ALGORITHM OPTIMIZATIONS IN GENOMIC ANALYSIS USING ENTROPIC DISSECTION

Jacob R. Danks

Thesis Prepared for the Degree of

MASTER OF SCIENCE

UNIVERSITY OF NORTH TEXAS

August 2015

APPROVED:

Kathleen Swigger, Major Professor
Rajeev Azad, Committee Member
Renee Bryce, Committee Member
Barrett Bryant, Chair of the Department of Computer Science and Engineering
Costas Tsatsoulis, Dean of the College of Engineering and Interim Dean of the Toulouse Graduate School
Danks, Jacob R. *Algorithm Optimizations in Genomic Analysis Using Entropic Dissection*. Master of Science (Computer Science), August 2015, 37 pp., 8 figures, 21 numbered references.

In recent years, the collection of genomic data has skyrocketed and databases of genomic data are growing at a faster rate than ever before. Although many computational methods have been developed to interpret these data, they tend to struggle to process the ever increasing file sizes that are being produced and fail to take advantage of the advances in multi-core processors by using parallel processing. In some instances, loss of accuracy has been a necessary trade off to allow faster computation of the data.

This thesis discusses one such algorithm that has been developed and how changes were made to allow larger input file sizes and reduce the time required to achieve a result without sacrificing accuracy. An information entropy based algorithm was used as a basis to demonstrate these techniques. The algorithm dissects the distinctive patterns underlying genomic data efficiently requiring no a priori knowledge, and thus is applicable in a variety of biological research applications. This research describes how parallel processing and object-oriented programming techniques were used to process larger files in less time and achieve a more accurate result from the algorithm. Through object oriented techniques, the maximum allowable input file size was significantly increased from 200 mb to 2000 mb. Using parallel processing techniques allowed the program to finish processing data in less than half the time of the sequential version. The accuracy of the algorithm was improved by reducing data loss throughout the algorithm. Finally, adding user-friendly options enabled the program to use requests more effectively and further customize the logic used within the algorithm.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF FIGURES</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAPTER 1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1. Segmentation Clustering Algorithm</td>
<td>2</td>
</tr>
<tr>
<td>1.2. Divergence Measures</td>
<td>2</td>
</tr>
<tr>
<td>1.3. Recursive Segmentation and Clustering</td>
<td>4</td>
</tr>
<tr>
<td>1.4. Effectiveness of the Algorithm</td>
<td>5</td>
</tr>
<tr>
<td>1.5. Drawbacks</td>
<td>7</td>
</tr>
<tr>
<td>CHAPTER 2. RELATED WORK</td>
<td>8</td>
</tr>
<tr>
<td>2.1. Object Oriented Architecture</td>
<td>8</td>
</tr>
<tr>
<td>2.2. Parallel Processing</td>
<td>9</td>
</tr>
<tr>
<td>2.3. Replacing Recursion with Iteration</td>
<td>9</td>
</tr>
<tr>
<td>CHAPTER 3. METHODS</td>
<td>11</td>
</tr>
<tr>
<td>3.1. Methods Used to Increase File Size</td>
<td>11</td>
</tr>
<tr>
<td>3.1.1. Refactor Recursive Methods</td>
<td>11</td>
</tr>
<tr>
<td>3.1.2. Reduce Stack Frame Size</td>
<td>14</td>
</tr>
<tr>
<td>3.2. Methods Used to Increase Speed</td>
<td>14</td>
</tr>
<tr>
<td>3.2.1. Parallel Processing</td>
<td>15</td>
</tr>
<tr>
<td>3.3. Methods Used to Increase Accuracy</td>
<td>16</td>
</tr>
<tr>
<td>3.3.1. Reducing Data Loss</td>
<td>17</td>
</tr>
<tr>
<td>3.3.2. Best Match Clustering</td>
<td>18</td>
</tr>
</tbody>
</table>
3.3.3. Global Oligomer Weights ................................................................. 19
3.3.4. Checking Statistical Significance .................................................. 21
3.4. Additional Benefits of an Object Oriented Architecture ................. 22
  3.4.1. Sustainability ................................................................................ 23
  3.4.2. Extendibility ................................................................................. 25
  3.4.3. Ease of Making Changes to the User Interface ......................... 25
  3.4.4. Ease of Making Changes to the Logic ......................................... 26

