroids[i] = centroid;
544   }
545
546   return centroids;
547 }
548
549 void DNAFileWrapper::recomputeCentroids(double ***
      runningAverageForEachReadPosition, size_t numberOfCentroids, string *
      centroids, unsigned short sequenceLength) {
550   for (size_t c = 0; c < numberOfCentroids; c++) {
551     string newCentroid = "";
552     // calculate new centroid based on its cluster
553     for (size_t r = 0; r < sequenceLength; r++) {
554       double aAverage = runningAverageForEachReadPosition[0][r][c];
555       double cAverage = runningAverageForEachReadPosition[1][r][c];
556       double tAverage = runningAverageForEachReadPosition[2][r][c];
```

```
557       double gAverage = runningAverageForEachReadPosition[3][r][c];
558
559       switch (CigarCoilUtilities::greatestOfFour(aAverage, cAverage,
    tAverage, gAverage)) {
560       case 1:
561         newCentroid += 'A';
562         break;
563       case 2:
564         newCentroid += 'C';
565         break;
566       case 3:
567         newCentroid += 'T';
568         break;
569       case 4:
570         newCentroid += 'G';
571         break;
572       }
573
574     }
575     centroids[c] = newCentroid;
576   }
577 }
578
579 // This method performs kmeans on a set of DNA read sequences
580 // this implementation of kmeans stops after the given number of iterations
     is completed or there is no change
581 // this function returns an array of vectors such that there is one vector
    for each cluster
582 vector<unsigned int> *DNAFileWrapper::kmeans(const vector<string> *
    sequences, size_t numberOfClusters, size_t maximumNumberOfIterations,
    unsigned short sequenceLength) {
583
584
585   clock_t start = clock();
586
587   double centroidBuildingTime = 0;
588   double wagnerFischerTime = 0;
589   double updatingAveragesTime = 0;
590   double binarySearchTime = 0;
591
592   // this struct will track which reads belong to which cluster
593   vector<unsigned int> *clusterings = new vector<unsigned int>[
     numberOfClusters];
594
595   for (size_t i = 0; i < numberOfClusters; i++) {
596     clusterings[i] = vector<unsigned int>();
597   }
598
599   string *centroids = initializeCentroids(numberOfClusters, sequenceLength)
      ;
600
601   bool isChange = false;
602   size_t currentIteration = 0;
603
604   // continues until there is either no change or the current number of
      iterations exceeds the maximum
605   do {
606     isChange = false;
```

```
607     unsigned int currentRead = 0;
608
609     // this vector is used to ascertain if a change has occurred
610     vector<unsigned int> *comparisonClusterings = new vector<unsigned int>[
    numberOfClusters];
611     for (size_t i = 0; i < numberOfClusters; i++) {
612       comparisonClusterings[i] = clusterings[i];
613       clusterings[i].clear();
614     }
615
616     for (size_t s = 0; s < sequences->size(); s++) {
617
618       unsigned short currentBestSimilarityMetric = 0;
619       size_t bestCentroid = 0;
620
621       clock_t editDistanceStart = clock();
622
623       const string * currentSequence = &sequences->at(s);
624
625       // determination of closest centroid to this particular read
626       for (size_t j = 0; j < numberOfClusters; j++) {
627         unsigned short similarityMetric = CigarCoilUtilities::
    getCheapSimilarityDistanceMetric(&centroids[j], &sequences->at(s),
    currentBestSimilarityMetric);
628         if (similarityMetric > currentBestSimilarityMetric) {
629           currentBestSimilarityMetric = similarityMetric;
630           bestCentroid = j;
631         }
632       }
633
634       wagnerFischerTime += (clock() - editDistanceStart) / (double)
    CLOCKS_PER_SEC;
635
636       // take note that this read belongs to this centroid for later
637       clusterings[bestCentroid].push_back(currentRead);
638       // if no change has been detected yet - check the original cluster
    for the current read
639       clock_t binSearchStart = clock();
640       if (!isChange && !(std::binary_search(comparisonClusterings[
    bestCentroid].begin(), comparisonClusterings[bestCentroid].end(),
    currentRead))) {
641         isChange = true;
642       }
643
644       binarySearchTime += (clock() - binSearchStart) / (double)
    CLOCKS_PER_SEC;
645
646       currentRead++;
647     }
648
649     delete[] comparisonClusterings;
650
651     // recompute centroids
652     if (isChange) {
653
654       // maintains averages for A C T, and G so that the centroids can be
    recomputed
655       // the first dimension is whether the average is for A C T or G
```

```
656       // the second dimension is for the position in the centroid
657       // the third dimension identifies the centroid
658       double ***runningAverageForEachReadPosition = new double**[4];
659       unsigned int currentRead = 0;
660
661       for (size_t k = 0; k < 4; k++) {
662         runningAverageForEachReadPosition[k] = new double*[sequenceLength];
663
664         for (size_t l = 0; l < sequenceLength; l++) {
665
666           runningAverageForEachReadPosition[k][l] = new double[
    numberOfClusters];
667
668           for (size_t m = 0; m < numberOfClusters; m++) {
669             runningAverageForEachReadPosition[k][l][m] = 0.0;
670           }
671         }
672       }
673
674
675       for (size_t s = 0; s < sequences->size(); s++) {
676
677         // Update running averages for the cluster that this read belongs
    to
678         for (size_t i = 0; i < numberOfClusters; i++) {
679
680           clock_t binarySearchStart = clock();
681
682           if (binary_search(clusterings[i].begin(), clusterings[i].end(),
    currentRead)) {
683
684             binarySearchTime += (clock() - binarySearchStart) / (double)
    CLOCKS_PER_SEC;
685
686             clock_t averageUpdateStart = clock();
687
688             size_t clusterSize = clusterings[i].size();
689             // update running average for this cluster
690             for (size_t j = 0; j < sequenceLength && j < sequences->at(s).
    length(); j++) {
691               switch (sequences->at(s).at(j)) {
692               case 'A':
693                 runningAverageForEachReadPosition[0][j][i] += (1.0) /
    clusterSize;
694                 break;
695               case 'C':
696                 runningAverageForEachReadPosition[1][j][i] += (1.0) /
    clusterSize;
697                 break;
698               case 'T':
699                 runningAverageForEachReadPosition[2][j][i] += (1.0) /
    clusterSize;
700                 break;
701               case 'G':
702                 runningAverageForEachReadPosition[3][j][i] += (1.0) /
    clusterSize;
703                 break;
704               }
```

```
705             }
706
707             updatingAveragesTime += (clock() - averageUpdateStart) / (
    double)CLOCKS_PER_SEC;
708
709             // found match - end this loop early
710             i = numberOfClusters;
711           }
712         }
713         currentRead++;
714       }
715
716       // recompute centroids based on running averages
717
718       clock_t centroidRecomputingStart = clock();
719
720       recomputeCentroids(runningAverageForEachReadPosition,
    numberOfClusters, centroids, sequenceLength);
721
722       centroidBuildingTime += (clock() - centroidRecomputingStart) / (
    double)CLOCKS_PER_SEC;
723
724       // cleanup memory
725       for (size_t k = 0; k < 4; k++) {
726         delete[] runningAverageForEachReadPosition[k];
727       }
728       delete[] runningAverageForEachReadPosition;
729
730     } // if change
731
732   } while (isChange && ((++currentIteration) < maximumNumberOfIterations));
733
734   // no longer care what the centroids were - reclaim memory
735   delete[] centroids;
736
737   double duration = (clock() - start) / (double)CLOCKS_PER_SEC;
738
739   printf("K means finished taking %f seconds...\n", duration);
740   printf("\t%f was spent running binary search\n", binarySearchTime);
741   printf("\t%f was spent updating averages\n", updatingAveragesTime);
742   printf("\t%f was spent computing edit distance\n", wagnerFischerTime);
743   printf("\t%f was spent recomputing centroids\n", centroidBuildingTime);
744
745   return clusterings;
746 }
747
748 // This method performs kmeans on a set of DNA read sequences
749 // this implementation of kmeans stops after the given number of iterations
      is completed or there is no change
750 // this function returns an array of vectors such that there is one vector
     for each cluster
751 vector<unsigned int> *DNAFileWrapper::kmeans(const char * sequenceFileName,
      size_t numberOfClusters, size_t maximumNumberOfIterations, unsigned
    short sequenceLength) {
752
753
754   clock_t start = clock();
755
```

```
756   double centroidBuildingTime = 0;
757   double wagnerFischerTime = 0;
758   double updatingAveragesTime = 0;
759   double binarySearchTime = 0;
760
761   string sequenceLine = "";
762
763   // this struct will track which reads belong to which cluster
764   vector<unsigned int> *clusterings = new vector<unsigned int>[
       numberOfClusters];
765
766   for (size_t i = 0; i < numberOfClusters; i++) {
767     clusterings[i] = vector<unsigned int>();
768   }
769
770   string *centroids = initializeCentroids(numberOfClusters, sequenceLength)
       ;
771
772   bool isChange = false;
773   size_t currentIteration = 0;
774
775   // continues until there is either no change or the current number of
       iterations exceeds the maximum
776   do {
777     isChange = false;
778     unsigned int currentRead = 0;
779     ifstream sequenceFileStream(sequenceFileName);
780
781     // this vector is used to ascertain if a change has occurred
782     vector<unsigned int> *comparisonClusterings = new vector<unsigned int>[
       numberOfClusters];
783     for (size_t i = 0; i < numberOfClusters; i++) {
784       comparisonClusterings[i] = clusterings[i];
785       clusterings[i].clear();
786     }
787
788     while (sequenceFileStream) {
789
790       // consumes sequence line from file
791       if (!getline(sequenceFileStream, sequenceLine)) break;
792
793       unsigned short currentBestSimilarityMetric = 0;
794       size_t bestCentroid = 0;
795
796       clock_t editDistanceStart = clock();
797
798       // determination of closest centroid to this particular read
799       for (size_t j = 0; j < numberOfClusters; j++) {
800         unsigned short similarityMetric = CigarCoilUtilities::
       getCheapSimilarityDistanceMetric(&centroids[j], &sequenceLine,
       currentBestSimilarityMetric);
801         if (similarityMetric > currentBestSimilarityMetric) {
802           currentBestSimilarityMetric = similarityMetric;
803           bestCentroid = j;
804         }
805       }
806
807       wagnerFischerTime += (clock() - editDistanceStart) / (double)
```

```
    CLOCKS_PER_SEC;
808
809         // take note that this read belongs to this centroid for later
810         clusterings[bestCentroid].push_back(currentRead);
811         // if no change has been detected yet - check the original cluster
    for the current read
812         clock_t binSearchStart = clock();
813         if (!isChange && !(std::binary_search(comparisonClusterings[
    bestCentroid].begin(), comparisonClusterings[bestCentroid].end(),
    currentRead))) {
814           isChange = true;
815         }
816
817         binarySearchTime += (clock() - binSearchStart) / (double)
    CLOCKS_PER_SEC;
818
819         currentRead++;
820       }
821
822       sequenceFileStream.close();
823
824       delete[] comparisonClusterings;
825
826       // recompute centroids
827       if (isChange) {
828         sequenceFileStream.open(sequenceFileName);
829
830         // maintains averages for A C T, and G so that the centroids can be
    recomputed
831         // the first dimension is whether the average is for A C T or G
832         // the second dimension is for the position in the centroid
833         // the third dimension identifies the centroid
834         double ***runningAverageForEachReadPosition = new double**[4];
835         unsigned int currentRead = 0;
836
837         for (size_t k = 0; k < 4; k++) {
838           runningAverageForEachReadPosition[k] = new double*[sequenceLength];
839
840           for (size_t l = 0; l < sequenceLength; l++) {
841
842             runningAverageForEachReadPosition[k][l] = new double[
    numberOfClusters];
843
844             for (size_t m = 0; m < numberOfClusters; m++) {
845               runningAverageForEachReadPosition[k][l][m] = 0.0;
846             }
847           }
848         }
849
850
851         while (sequenceFileStream) {
852
853           // consumes sequence line from file
854           if (!getline(sequenceFileStream, sequenceLine)) break;
855
856           // Update running averages for the cluster that this read belongs
    to
857           for (size_t i = 0; i < numberOfClusters; i++) {
```

```
858
859         clock_t binarySearchStart = clock();
860
861         if (binary_search(clusterings[i].begin(), clusterings[i].end(),
    currentRead)) {
862
863           binarySearchTime += (clock() - binarySearchStart) / (double)
    CLOCKS_PER_SEC;
864
865           clock_t averageUpdateStart = clock();
866
867           size_t clusterSize = clusterings[i].size();
868           // update running average for this cluster
869           for (size_t j = 0; j < sequenceLength && j < sequenceLine.
    length(); j++) {
870             switch (sequenceLine.at(j)) {
871             case 'A':
872               runningAverageForEachReadPosition[0][j][i] += (1.0) /
    clusterSize;
873               break;
874             case 'C':
875               runningAverageForEachReadPosition[1][j][i] += (1.0) /
    clusterSize;
876               break;
877             case 'T':
878               runningAverageForEachReadPosition[2][j][i] += (1.0) /
    clusterSize;
879               break;
880             case 'G':
881               runningAverageForEachReadPosition[3][j][i] += (1.0) /
    clusterSize;
882               break;
883             }
884           }
885
886           updatingAveragesTime += (clock() - averageUpdateStart) / (
    double)CLOCKS_PER_SEC;
887
888           // found match - end this loop early
889           i = numberOfClusters;
890         }
891       }
892       currentRead++;
893     }
894
895     sequenceFileStream.close();
896
897     // recompute centroids based on running averages
898
899     clock_t centroidRecomputingStart = clock();
900
901     recomputeCentroids(runningAverageForEachReadPosition,
    numberOfClusters, centroids, sequenceLength);
902
903     centroidBuildingTime += (clock() - centroidRecomputingStart) / (
    double)CLOCKS_PER_SEC;
904
905     // cleanup memory
```

```
906         for (size_t k = 0; k < 4; k++) {
907           delete[] runningAverageForEachReadPosition[k];
908         }
909         delete[] runningAverageForEachReadPosition;
910 
911 
912 
913       } // if change
914 
915     } while (isChange && ((++currentIteration) < maximumNumberOfIterations));
916 
917     // no longer care what the centroids were - reclaim memory
918     delete[] centroids;
919 
920     double duration = (clock() - start) / (double)CLOCKS_PER_SEC;
921 
922     printf("K means finished taking %f seconds...\n", duration);
923     printf("\t%f was spent running binary search\n", binarySearchTime);
924     printf("\t%f was spent updating averages\n", updatingAveragesTime);
925     printf("\t%f was spent computing edit distance\n", wagnerFischerTime);
926     printf("\t%f was spent recomputing centroids\n", centroidBuildingTime);
927 
928     return clusterings;
929 }
930 
931 // applies cigarcoil compression to the current FASTA or FASTQ file using
      clustering
932 //specified by the parameters and writing the result to the given file path
933 void DNAFileWrapper::encode(const char *encodedFileName, size_t
      numberOfClusters, size_t maximumNumberOfIterations, bool
      isBeingConservativeWithMainMemory, bool
      isUsingWagnerFischerForEdgeWeigths) {
934 
935     clock_t start = std::clock();
936 
937     string sequenceFileName = CigarCoilUtilities::createTemporaryFile();
938 
939     string idLine = "";
940     string sequenceLine = "";
941     string qualityScoreLine = "";
942     string plusSignLine = "";
943 
944     ofstream sequenceStream;
945 
946     sequenceStream.open(sequenceFileName.c_str(), ios::out);
947 
948     ifstream fileStream;
949 
950     fileStream.open(myFileName.c_str());
951 
952     stringstream sstream;
953 
954     vector<string> sequences = vector<string>();
955 
956     bool isStoringSequencesInMainMemoryDuringClustering = !
      isBeingConservativeWithMainMemory;
957 
958     unsigned short centroidReadLength = 0;
```

```
959   unsigned int numberOfReads = 0;
960
961   while (fileStream) {
962
963     if (!getline(fileStream, idLine)) break;
964
965     // consumes sequence line from file
966     if (!getline(fileStream, sequenceLine)) break;
967
968     // FASTQ only
969     if (fileType == FASTQ) {
970       if (!getline(fileStream, plusSignLine)) break;
971       if (!getline(fileStream, qualityScoreLine)) break;
972
973     }
974
975     numberOfReads++;
976
977     // update cumulative average of centroid length
978     centroidReadLength = (unsigned short)round((centroidReadLength * ((
      numberOfReads - 1.0) / numberOfReads)) + (sequenceLine.length() * (1.0 /
       numberOfReads)));
979
980     if (isStoringSequencesInMainMemoryDuringClustering) {
981       sequences.push_back(sequenceLine);
982     }
983     else {
984
985       sstream << sequenceLine << "\n";
986
987       sstream.seekp(0, ios::end);
988       stringstream::pos_type streamLength = sstream.tellp();
989
990       if (streamLength > 5000000) {
991         sequenceStream << sstream.rdbuf();
992         // clear stream content
993         sstream.str(string());
994       }
995     }
996   }
997
998   if (isStoringSequencesInMainMemoryDuringClustering) {
999     sequenceStream << sstream.rdbuf();
1000  }
1001
1002  sequenceStream.close();
1003
1004  double duration = (clock() - start) / (double)CLOCKS_PER_SEC;
1005
1006  printf("Took %f seconds to scan over file\n", duration);
1007
1008  start = clock();
1009
1010  vector<unsigned int> * clusterings =
       isStoringSequencesInMainMemoryDuringClustering ?
1011     kmeans(&sequences, numberOfClusters, maximumNumberOfIterations,
       centroidReadLength):
1012     kmeans(sequenceFileName.c_str(), numberOfClusters,
```