CHAPTER 4. EXPERIMENTAL RESULTS AND DISCUSSION ...................... 27
  4.1. Increased File Size ............................................................................ 27
  4.2. Parallel Processing Speedup ............................................................. 27
  4.3. Accuracy .......................................................................................... 30

CHAPTER 5. CONCLUSIONS .................................................................. 33

REFERENCES .......................................................................................... 36
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Recursive Implementation</td>
<td>13</td>
</tr>
<tr>
<td>3.2</td>
<td>Iterative Implementation</td>
<td>13</td>
</tr>
<tr>
<td>3.3</td>
<td>Global Oligomer Weight Example</td>
<td>20</td>
</tr>
<tr>
<td>4.1</td>
<td>Runtime Comparison</td>
<td>28</td>
</tr>
<tr>
<td>4.2</td>
<td>Segmentation Step Comparison</td>
<td>29</td>
</tr>
<tr>
<td>4.3</td>
<td>Total Runtime Comparison</td>
<td>30</td>
</tr>
<tr>
<td>4.4</td>
<td>Cluster Purity Using One Percent Donor Data</td>
<td>31</td>
</tr>
<tr>
<td>4.5</td>
<td>Cluster Distribution Using Ten Percent Donor Data</td>
<td>32</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

Many algorithms have been developed for the purpose of genomic analysis and while they provide valuable insight, the problem of genomic analysis is not a static one. Because of the increasing size of the datasets that are being produced and the advancements in computer hardware, some of the techniques used in these algorithms could be improved if they were given another look in light of current requirements and available tools. This paper will show how using an object oriented architecture, parallel processing, and iterative methods in place of recursive ones, can provide improvements in the file sizes that can be used for input, the runtime of the algorithms, and the accuracy of the output.

This paper draws from the previous work of Azad and Li, which describes an algorithm that uses Markovian Jensen-Shannon divergence (MJSD) [2]. The MJSD algorithm uses segmentation and clustering to identify underlying similarities shared by genomic segments that are disparate in terms of proximity. No a priori knowledge about the genomic data is required, allowing this algorithm to be used on a wide variety of different genomic sequences from those that are well researched to more recently acquired sequences. The algorithm was initially implemented in the C programming language and was written procedurally. This paper explores how an object oriented architecture, parallel programming techniques, and alternatives to recursion can be used to allow larger file sizes to be processed, increase the speed at which the processing is accomplished, and allow for more accurate results.
1.1 Segmentation Clustering Algorithm

To develop the MJSD algorithm, Azad and Li used an entropy-based algorithm in which a genomic sequence is first segmented using the Jensen-Shannon divergence, and after which the segments are clustered according to the genomic features that are present in each segment [2]. The implementation allows the user to specify Markov order 0, 1, or 2 resulting in a measure with 4, 16, or 64 features for each genomic segment. These features are a count of the oligomer frequency that occurs within the segment and are used in comparing the segments to each other within the algorithm. The result of running the algorithm is a set of clusters that contain similar segments of genomic data.

The algorithm uses four steps, and individual confidence levels for each step are entered as parameters to the algorithm. The first step is segmentation where the entire genomic sequence is broken into small segments, followed by the clustering step where only contiguous segments are considered for combining, then a clustering step that compares non-contiguous clusters of segments occurs next, and finally the non-contiguous clustering step is repeated.

1.2 Divergence Measures

The divergence between two genomic segments $p_1$ and $p_2$, is measured using the Jensen-Shannon divergence measure, $D(p_1,p_2)$, defined as [9,13]:

\[
D(p_1, p_2) = H(\pi_1 p_1 + \pi_2 p_2) - \pi_1 H(p_1) - \pi_2 H(p_2)
\]

where $H(.) = -\sum_x p_i(x) \log_2 p_i(x)$ is the Shannon entropy function, $\pi_i$ is the weight factor assigned to $p_i$, $\sum \pi_i = 1$. The importance associated with probability distribution $p_i$ is represented by $\pi_i$ and for each probability distribution $p_i$, $\sum x p_i(x) = 1$. In this algorithm, the segment $S_i$ has length $l_i$ and is made up of a specific set of oligomers. The occurrence
frequency $f_x$ of each oligomer within the sequence are represented by $p_i(x)$. If the weight factor $\pi_i$ is normalized using the length of the segment, $l_i$, the Jensen–Shannon divergence between our two sequences $S_1$ and $S_2$ can be calculated using the formula:

$$D(S_1, S_2) = H(S) - \left( \frac{l_1}{L} H(S_1) + \frac{l_2}{L} H(S_2) \right).$$