```
    maximumNumberOfIterations, centroidReadLength);

remove(sequenceFileName.c_str());
sequences.clear();

duration = (clock() - start) / (double)CLOCKS_PER_SEC;

printf("Took %f seconds to cluster reads\n", duration);

size_t numberOfNonEmptyClusters = 0;
vector<vector<unsigned int>> nonEmptyClusterings = vector<vector<unsigned
    int>>();
for (size_t c = 0; c < numberOfClusters; c++) {
  if (clusterings[c].size() > 0) {
    numberOfNonEmptyClusters++;
    nonEmptyClusterings.push_back(clusterings[c]);
  }
}

// create temporary file for each nonempty cluster
stringstream * uncompressedIdAndQualityScoresStringStreams = new
    stringstream[numberOfNonEmptyClusters];
stringstream * uncompressedSequencesStringStreams = new stringstream[
    numberOfNonEmptyClusters];
string* temporaryUncompressedIdsAndQualityScores = new string[
    numberOfNonEmptyClusters];
string* temporaryUncompressedSequences = new string[
    numberOfNonEmptyClusters];
for (size_t n = 0; n < numberOfNonEmptyClusters; n++) {
  temporaryUncompressedIdsAndQualityScores[n] = CigarCoilUtilities::
    createTemporaryFile();
  temporaryUncompressedSequences[n] = CigarCoilUtilities::
    createTemporaryFile();
}

size_t currentRead = 0;

fileStream.close();

start = clock();

fileStream.open(myFileName.c_str());

// split input file into files based on clusters
while (fileStream) {
  if (!getline(fileStream, idLine)) break;

  // consumes sequence line from file
  if (!getline(fileStream, sequenceLine)) break;

  // FASTQ only
  if (fileType == FASTQ) {
    if (!getline(fileStream, plusSignLine)) break;
    if (!getline(fileStream, qualityScoreLine)) break;
  }

  // split original file based on its clusters
  for (size_t i = 0; i < numberOfNonEmptyClusters; i++) {
```

```
1063      // use binary search to find the cluster that this read belongs to
    since clusters are presorted
1064      if (binary_search(nonEmptyClusterings[i].begin(), nonEmptyClusterings
    [i].end(), currentRead)) {
1065
1066        uncompressedIdAndQualityScoresStringStreams[i].seekp(0, ios::end);
1067        stringstream::pos_type streamLength =
    uncompressedIdAndQualityScoresStringStreams[i].tellp();
1068
1069        if (streamLength > 5000000) {
1070          ofstream relevantIdAndQualityScoreFile;
1071          relevantIdAndQualityScoreFile.open(
    temporaryUncompressedIdsAndQualityScores[i], ios::app);
1072          relevantIdAndQualityScoreFile <<
    uncompressedIdAndQualityScoresStringStreams[i].rdbuf();
1073          uncompressedIdAndQualityScoresStringStreams[i].str(string());
1074          relevantIdAndQualityScoreFile.close();
1075        }
1076
1077        uncompressedSequencesStringStreams[i].seekp(0, ios::end);
1078        streamLength = uncompressedSequencesStringStreams[i].tellp();
1079
1080        if (streamLength > 5000000) {
1081          ofstream relevantSequenceFile;
1082          relevantSequenceFile.open(temporaryUncompressedSequences[i], ios
    ::app);
1083          relevantSequenceFile << uncompressedSequencesStringStreams[i].
    rdbuf();
1084          uncompressedSequencesStringStreams[i].str(string());
1085          relevantSequenceFile.close();
1086        }
1087
1088        uncompressedIdAndQualityScoresStringStreams[i] << idLine << "\n";
1089        if (isFASTQ) {
1090          uncompressedIdAndQualityScoresStringStreams[i] <<
    qualityScoreLine << "\n";
1091        }
1092
1093        uncompressedSequencesStringStreams[i] << sequenceLine << "\n";
1094
1095        // found match - end this loop early
1096        i = numberOfClusters;
1097      }
1098    }
1099    currentRead++;
1100  }
1101
1102  fileStream.close();
1103
1104  delete[] clusterings;
1105  nonEmptyClusterings.clear();
1106
1107  // finished writing uncompressed files clean up
1108  for (size_t n = 0; n < numberOfNonEmptyClusters; n++) {
1109
1110    ofstream relevantIdAndQualityScoreFile;
1111    relevantIdAndQualityScoreFile.open(
    temporaryUncompressedIdsAndQualityScores[n], ios::app);
```

```
1112      relevantIdAndQualityScoreFile <<
          uncompressedIdAndQualityScoresStringStreams[n].rdbuf();
1113      uncompressedIdAndQualityScoresStringStreams[n].str(string());
1114      relevantIdAndQualityScoreFile.close();
1115
1116      ofstream relevantSequenceFile;
1117      relevantSequenceFile.open(temporaryUncompressedSequences[n], ios::app);
1118      relevantSequenceFile << uncompressedSequencesStringStreams[n].rdbuf();
1119      uncompressedSequencesStringStreams[n].str(string());
1120      relevantSequenceFile.close();
1121    }
1122
1123    // no longer need these
1124    delete[] uncompressedIdAndQualityScoresStringStreams;
1125    delete[] uncompressedSequencesStringStreams;
1126
1127    duration = (clock() - start) / (double)CLOCKS_PER_SEC;
1128
1129    printf("Took %f seconds to separate input file into cluster order\n",
          duration);
1130
1131    string reorderedIdAndQualityScoresFile = CigarCoilUtilities::
          createTemporaryFile();
1132
1133    concatenateFilesTogether(temporaryUncompressedIdsAndQualityScores,
          numberOfNonEmptyClusters, reorderedIdAndQualityScoresFile);
1134
1135    start = clock();
1136    string encodedIdFieldFile = CigarCoilUtilities::encodeZpaq(
          reorderedIdAndQualityScoresFile.c_str());
1137
1138    duration = (clock() - start) / (double)CLOCKS_PER_SEC;
1139
1140    printf("Took %f seconds to compress IDs and quality scores\n", duration);
1141
1142    remove(reorderedIdAndQualityScoresFile.c_str());
1143
1144    string encodedQualityFieldFile = "";
1145
1146    for (size_t n = 0; n < numberOfNonEmptyClusters; n++) {
1147      remove(temporaryUncompressedIdsAndQualityScores[n].c_str());
1148    }
1149
1150    delete[] temporaryUncompressedIdsAndQualityScores;
1151
1152    string* temporaryCompressedSequenceFiles = new string[
          numberOfNonEmptyClusters];
1153
1154    // individually compress each cluster
1155    for (size_t n = 0; n < numberOfNonEmptyClusters; n++) {
1156      temporaryCompressedSequenceFiles[n] = CigarCoilUtilities::
          createTemporaryFile();
1157      string toBeDeleted = temporaryCompressedSequenceFiles[n];
1158      temporaryCompressedSequenceFiles[n] = encodeSequenceFields(
          temporaryUncompressedSequences[n].c_str(), centroidReadLength,
          isUsingWagnerFischerForEdgeWeigths);
1159      remove(toBeDeleted.c_str());
1160      printf("Encoded cluster %d out of %d\n", n, numberOfNonEmptyClusters);
```

```
1161    }
1162
1163    start = clock();
1164
1165    concatenateCompressedSequencesTogether(temporaryCompressedSequenceFiles,
          numberOfNonEmptyClusters, encodedFileName);
1166
1167    duration = (clock() - start) / (double)CLOCKS_PER_SEC;
1168
1169    printf("Took %f seconds to compress sequences\n", duration);
1170
1171    for (size_t n = 0; n < numberOfNonEmptyClusters; n++) {
1172      remove(temporaryCompressedSequenceFiles[n].c_str());
1173      remove(temporaryUncompressedSequences[n].c_str());
1174    }
1175
1176    string * filesToAppendToEndOfSequences = new string[1];
1177    filesToAppendToEndOfSequences[0] = encodedIdFieldFile;
1178
1179    concatenateFilesTogether(filesToAppendToEndOfSequences, 1,
          encodedFileName);
1180
1181    delete[] filesToAppendToEndOfSequences;
1182
1183    // no longer need this array of file names;
1184    delete[] temporaryUncompressedSequences;
1185
1186    remove(encodedIdFieldFile.c_str());
1187
1188    // finish up garbage collection for this method
1189    delete[] temporaryCompressedSequenceFiles;
1190  }
1191
1192  #ifndef USING_THREAD
1193
1194  void * pthreadAddEdges(void *arguments)
1195  {
1196    struct argumentStruct *args = (argumentStruct *)arguments;
1197    DNAFileWrapper::parallelAddEdges(args->graph, args->indices, args->
          startingPosition, args->stoppingPosition, args->fileName, args->
          isUsingWagnerFischerForEdgeWeigths);
1198    return NULL;
1199  }
1200
1201  #endif // !USING_THREAD
1202
1203  // encodes file of sequence values and returns name of encoded file
1204  string DNAFileWrapper::encodeSequenceFields(const char *sequenceFile,
          unsigned short averageReadLength, bool
          isUsingWagnerFischerForEdgeWeigths) {
1205    string encodedFileName = CigarCoilUtilities::createTemporaryFile();
1206    ifstream fileStream;
1207    fileStream.open(sequenceFile);
1208
1209  #ifdef  unix
1210    unsigned numberOfCoresAvailable = get_nprocs();
1211  #else
1212    unsigned numberOfCoresAvailable = std::thread::hardware_concurrency();
```

```
1213 #endif //  unix
1214
1215
1216   // if hardware concurrency method fails then it returns 0 assume only 1
       core if this happens
1217   numberOfCoresAvailable = (numberOfCoresAvailable == 0) ? 1 :
       numberOfCoresAvailable;
1218
1219   SimilarityGraph similarityGraph = SimilarityGraph();
1220
1221   unsigned char partitionSize = averageReadLength > 84 ? 17 : 7;
1222
1223   HashBucketIndex hashBuckets = HashBucketIndex(partitionSize,
       averageReadLength);
1224
1225   string sequenceLine = "";
1226
1227   unsigned int numberOfReads = 0;
1228   // Reading in the file and constructing read objects
1229
1230   clock_t start = std::clock();
1231
1232   while (fileStream) {
1233
1234
1235     streampos filePosition = fileStream.tellg();
1236
1237     // consumes sequence line from file
1238     if (!getline(fileStream, sequenceLine)) break;
1239
1240     Read read = Read(sequenceLine, filePosition, averageReadLength,
       partitionSize);
1241
1242     similarityGraph.addRead(read);
1243     vector<unsigned char> currentReadsPartitionValues = read.partitions;
1244
1245     for (size_t p = 0; p < currentReadsPartitionValues.size(); p++) {
1246       hashBuckets.insert(p, currentReadsPartitionValues.at(p),
       numberOfReads);
1247     }
1248
1249     numberOfReads++;
1250   }
1251
1252   fileStream.close();
1253
1254   for (size_t i = 1; i < similarityGraph.getVectorSize(); i++) {
1255     similarityGraph.addEdge(i, 0, UCHAR_MAX);
1256   }
1257
1258
1259 #ifdef USING_THREAD
1260   thread *myThreads = new thread[numberOfCoresAvailable];
1261   unsigned int numberOfReadsPerThread = numberOfReads /
       numberOfCoresAvailable;
1262
1263   for (unsigned int t = 0; t < numberOfCoresAvailable; t++) {
1264
```

```
1265      unsigned int startingPoint = numberOfReadsPerThread * t;
1266
1267      // all reads have edge to 0
1268      if (startingPoint == 0)
1269            startingPoint++;
1270
1271      // last thread takes care of remainder of reads
1272      unsigned int stoppingPoint = (t == numberOfCoresAvailable - 1) ?
1273        numberOfReads :
1274        numberOfReadsPerThread * (t + 1);
1275
1276      myThreads[t] = thread(parallelAddEdges, &similarityGraph, &hashBuckets,
          startingPoint, stoppingPoint, sequenceFile,
        isUsingWagnerFischerForEdgeWeigths);
1277    }
1278
1279    // anticipate threads to end in descending order
1280    for (unsigned int t = 0; t < numberOfCoresAvailable; t++) {
1281      myThreads[t].join();
1282    }
1283  #else
1284    pthread_t *myThreads = new pthread_t[numberOfCoresAvailable];
1285
1286    unsigned int numberOfReadsPerThread = numberOfReads /
        numberOfCoresAvailable;
1287
1288    for (unsigned int t = 0; t < numberOfCoresAvailable; t++) {
1289
1290      unsigned int startingPoint = numberOfReadsPerThread * t;
1291      // all reads have edge to 0
1292      if (startingPoint == 0)
1293        startingPoint++;
1294
1295      // last thread takes care of remainder of reads
1296      unsigned int stoppingPoint = (t == numberOfCoresAvailable - 1) ?
1297        numberOfReads :
1298        numberOfReadsPerThread * (t + 1);
1299
1300      argumentStruct *arguments = new argumentStruct;
1301      arguments->fileName = sequenceFile;
1302      arguments->graph = &similarityGraph;
1303      arguments->indices = &hashBuckets;
1304      arguments->startingPosition = startingPoint;
1305      arguments->stoppingPosition = stoppingPoint;
1306
1307      pthread_create(&myThreads[t], NULL, pthreadAddEdges, (void *)&arguments
          );
1308    }
1309
1310    // anticipate threads to end in descending order
1311    for (unsigned int t = 0; t < numberOfCoresAvailable; t++) {
1312      pthread_join(myThreads[t], NULL);
1313    }
1314  #endif
1315
1316
1317    // don't need these now that edges are added
1318    similarityGraph.clearReadPartitionInfo();
```

```
1319
1320    // Create Minimum Spanning Tree
1321    unsigned int *parents = CigarCoilUtilities::PrimMST(&similarityGraph);
1322
1323    for (size_t i = 1; i < similarityGraph.getVectorSize(); i++) {
1324      if (parents[i] >= similarityGraph.getVectorSize())
1325        parents[i] = 0;
1326    }
1327
1328    // reopen the file stream
1329    fileStream.open(sequenceFile);
1330
1331    unsigned int root = 0;
1332
1333    unsigned int *minimumHeightParentArray = CigarCoilUtilities::
       getMinimumHeightTree(parents, similarityGraph.getVectorSize(), &root);
1334
1335    delete[] parents;
1336
1337    // Encode The MST
1338    encodeMSTAndWriteToFile(root, minimumHeightParentArray, &similarityGraph,
       encodedFileName, sequenceFile, &fileStream);
1339
1340    fileStream.close();
1341    delete[]minimumHeightParentArray;
1342
1343    return encodedFileName;
1344  }
1345
1346  // prints all not yet visited vertices reachable from s
1347  void DNAFileWrapper::encodeMSTAndWriteToFile(unsigned int root, unsigned
       int *parents, SimilarityGraph *similarityGraph, string outputFileName,
       string inputFileName, ifstream *fileStream)
1348  {
1349    ofstream outputFile(outputFileName.c_str(), std::ofstream::out);
1350
1351    // root of tree is identified by being its own parent
1352    parents[root] = root;
1353
1354    // writes special identifying 4 bytes to signal that this file is a
       cigarcoil file
1355    outputFile.write(cigarFileMarker, 4);
1356
1357    outputFile.write(reinterpret_cast<char *>(parents), sizeof(int) *
       similarityGraph->getVectorSize());
1358
1359    unsigned int arrayEnding = UINT32_MAX;
1360    outputFile.write(reinterpret_cast<char *>(&arrayEnding), sizeof(int));
1361
1362    stringstream sstream;
1363
1364    for (size_t i = 0; i < similarityGraph->getVectorSize(); i++) {
1365
1366      sstream.seekp(0, ios::end);
1367      stringstream::pos_type streamLength = sstream.tellp();
1368
1369      if (streamLength > 5000000) {
1370        outputFile << sstream.rdbuf();
```

```
1371        // clear stream content
1372        sstream.str(string());
1373      }
1374
1375      Read *childRead = &similarityGraph->getReadAt(i);
1376
1377      if (i == root) {
1378        // write root explicitly
1379        sstream << CigarCoilUtilities::getDataAtFilePosition(childRead->
      getSequencePos(), childRead->getSequenceLength(), fileStream) << "\n";
1380      }
1381      else {
1382
1383        Read *parentRead = &similarityGraph->getReadAt(parents[i]);
1384        if (!fileStream->is_open()) {
1385          fileStream->open(inputFileName.c_str());
1386        }
1387        string childSequence = CigarCoilUtilities::getDataAtFilePosition(
      childRead->getSequencePos(), childRead->getSequenceLength(), fileStream)
      ;
1388        if (!fileStream->is_open()) {
1389          fileStream->open(inputFileName.c_str());
1390        }
1391        string parentSequence = CigarCoilUtilities::getDataAtFilePosition(
      parentRead->getSequencePos(), parentRead->getSequenceLength(),
      fileStream);
1392        WagnerFischerMatrix matrix = WagnerFischerMatrix(&parentSequence, &
      childSequence);
1393        string cigarLine = matrix.getCigar();
1394
1395        sstream << cigarLine << "\n";
1396
1397      }
1398
1399    }
1400
1401    outputFile << sstream.rdbuf();
1402    outputFile.close();
1403  }
1404
1405
1406  void DNAFileWrapper::decode(const char *decodedFileName) {
1407
1408    if (fileType != DNAFileType::CIGARCOIL)
1409      return;
1410
1411    stack<Read> readStack = stack<Read>();
1412    unsigned int lineNumber = 0;
1413    std::ifstream fileStream;
1414    fileStream.open(myFileName.c_str());
1415
1416    string idLine = "";
1417    string cigarLine = "";
1418    string qualityLine = "";
1419    string rootSequenceLine = "";
1420
1421    const size_t bufferSize = sizeof(int);
1422    char buffer[bufferSize];
```