In this equation, $L = l_1 + l_2$, $S = S_1 \oplus S_2$, and $H(S) = -\sum f_x \log_2 f_x$.

The Jensen–Shannon divergence is usually used to measure the difference between distributions where the frequency of one character is not affected by the frequency or proximity of any others. In the domain of genomic data, though, the presence of one oligomer can greatly increase or decrease the occurrence of another, especially when they are in close proximity. Using a Markov chain model, this correlation can be used by reformulating the divergence measure to account for order of occurrence of symbols. The generalized Jensen–Shannon divergence measure for a Markov source of order $m$ is defined as [1, 21]:

$$D^m(p_1, p_2) = H^m(\pi_1 p_1 + \pi_2 p_2) - \pi_1 H^m(p_1) - \pi_2 H^m(p_2)$$

In this formula, the Shannon entropy function $H^m(.)$ uses the Markov order $m$:

$$H^m(p_i) = -\sum_w P(w) \sum_{x \in A} P(x|w) \log_2 P(x|w)$$

The symbol that succeeds string $w$ of $m$ symbols is represented by $x$, and the probability that a transition will be made from $w$ to $x$ is shown using $P(x|w)$. Using $P(w)$ as the probability of string $w$, the conditional entropy function $H^m(.)$ measures the information content when the occurrence of a symbol $i$ is only affected by $m$ preceding symbols. When Markov order zero is used, the formula is the same as the standard Jensen–Shannon divergence measure.
The generalized formula of the Jensen-Shannon divergence is defined as:

\[ D^m(S_1, S_2) = H^m(S) - \left( \frac{l_1}{L} H^m(S_1) + \frac{l_2}{L} H^m(S_2) \right) \]

In this formula, the conditional entropy for the segment \( S_i \) is defined as \( H^m(S_i) \). Using \( N(.) \) as the count of strings \( w \) and \( wx \) in the sequence \( S_i \), the values of transition and marginal probabilities can be estimated: \( P(w) \approx \frac{N(w)}{(l_i - m + 1)} \) and \( P(x|w) \approx \frac{N(w \oplus x)}{N(w)} \).

1.3 Recursive Segmentation and Clustering

The Jensen-Shannon divergence measure is one technique that is frequently used for determining similarity within the organizational structure of genomic sequences \([3, 5, 9, 17]\). This divergence measure can be generalized for use with Markov models such that it accounts for the order that short nucleotide sequences appear. By taking Markov models into account, this generalization of the Jensen-Shannon divergence measure has allowed for greater insight into how the organization of nucleotides affects the function of genomic sequences \([1, 21]\).

The segmentation process starts with the entire genomic sequence. The generalization of the Jensen-Shannon divergence is calculated for the left and right sides of all the possible segmentation points. Once all possible segmentation points have been calculated, the point of maximum divergence is checked against the segmentation confidence factor that the user supplied. If the divergence is greater than or equal to the confidence factor, the sequence is split and the segmentation continues recursively.

The algorithm then begins the clustering steps. First only contiguous clusters are evaluated and then combined because proximity is also an indication that oligomers may be related. As in the segmentation step, a confidence factor supplied by the user is used to
determine if the clusters are sufficiently similar before they are combined. Then the clusters are evaluated against non-contiguous clusters to allow genomic segments with similar features to be combined even though they are not close in proximity.