```
1423
1424  streamsize amountRead = 0;
1425
1426  fileStream.seekg(readPositions[root], ios_base::beg);
1427
1428  getline(fileStream, rootSequenceLine);
1429
1430  decodedReads.insert(root, rootSequenceLine);
1431
1432  string encodedIdsAndQualityScores = CigarCoilUtilities::
       createTemporaryFile();
1433  string decodedSequences = CigarCoilUtilities::createTemporaryFile();
1434
1435  ofstream decodedSequencesStream(decodedSequences, std::ofstream::out);
1436
1437  stringstream sstream;
1438
1439  for (size_t child = 0; child < parentArray.size(); child++) {
1440
1441    sstream.seekp(0, ios::end);
1442    stringstream::pos_type streamLength = sstream.tellp();
1443
1444    if (streamLength > 5000000) {
1445      decodedSequencesStream << sstream.rdbuf();
1446      // clear stream content
1447      sstream.str(string());
1448    }
1449
1450    if (child != root) {
1451
1452      fileStream.seekg(readPositions[child], ios_base::beg);
1453      getline(fileStream, cigarLine);
1454
1455      string uncompressedSequence = cigarCoilFileAccess(child, cigarLine, &
       fileStream);
1456
1457      // write decoded content to output file
1458      sstream << uncompressedSequence << "\n";
1459    }
1460    else {
1461      sstream << rootSequenceLine << "\n";
1462    }
1463  }
1464
1465  decodedSequencesStream << sstream.rdbuf();
1466  decodedSequencesStream.close();
1467
1468  sstream.str(string());
1469
1470  ofstream encodedIdAndQualityScoresStream;
1471  encodedIdAndQualityScoresStream.open(encodedIdsAndQualityScores.c_str(),
       ios_base::binary);
1472
1473  fileStream.close();
1474  fileStream.open(myFileName.c_str(), ios_base::binary);
1475  fileStream.seekg(idQualityStart, ios_base::beg);
1476
1477  // rest of file should be zpaq
```

```
1478    do {
1479      sstream.seekp(0, ios::end);
1480      stringstream::pos_type streamLength = sstream.tellp();
1481
1482      if (streamLength > 5000000) {
1483        encodedIdAndQualityScoresStream << sstream.rdbuf();
1484        // clear stream content
1485        sstream.str(string());
1486      }
1487
1488      sstream << (char)fileStream.get();
1489    } while (!fileStream.eof());
1490
1491    encodedIdAndQualityScoresStream << sstream.rdbuf();
1492    encodedIdAndQualityScoresStream.close();
1493
1494    string decodedIdAndQualityScoresFile = CigarCoilUtilities::decodeZpaq(
       encodedIdsAndQualityScores.c_str());
1495
1496    ofstream outputStream(decodedFileName, std::ofstream::out);
1497    ifstream sequencesStream(decodedSequences.c_str());
1498    ifstream idAndQualityScoresStream(decodedIdAndQualityScoresFile);
1499
1500    string firstLine = "";
1501    string secondLine = "";
1502
1503    getline(idAndQualityScoresStream, firstLine);
1504    getline(idAndQualityScoresStream, secondLine);
1505
1506    bool isFASTA = (secondLine.at(0) == '@' || secondLine.at(0) == '>');
1507
1508    idAndQualityScoresStream.seekg(0, ios_base::beg);
1509
1510    string sequenceLine = "";
1511
1512    for (size_t i = 0; i < parentArray.size(); i++) {
1513      getline(sequencesStream, sequenceLine);
1514      if (isFASTA) {
1515        getline(idAndQualityScoresStream, idLine);
1516        outputStream << idLine << "\n" << sequenceLine << "\n";
1517      }
1518      else {
1519        getline(idAndQualityScoresStream, idLine);
1520        getline(idAndQualityScoresStream, qualityLine);
1521        outputStream << idLine << "\n" << sequenceLine << "\n" << "+\n" <<
       qualityLine << "\n";
1522      }
1523    }
1524
1525    idAndQualityScoresStream.close();
1526    sequencesStream.close();
1527
1528    remove(decodedSequences.c_str());
1529    remove(decodedIdAndQualityScoresFile.c_str());
1530
1531    outputStream.close();
1532  }
1533
```

```
1534 std::string DNAFileWrapper::fastQFileAccess(size_t i) {
1535   std::ifstream fileStream;
1536   fileStream.open(myFileName.c_str());
1537   fileStream.seekg(readPositions[i], ios::beg);
1538   std::string result;
1539   std::getline(fileStream, result);
1540   fileStream.close();
1541   return result;
1542 }
1543
1544 std::string DNAFileWrapper::fastAFileAccess(size_t i) {
1545   std::ifstream fileStream;
1546   fileStream.open(myFileName.c_str());
1547   fileStream.seekg(readPositions[i], ios::beg);
1548   std::string result;
1549   std::getline(fileStream, result);
1550   fileStream.close();
1551   return result;
1552 }
1553
1554 string DNAFileWrapper::cigarCoilFileAccess(size_t readNumber) {
1555
1556   if (decodedReads.isKnown(readNumber)) {
1557     // this read has already been requested before
1558     return decodedReads.at(readNumber);
1559   }
1560   else {
1561
1562     string childCigarString = "";
1563     string result = "";
1564
1565     ifstream stream(myFileName.c_str());
1566
1567     stream.seekg(readPositions[readNumber], ios_base::beg);
1568
1569     // this line is the sequence for the string
1570     getline(stream, childCigarString);
1571
1572     while (childCigarString == "") {
1573       if (!stream.is_open()) {
1574         printf("stream is not open\n");
1575         stream.open(myFileName.c_str());
1576       }
1577       if (stream.bad()) {
1578         printf("stream is bad\n");
1579       }
1580         if (stream.eof()) {
1581         printf("at end of file\n");
1582       }
1583       printf("file stream at %d\n", stream.tellg());
1584       stream.seekg(readPositions[readNumber], ios_base::beg);
1585       printf("file stream at %d after seeking to %d\n", stream.tellg(),
    readPositions[readNumber]);
1586       getline(stream, childCigarString);
1587       while (childCigarString == "") {
1588         getline(stream, childCigarString);
1589       }
1590
```

```
1591      }
1592
1593      stream.close();
1594
1595      if (readNumber != 0) {
1596        result = cigarCoilFileAccess(readNumber, childCigarString);
1597        //result = cigarCoilFileAccess(readNumber, childCigarString, &stream)
      ;
1598        //stream.close();
1599        return result;
1600      }
1601      else {
1602        //stream.close();
1603        return childCigarString;
1604      }
1605    }
1606
1607  }
1608
1609  string DNAFileWrapper::cigarCoilFileAccess(size_t readNumber, ifstream *
      fileStream) {
1610
1611    if (decodedReads.isKnown(readNumber)) {
1612      // this read has already been requested before
1613      return decodedReads.at(readNumber);
1614    }
1615    else {
1616
1617      string childCigarString = "";
1618      string result = "";
1619
1620      if (!fileStream || !fileStream->is_open() || fileStream->bad() || !
      fileStream->good()) {
1621        printf("closed\n");
1622      }
1623      else if (fileStream->eof()) {
1624        printf("at end of file\n");
1625      }
1626
1627      ifstream stream(myFileName.c_str());
1628      stream.seekg(readPositions[readNumber], ios_base::beg);
1629      fileStream->seekg(readPositions[readNumber], ios_base::beg);
1630
1631      // this line is the sequence for the string
1632      getline(*fileStream, childCigarString);
1633
1634      while (childCigarString == "") {
1635        getline(*fileStream, childCigarString);
1636        printf("file stream now at %d\n", fileStream->tellg());
1637      }
1638
1639      if (readNumber != root) {
1640        result = cigarCoilFileAccess(readNumber, childCigarString);
1641        //result = cigarCoilFileAccess(readNumber, childCigarString,
      fileStream);
1642        //fileStream->close();
1643        return result;
1644      }
```

```
1645      else {
1646        //fileStream->close();
1647        return childCigarString;
1648      }
1649    }
1650
1651  }
1652
1653  string DNAFileWrapper::cigarCoilFileAccess(size_t i, string
        childCigarString) {
1654
1655    if (decodedReads.isKnown(i)) {
1656      // this read has already been requested before
1657      return decodedReads.at(i);
1658    }
1659    else {
1660      unsigned parentNumber = parentArray[i];
1661      // recursively discover parents
1662      string parentSequence = cigarCoilFileAccess(parentNumber);
1663
1664      // decode child sequence relative to parent
1665      string decodedSequence = decodeChildSequenceRelativeToParent(&
        childCigarString, &parentSequence);
1666
1667      decodedReads.insert(i, decodedSequence);
1668
1669      return decodedSequence;
1670
1671    }
1672  }
1673
1674
1675  string DNAFileWrapper::cigarCoilFileAccess(size_t i, string
        childCigarString, ifstream *fileStream) {
1676
1677    if (decodedReads.isKnown(i)) {
1678      // this read has already been requested before
1679      return decodedReads.at(i);
1680    }
1681    else {
1682      unsigned parentNumber = parentArray[i];
1683      // recursively discover parents
1684      string parentSequence = cigarCoilFileAccess(parentNumber, fileStream);
1685
1686      // decode child sequence relative to parent
1687      string decodedSequence = decodeChildSequenceRelativeToParent(&
        childCigarString, &parentSequence);
1688
1689      decodedReads.insert(i, decodedSequence);
1690
1691      return decodedSequence;
1692    }
1693  }
1694
1695  std::string DNAFileWrapper::at(size_t i) {
1696    switch (fileType) {
1697    case DNAFileType::CIGARCOIL:
1698      return cigarCoilFileAccess(i);
```

```
1699      break;
1700    case DNAFileType::FASTA:
1701      return fastAFileAccess(i);
1702      break;
1703    case DNAFileType::FASTQ:
1704      return fastQFileAccess(i);
1705      break;
1706    case DNAFileType::SAM:
1707      printf("not implemented\n");
1708      break;
1709    }
1710  }
1711
1712  std::string DNAFileWrapper::operator[](size_t i) {
1713    return getElement(i);
1714  }
1715
1716  void DNAFileWrapper::updateReadSequence(size_t i, string sequence) {
1717
1718    string sequenceBeforeChange = cigarCoilFileAccess(i);
1719
1720    string parentSequence = cigarCoilFileAccess(parentArray.at(i));
1721
1722    string newCIGAR = WagnerFischerMatrix(&parentSequence, &sequence).
       getCigar();
1723
1724    string temporaryFileName = CigarCoilUtilities::createTemporaryFile();
1725
1726    ofstream temporaryFileStream(temporaryFileName.c_str(), std::ofstream::
       out);
1727
1728    // writes special identifying 4 bytes to signal that this file is a
       cigarcoil file
1729    temporaryFileStream.write(cigarFileMarker, 4);
1730
1731    // there should be no change to the parent array since the content of a
       read is being modified
1732    unsigned int* parents = &parentArray[0];
1733    temporaryFileStream.write(reinterpret_cast<char *>(parents), sizeof(int)
       * parentArray.size());
1734
1735    unsigned int arrayEnding = UINT32_MAX;
1736    temporaryFileStream.write(reinterpret_cast<char *>(&arrayEnding), sizeof(
       int));
1737
1738    ifstream oldFileStream(myFileName.c_str());
1739    oldFileStream.seekg(readPositions.at(0), ios_base::beg);
1740
1741    stringstream sstream;
1742
1743    //unordered_map<unsigned int, string> childrenSequences = unordered_map<
       unsigned int, string>();
1744
1745    string idLine = "";
1746    string sequenceLine = "";
1747    string qualityLine = "";
1748
1749    // populate childrensequences map
```

```
1750   for (size_t p = 0; p < parentArray.size(); p++) {
1751
1752     sstream.seekp(0, ios::end);
1753     stringstream::pos_type streamLength = sstream.tellp();
1754
1755     if (streamLength > 5000000) {
1756       temporaryFileStream << sstream.rdbuf();
1757       // clear stream content
1758       sstream.str(string());
1759     }
1760
1761     getline(oldFileStream, idLine);
1762     getline(oldFileStream, sequenceLine);
1763
1764     if (isFASTQ) {
1765       getline(oldFileStream, qualityLine);
1766     }
1767
1768     sstream << idLine << "\n";
1769     if (parentArray[p] == i) {
1770       string decodedChildSequence = decodeChildSequenceRelativeToParent(&
     sequenceLine, &sequenceBeforeChange);
1771       string newChildCigar = WagnerFischerMatrix(&sequence, &
     decodedChildSequence).getCigar();
1772       sstream << newChildCigar << "\n";
1773     }
1774     else if (p == i) {
1775       sstream << newCIGAR << "\n";
1776     }
1777     else {
1778       sstream << sequenceLine << "\n";
1779     }
1780
1781     if (isFASTQ) {
1782       sstream << qualityLine << "\n";
1783     }
1784   }
1785
1786   temporaryFileStream << sstream.rdbuf();
1787
1788   oldFileStream.close();
1789   temporaryFileStream.close();
1790
1791   // remove the file prior to this change
1792   remove(myFileName.c_str());
1793
1794   // rename the changed file to the old file's name
1795   rename(temporaryFileName.c_str(), myFileName.c_str());
1796
1797   // update this file wrapper
1798   DNAFileWrapper postEditFileWrapper = DNAFileWrapper((char *)myFileName.
     c_str());
1799   *this = postEditFileWrapper;
1800 }
1801
1802 DNAFileWrapper DNAFileWrapper::concatenate(DNAFileWrapper *childFile,
     string concatenatedFileName) {
1803   vector<unsigned int> baseParentArray = parentArray;
```

```
1804  vector<unsigned int> *childParentArray = &childFile->parentArray;
1805
1806  unsigned int initialNumberOfReadsInBaseFile = baseParentArray.size();
1807
1808  // The root of the base tree will be the root of the child tree.
1809  baseParentArray.push_back(root);
1810
1811  // add updated parent entries of child file to the original base parent
       array
1812  for (size_t i = 1; i < childParentArray->size(); i++) {
1813    baseParentArray.push_back(childParentArray->at(i) +
       initialNumberOfReadsInBaseFile);
1814  }
1815
1816  string cigarForChildRoot = WagnerFischerMatrix(&cigarCoilFileAccess(root)
       , &childFile->cigarCoilFileAccess(childFile->root)).getCigar();
1817
1818  // open the file streams
1819  ifstream baseFileStream;
1820  baseFileStream.open(myFileName.c_str());
1821  ifstream childFileStream;
1822
1823  childFileStream.open(childFile->myFileName.c_str());
1824
1825  ofstream concatenatedFileStream(concatenatedFileName.c_str(), std::
       ofstream::out);
1826
1827  // writes special identifying 4 bytes to signal that this file is a
       cigarcoil file
1828  concatenatedFileStream.write(cigarFileMarker, 4);
1829
1830  // writes the new parent array to the file
1831  unsigned int* parents = &baseParentArray[0];
1832  concatenatedFileStream.write(reinterpret_cast<char *>(parents), sizeof(
       int) * baseParentArray.size());
1833
1834  unsigned int arrayEnding = UINT32_MAX;
1835  concatenatedFileStream.write(reinterpret_cast<char *>(&arrayEnding),
       sizeof(int));
1836
1837  string currentLine = "";
1838
1839  baseFileStream.seekg(readPositions.at(0), ios_base::beg);
1840
1841  stringstream sstream;
1842
1843  while (baseFileStream) {
1844
1845    sstream.seekp(0, ios::end);
1846    stringstream::pos_type streamLength = sstream.tellp();
1847
1848    if (streamLength > 5000000) {
1849      concatenatedFileStream << sstream.rdbuf();
1850      // clear stream content
1851      sstream.str(string());
1852    }
1853
1854    getline(baseFileStream, currentLine);
```

```
1855
1856      if (currentLine.length() > 0)
1857        sstream << currentLine << "\n";
1858
1859    }
1860
1861    concatenatedFileStream << sstream.rdbuf();
1862    // clear stream content
1863    sstream.str(string());
1864
1865    string childRootIdLine = "";
1866
1867
1868    childFileStream.seekg(childFile->readPositions.at(0), ios_base::beg);
1869    getline(childFileStream, childRootIdLine);
1870    sstream << childRootIdLine << "\n";
1871    sstream << cigarForChildRoot << "\n";
1872    if (childFile->isFASTQ) {
1873      string childQualityLine = "";
1874      // first getline is the childs sequence
1875      getline(childFileStream, childQualityLine);
1876      // this line should be the quality line
1877      getline(childFileStream, childQualityLine);
1878
1879      sstream << childQualityLine << "\n";
1880    }
1881
1882    childFileStream.seekg(childFile->readPositions.at(1), ios_base::beg);
1883
1884    while (childFileStream) {
1885
1886      sstream.seekp(0, ios::end);
1887      stringstream::pos_type streamLength = sstream.tellp();
1888
1889      if (streamLength > 5000000) {
1890        concatenatedFileStream << sstream.rdbuf();
1891        // clear stream content
1892        sstream.str(string());
1893      }
1894
1895      getline(childFileStream, currentLine);
1896
1897      if (currentLine.length() > 0)
1898        sstream << currentLine << "\n";
1899
1900    }
1901
1902    concatenatedFileStream << sstream.rdbuf();
1903
1904    baseFileStream.close();
1905    childFileStream.close();
1906    concatenatedFileStream.close();
1907
1908    DNAFileWrapper result = DNAFileWrapper((char *)concatenatedFileName.c_str
       ());
1909
1910    return result;
1911  }
```