1.4 Effectiveness of the Algorithm

In order to prove the effectiveness of the MJSD algorithm, Azad and Li comparatively assessed it with four popular methods: the HMM-driven Bayesian method [6], the HMM based method [RHOM, (16)], the generalized Gibbs sampler-based Bayesian method [11, 12], and the optimization method [K-H segmentation, (8)]. In order to test the algorithm for effectiveness in cases when many times bacterial genomes have close to 60% native genes and the rest from multiple donors, the artificial chimeric *E. Coli* genomes was used. Ideally, the algorithms would be able to group the native regions into one cluster and the segments from each donor would be grouped into a homogeneous cluster. While there were some algorithms that performed well, Azad and Li found that none were able to identify 9 of the 10 donors except for the MJSD algorithm. Many of the algorithms also produced multiple clusters with *E. Coli* sequences in them, but the MJSD algorithm correctly produced one cluster with most of the *E. Coli* data. In addition, the input file needed to be broken into sections less than 60 kb in order for to be used in the HMM Bayesian method because of file size constraints. While the MJSD algorithm required no a priori knowledge, all of the other algorithms required some input from the user to determine how many donor clusters should be present.

Azad and Li also conducted tests using the genuine *Salmonella enterica typhi CT18* genome. Using a set of genomic islands that have been produced with high confidence by Vernikos and Parkhill using the HGT process [7], all methods were tested for their ability to
identify the compositionally distinct segments. All of the algorithms struggled to classify the island sections into the non-native clusters to varying degrees, and the MJSD method appeared to not only be more robust but was also more sensitive to changes in the confidence factors used when classifying the compositionally distinct regions in both artificial and genuine genomes[2].

Tests were also run by Azad and Li to see if the methods could distinguish between native and alien genomic data when genomic data from six strains of Salmonella were in the same input file with fine non-Salmonella taxa. The MJSD algorithm was able to combine 99% of the native segments into two native clusters and group the rest into seven non-native clusters of varying purity. The K-H segmentation method was able to produce one cluster each for native and non-native segments at K = 100, H = 2, but the Bayesian and RHOM methods were unable to distinguish between native and alien genomic data.

Tests were also conducted by Azad and Li on the MJSD algorithm alone to see how it would perform when given the task of differentiating host from parasite genomic data. The N. Gonorrhoeae bacterial pathogen was selected and the results of these tests were also promising. Thus the MJSD algorithm was shown to effectively group genomic regions that are representative of distinct evolutionary patterns by discovering the underlying structure within segments. This is made all the more appealing for use on newly sequenced and uncharacterized genomes by the fact that the algorithm does not require a priori knowledge about the genome.
1.5 Drawbacks

The initial implementation of the MJSD algorithm was only able to process files up to 200 mb, and larger files were required to be broken up into parts removing the opportunity for a segment at the beginning of the file to be clustered with a segment near the end. The desire was for the algorithm to be able to process files up to 1 Gb. The runtime was also problematic in that files nearing the file size limit took a restrictively large amount of time to run. Finally the accuracy of the program was sometimes compromised because some data was lost throughout the program runs when calculating divergence.
In order to address the issues of file size limitations, long runtimes, and accuracy concerns several methods have been introduced in the field of genomics. These include converting procedural programs to an object oriented architecture, the use of parallel processing, and using iterative methods as opposed to recursive ones in cases where many method calls will be required.

2.1 Object Oriented Architecture

Gupta et al. (2003) used an object oriented architecture to refactor existing sequential programs in the field of molecular modeling. They were motivated by the need for programmers to write efficient programs with more speed and, at the same time, increase the understandability, portability, and extensibility of the programs [14]. They found that by structuring the programs not around the tasks to be performed, but based on the data that the tasks would be manipulating, they were able to create reusable modules that were easy to understand and maintain. Not only did this lead to a reduction in errors, but it also reduced the development time required to produce new functionality. In addition to the improvements mentioned in this paper, there are plans to expand and adapt the Markovian Jensen-Shannon divergence (MJSD) algorithm to additional problems and datasets in the future. In order to make the code more understandable and to make any future work on the program quick, efficient, and straightforward, an object oriented architecture was chosen.
2.2 Parallel Processing

McKenna et al. (2010) developed a genome analysis toolkit (GATK) with the intent of making it easier for developers to write feature-rich, efficient, and robust analysis tools [15]. By separating the calculation steps from the work of moving data around, they were able to use both shared and distributed memory parallelization. In this way they were able to add CPU resources in order to reduce the runtime of an analysis. While a program cannot obtain a speed-up of more than the number of processors used due to communication time between processors, they were able to see a speed-up of nearly 12 times by using 12 processors when genotyping the NA12878 chromosome [15], meaning that very little time was consumed by communication.