```
1912
1913 string DNAFileWrapper::decodeChildSequenceRelativeToParent(const string *
       childCigar,const string *parentSequence) {
1914   unsigned char cigarOperationBuffer[2];
1915   string result = "";
1916   size_t positionInParentSequence = 0;
1917
1918   for (size_t cigarSize = 0; cigarSize < childCigar->size(); cigarSize +=
       2) {
1919
1920     cigarOperationBuffer[0] = childCigar->at(cigarSize);
1921     cigarOperationBuffer[1] = childCigar->at(cigarSize + 1);
1922
1923     CigarOperation currentOperation = CigarOperation(cigarOperationBuffer);
1924
1925     if (currentOperation.isMatch()) {
1926       unsigned int matchLength = currentOperation.getValueNumeric();
1927
1928       if (positionInParentSequence + matchLength > parentSequence->length()
       ) {
1929
1930         int difference = (matchLength + positionInParentSequence) -
       parentSequence->length();
1931
1932         while (positionInParentSequence < parentSequence->length()) {
1933           result += parentSequence->at(positionInParentSequence);
1934           positionInParentSequence++;
1935         }
1936       }
1937       else {
1938         result += parentSequence->substr(positionInParentSequence,
       matchLength);
1939         positionInParentSequence += matchLength;
1940       }
1941
1942
1943
1944     }
1945     else if (currentOperation.isDeletion()) {
1946       unsigned int deletionLength = currentOperation.getValueNumeric();
1947       positionInParentSequence += deletionLength;
1948     }
1949     else if (currentOperation.isSubstitution()) {
1950       string substitution = currentOperation.getValueString();
1951       positionInParentSequence += substitution.size();
1952       result += substitution;
1953     }
1954     else if (currentOperation.isInsertion()) {
1955       string insertion = currentOperation.getValueString();
1956       result += insertion;
1957     }
1958     else {
1959       printf("invalid operation detected\n");
1960     }
1961
1962   }
1963
1964   return result;
```

```
1965 }
1966
1967 void DNAFileWrapper::initialize() {
1968   if (!isInitialized) {
1969
1970     string s = "";
1971
1972     // store read positions for random access
1973     if (fileType == FASTA || fileType == FASTQ) {
1974       readPositions.clear();
1975       ifstream fileStream(myFileName.c_str());
1976       while (fileStream) {
1977         getline(fileStream, s);
1978         readPositions.push_back(fileStream.tellg());
1979         getline(fileStream, s);
1980         if (fileType == FASTQ) {
1981           getline(fileStream, s);
1982           getline(fileStream, s);
1983         }
1984       }
1985     }
1986
1987
1988     isInitialized = true;
1989     cachedElements.clear();
1990     for (size_t i = 0; i < numberOfElementsToCache; i++) {
1991       cachedElements.push_back(at(i));
1992     }
1993     idOfFirstElementCached = 0;
1994     idOfLastElementCached = cachedElements.size() - 1;
1995
1996
1997
1998   }
1999 }
2000
2001 int DNAFileWrapper::getBestActionForAState(size_t state) {
2002   int bestAction = 0;
2003   double bestValue = -1 * FLT_MAX;
2004   for (size_t j = 0; j < numberOfActions; j++) {
2005     if (stateActionPairs[state][j] > bestValue) {
2006       bestValue = stateActionPairs[state][j];
2007       bestAction = j;
2008     }
2009   }
2010   return bestAction;
2011 }
2012
2013 void DNAFileWrapper::fetchElementsForward(size_t start, size_t
    numberOfElements) {
2014   unsigned int previousIdOfFirstElementCached = idOfFirstElementCached;
2015   unsigned int previousIdOfLastElementCached = idOfLastElementCached;
2016
2017   // fetch new additions
2018   vector<string> newAdditions = vector<string>();
2019
2020   for (size_t i = 0; i < numberOfElements; i++) {
2021     long position = i + previousIdOfLastElementCached;
```

```
2022
2023      if (position < readPositions.size()) {
2024        newAdditions.push_back(at(position));
2025      }
2026    }
2027
2028    // move cached elements forward to make room for new additions
2029    for (size_t j = newAdditions.size(); j < numberOfElementsToCache; j++) {
2030      cachedElements[j - newAdditions.size()] = cachedElements[j];
2031    }
2032
2033    // add new additions to end of cached elements
2034    size_t positionInNewAdditions = 0;
2035    for (long k = numberOfElementsToCache - newAdditions.size(); k <
       numberOfElementsToCache; k++) {
2036      cachedElements[k] = newAdditions[positionInNewAdditions++];
2037    }
2038
2039    // move window of cached elements forward
2040    idOfFirstElementCached += newAdditions.size();
2041    idOfLastElementCached += newAdditions.size();
2042  }
2043
2044  void DNAFileWrapper::fetchElementsBackward(size_t start, size_t
       numberOfElements) {
2045    unsigned int previousIdOfFirstElementCached = idOfFirstElementCached;
2046    unsigned int previousIdOfLastElementCached = idOfLastElementCached;
2047
2048    // fetch new additions
2049    vector<string> newAdditions = vector<string>();
2050
2051    for (size_t i = 1; i <= numberOfElements; i++) {
2052      long position = previousIdOfFirstElementCached - i;
2053      if (position >= 0) {
2054        newAdditions.push_back(at(position));
2055      }
2056    }
2057
2058    // shift elements up to make room for new additions
2059    for (long j = numberOfElementsToCache - 1; j >= newAdditions.size(); j--)
       {
2060      if (j < 0) {
2061        break;
2062      }
2063      cachedElements[j] = cachedElements[j - newAdditions.size()];
2064    }
2065
2066    // insert new additions to first elements of cache
2067    size_t positionInNewAdditions = 0;
2068    for (long k = newAdditions.size() - 1; k >= 0; k--) {
2069      cachedElements[k] = newAdditions[positionInNewAdditions++];
2070    }
2071
2072    // move window of cached elements backward
2073    idOfFirstElementCached -= newAdditions.size();
2074    idOfLastElementCached -= newAdditions.size();
2075
2076  }
```

```
2077
2078 // map user's requested element to a particular state
2079 unsigned int DNAFileWrapper::determineState(unsigned int i) {
2080   if (i < idOfFirstElementCached) {
2081     return 11;
2082   }
2083   else if (i >= idOfLastElementCached) {
2084     return 12;
2085   }
2086   else {
2087     size_t progress = i - idOfFirstElementCached;
2088     double percent = (progress * 1.0) / numberOfElementsToCache;
2089     return (int)floor(percent);
2090   }
2091 }
2092
2093 void DNAFileWrapper::qLearningPrediction(size_t requestedId) {
2094
2095   // state = % in cached reads [0,10);[10,20)...
2096   // state 11 = outside range backward
2097   // state 12 = outside range forward
2098
2099   // state 0: i + 1 is unknown other is known
2100   // state 1: i - 1 is unknown other is known
2101   // state 2: i + 1 and i - 1  are unknown
2102   // state 3: prev diff + unknown
2103   // state 4: prev diff - unknown
2104   // state 5: prev diff + and - are unknown
2105   // state 6: is start
2106   // state 7: is end
2107
2108   // action0: do nothing
2109   // action1 - 10: reverse window 10% multiple
2110   // action11 - 20: advance window 10% multiple
2111
2112   unsigned int currentState = determineState(requestedId);
2113
2114   unsigned int currentPosition = 0;
2115
2116   // find best action to take from policy matrix
2117   unsigned int bestAction = 0;
2118   float randomValue = static_cast <float> (rand()) / static_cast <float> (
      RAND_MAX);
2119   if (randomValue < EPSILON) {
2120     bestAction = rand() % numberOfActions;
2121
2122   }
2123   else {
2124
2125     float bestValue = stateActionPairs[currentState][0];
2126
2127     for (unsigned int i = 1; i < numberOfActions; i++) {
2128       if (stateActionPairs[currentState][i] > bestValue) {
2129         bestAction = i;
2130         bestValue = stateActionPairs[currentState][i];
2131       }
2132     }
2133   }
```

```
2134
2135  unsigned int previousState = currentState;
2136
2137  // take action and determine next state
2138  if (bestAction == 0) {
2139    // do nothing
2140  }
2141  else {
2142    if (bestAction > 0 && bestAction <= 10) {
2143      double percent = bestAction / 10.0;
2144      unsigned int numberToFetch = round(numberOfElementsToCache * percent)
      ;
2145      fetchElementsBackward(idOfFirstElementCached, numberToFetch);
2146    }
2147    else if (bestAction > 10 && bestAction <= 20) {
2148
2149      double percent = (20 - bestAction) / 10.0;
2150      unsigned int numberToFetch = round(numberOfElementsToCache * percent)
      ;
2151      fetchElementsForward(idOfLastElementCached, numberToFetch);
2152    }
2153    currentState = determineState(requestedId);
2154  }
2155
2156  // determine reward
2157  float reward = 0.0;
2158
2159  switch (currentState) {
2160  case 0:
2161    // first 10%
2162    reward = 0.1;
2163    break;
2164  case 1:
2165    // first 20%
2166    reward = 0.2;
2167    break;
2168  case 2:
2169    // first 30%
2170    reward = 0.3;
2171    break;
2172  case 3:
2173    // first 40%
2174    reward = 0.4;
2175    break;
2176  case 4:
2177    // first 50%
2178    reward = 0.45;
2179    break;
2180  case 5:
2181    // first 60%
2182    reward = 0.5;
2183    break;
2184  case 6:
2185    // first 70%
2186    reward = 0.45;
2187    break;
2188  case 7:
2189    // first 80%
```

```
2190      reward = 0.4;
2191      break;
2192    case 8:
2193      // first 90%
2194      reward = 0.3;
2195      break;
2196    case 9:
2197      // first 100%
2198      reward = 0.2;
2199      break;
2200    case 10:
2201    case 11:
2202    case 12:
2203      // out of current bounds of cache
2204      reward = -10;
2205      break;
2206    }
2207
2208    float prevQValue = stateActionPairs[previousState][bestAction];
2209
2210    float bestNextQValue = stateActionPairs[currentState][0];
2211
2212    for (unsigned int i = 1; i < numberOfActions; i++) {
2213      if (stateActionPairs[currentState][i] > bestNextQValue) {
2214        bestAction = i;
2215        bestNextQValue = stateActionPairs[currentState][i];
2216      }
2217    }
2218
2219    float update = (1 - ALPHA) * prevQValue + (ALPHA * (reward + GAMMA *
       bestNextQValue));
2220    stateActionPairs[previousState][bestAction] += update;
2221
2222  }
2223
2224  std::string DNAFileWrapper::getElement(size_t element) {
2225
2226    if (fileType == CIGARCOIL) {
2227      return cigarCoilFileAccess(element);
2228    }
2229    else if (fileType == FASTA) {
2230      return fastAFileAccess(element);
2231    }
2232    else if (fileType == FASTQ) {
2233      return fastQFileAccess(element);
2234    }
2235
2236    initialize();
2237
2238    clock_t cacheStart = clock();
2239
2240
2241    while (!(element >= idOfFirstElementCached && element <=
       idOfLastElementCached)) {
2242
2243
2244      qLearningPrediction(element);
2245
```

```
2246      if (element >= idOfFirstElementCached && element <=
          idOfLastElementCached) {
2247        double cacheDuration = ((clock() - cacheStart) / (double)
          CLOCKS_PER_SEC);
2248
2249        ofstream cacheTime("C:\\projectInputFiles\\cacheTime.csv", ios::app);
2250        cacheTime << cacheDuration << "\n";
2251        cacheTime.close();
2252
2253      }
2254    }
2255    return cachedElements[element - idOfFirstElementCached];
2256  }
2257
2258  /* Destructor for DNAFileParser */
2259  DNAFileWrapper::~DNAFileWrapper() {
2260  }
2261
2262
2263  void DNAFileWrapper::reconstructCompressedFile(const char *
          uncompressedFileName, const char *reconstructedCompressedFileName, const
           char *compressedIdAndQualityFileName, bool isBeingMemoryConservative) {
2264    DNAFileWrapper uncompressedFile = DNAFileWrapper(uncompressedFileName);
2265
2266    ofstream outputFile(reconstructedCompressedFileName, std::ofstream::out);
2267
2268    vector<string> sequences = vector<string>();
2269
2270    // writes special identifying 4 bytes to signal that this file is a
          cigarcoil file
2271    outputFile.write(cigarFileMarker, 4);
2272
2273    outputFile.write(reinterpret_cast<char *>(&parentArray[0]), sizeof(int) *
          parentArray.size());
2274
2275    unsigned int arrayEnding = UINT32_MAX;
2276    outputFile.write(reinterpret_cast<char *>(&arrayEnding), sizeof(int));
2277
2278    vector<streampos> positionsInOriginalFile = vector<streampos>();
2279    ifstream uncompressedFileStream(uncompressedFileName);
2280    string temp = "";
2281
2282    for (size_t i = 0; i < parentArray.size(); i++) {
2283      getline(uncompressedFileStream, temp);
2284
2285      if (isBeingMemoryConservative) {
2286        positionsInOriginalFile.push_back(uncompressedFileStream.tellg());
2287      }
2288      getline(uncompressedFileStream, temp);
2289      if (!isBeingMemoryConservative) {
2290        sequences.push_back(temp);
2291      }
2292      if (uncompressedFile.isFASTQ) {
2293        getline(uncompressedFileStream, temp);
2294        getline(uncompressedFileStream, temp);
2295      }
2296    }
2297
```

```
2298  uncompressedFileStream.close();
2299  if (isBeingMemoryConservative)
2300    uncompressedFileStream.open(uncompressedFileName);
2301
2302  stringstream sstream;
2303
2304
2305
2306  for (size_t i = 0; i < parentArray.size(); i++) {
2307
2308    sstream.seekp(0, ios::end);
2309    stringstream::pos_type streamLength = sstream.tellp();
2310
2311    if (streamLength > 5000000) {
2312      outputFile << sstream.rdbuf();
2313      // clear stream content
2314      sstream.str(string());
2315    }
2316
2317    string childSequence = "";
2318    string parentSequence = "";
2319
2320    if (isBeingMemoryConservative) {
2321      if (!uncompressedFileStream.is_open()) {
2322        uncompressedFileStream.open(uncompressedFileName);
2323      }
2324      uncompressedFileStream.seekg(positionsInOriginalFile[i], ios::beg);
2325      getline(uncompressedFileStream, childSequence);
2326    }
2327    else {
2328      childSequence = sequences[i];
2329    }
2330
2331
2332    if (i == root) {
2333
2334      // write root explicitly
2335      sstream << childSequence << "\n";
2336    }
2337    else {
2338
2339      unsigned int parentId = parentArray[i];
2340
2341      if (isBeingMemoryConservative) {
2342        if (!uncompressedFileStream.is_open()) {
2343          uncompressedFileStream.open(uncompressedFileName);
2344        }
2345        uncompressedFileStream.seekg(positionsInOriginalFile[parentId], ios
    ::beg);
2346        getline(uncompressedFileStream, parentSequence);
2347      }
2348      else {
2349        parentSequence = sequences[parentId];
2350      }
2351
2352      WagnerFischerMatrix matrix = WagnerFischerMatrix(&parentSequence, &
    childSequence);
2353      string cigarLine = matrix.getCigar();
```

```
2354
2355        sstream << cigarLine << "\n";
2356
2357      }
2358
2359    }
2360
2361    outputFile << sstream.rdbuf();
2362    outputFile.close();
2363
2364    string *myFiles = new string[2];
2365    myFiles[0] = string(reconstructedCompressedFileName);
2366    myFiles[1] = string(compressedIdAndQualityFileName);
2367    concatenateFilesTogether(myFiles, 2, string(
       reconstructedCompressedFileName) + ".final");
2368    delete[] myFiles;
2369
2370  }
```

Appendix I

Decoded Reads

I.1 Header File

```
#ifndef DECODED_READS_H
#define DECODED_READS_H

#include<vector>
#include<string>

const unsigned int numberOfGenerations = 10;
const unsigned int numberOfInsertionsBeforeCleanup = 10000000;

// class for storing previously decoded reads. After a threshold is reached
    only keep most frequently accessed elements
class DecodedReads {
private:
  // vector of stored reads
  std::vector<std::string> reads;
  // track popularity of a particular read
  std::vector<unsigned int> readGenerations;
  //// number of reads to store
  size_t _numberOfReads;
  // tracks how many insertions have occurred since last cleanup
  unsigned int currentInsertionNumber;
public:
  // constructurs
  DecodedReads();
  DecodedReads(size_t numberOfReads);
  // is the requested element stored in this structure?
  bool isKnown(size_t i) const;
  // insert an element into this structure
  void insert(size_t i, std::string s);
  // returns the requested element
  std::string at(size_t i);
  ~DecodedReads();
};

#endif // !DECODED_READS_H
```

I.2 Definitions

```
#include "DecodedReads.h"
```

```
2
3  DecodedReads::DecodedReads() {
4    _numberOfReads = 0;
5    reads = std::vector<std::string>();
6    readGenerations = std::vector<unsigned int>();
7    currentInsertionNumber = 0;
8  }
9
10 DecodedReads::DecodedReads(size_t numberOfReads) {
11   _numberOfReads = numberOfReads;
12   reads = std::vector<std::string>();
13   for (size_t i = 0; i < numberOfReads; i++) {
14     reads.push_back("");
15   }
16   readGenerations = std::vector<unsigned int>();
17   for (size_t j = 0; j < numberOfReads; j++) {
18     readGenerations.push_back(0);
19   }
20   currentInsertionNumber = 0;
21 }
22
23 // is the requested element stored in this structure
24 bool DecodedReads::isKnown(size_t i) const{
25   if (_numberOfReads <= i)
26     return false;
27   return reads[i] != "";
28 }
29
30 // adds element to the set of decoded reads
31 void DecodedReads::insert(size_t i, std::string s) {
32   reads[i] = s;
33   readGenerations[i] = numberOfInsertionsBeforeCleanup / 3;
34   // perform memory cleanup if threshold exceeded
35   if (currentInsertionNumber++ > numberOfInsertionsBeforeCleanup) {
36     for (size_t i = 0; i < _numberOfReads; i++) {
37       if (readGenerations[i] < numberOfInsertionsBeforeCleanup / 2) {
38         reads[i] = "";
39         readGenerations[i] = 0;
40       }
41       else {
42         readGenerations[i] = readGenerations[i] - (
    numberOfInsertionsBeforeCleanup / 2);
43       }
44     }
45     currentInsertionNumber = 0;
46   }
47 }
48
49 std::string DecodedReads::at(size_t i) {
50   readGenerations[i] = readGenerations[i] + 1;
51   return reads[i];
52 }
53
54 DecodedReads::~DecodedReads() {
55   reads.clear();
56   readGenerations.clear();
57 }
```