Rognes et al. (2000) applied distributed memory parallelism in combination with other optimizations to the Smith-Waterman database to obtain a six fold speed-up over the previous fastest implementation [18] when using 8 processors. This implementation is very specific to the Smith-Waterman database and algorithm, even requiring some code to be written in Assembly language, but it does demonstrate the potential gains that can be obtained using parallel processing. The MJSD algorithm shared some common characteristics with the Smith-Waterman algorithm in that it required many small calculations that did not rely on each other for data, making parallel processing a natural choice to improve the speed at which files can be processed.

2.3 Replacing Recursion with Iteration

Liu and Stoller (1999) described several methods for optimizing algorithms by replacing recursive methods in programs with iterative methods. They described the steps of identifying
where the input can be incremented and how to derive an alternative computation order that increments through the input instead of using recursion. They show how this process not only eliminates the space that is required to store a stack frame in memory for each recursive method call, but also reduces the overhead time that is used to allocate and de-allocate the frames [14]. While Liu and Stoller saw dramatic improvements in their tests using tests with up to several hundred iterations, there is even greater potential in fields like genome analysis where there may be millions of these types of calls. The initial implementation of the MJSD algorithm suffered from difficulties with too many stack frames being held in memory leading to the choice to refactor the recursive methods into iterative ones as described by Liu and Stoller.
CHAPTER 3

METHODS

This paper examines the effectiveness of using an object oriented architecture, parallel processing, and the removal of recursive methods from an algorithm that uses the Jenson-Shannon divergence to segment and cluster a genomic sequence in order to remove many of the obstacles that this and similar algorithms face, such as file size limitations, long runtimes, and accuracy trade-offs.

3.1 Methods Used to Increase File Size

The initial implementation of the algorithm was limited to running files smaller than 200 mb, while larger files (up to 1 Gb) were being processed by breaking up the file into parts and then running each part individually. This not only made the algorithm more complicated to run, it made it impossible to cluster segments near the beginning of the genome file with segments near the end, regardless of their similarity. The main barriers that were preventing the algorithm from running larger files were that the computer was running out of memory due to the large number of recursive calls that added stack frames to the stack and that the size of each stack frame needed to be enlarged because of the number of required variables. To address these issues recursive methods were refactored to run iteratively, and an effort was made to reduce the size of the stack frames that were stored.

3.1.1 Refactor Recursive Methods

Since recursive methods for segmentation were first introduced to measure long-range fractal correlations in DNA sequences, significant improvements have been realized by incorporating the Markov chain model [4]. As input files increase in size, though, the nature of
recursive methods becomes problematic. Every time the genomic sequence is segmented, another stack frame is placed on the stack in memory. With larger files, this could require up to one million nested calls to the same method.

The new algorithm maintains the same logic and flow of data without the recursive structure. One call is made to the main PerformSegmentation method, which acts as a controller that calls a private FindSegmentationPoint method to find the best segmentation point. Once the optimal segmentation point is returned by the method, if the segments are sufficiently different, there are two segments that need to be evaluated for segmentation. These two segments are added to a queue that continues calling the FindSegmentationPoint method until no more segments are eligible to be segmented.

The advantages to this are two-fold: this structure maintains a very light load on the stack, requiring only two stack frames at any time for the Segmentation process, and by isolating the logic that determines the best segmentation point, the opportunity for parallel processing of these two segments can be realized.
Figure 3.1. Recursive implementation.

Figure 3.1 shows how performing many recursive method calls require more and more resources to keep all of these stack frames in memory in this nested fashion.

Figure 3.2. Iterative implementation.
Figure 3.2 shows that by isolating the code that finds the best segmentation point, the call stack maintains a light load allowing all available memory to be used for the computation and prevents the size of the call stack from being a constraint on the number of times the sequence can be segmented.

3.1.2 Reduced Stack Frame Size

The memory in a computer must be managed to not only hold the instructions of a program and values assigned to variables within the program, but also to keep track of stack frames that contain information about where the program needs to return to when a method that has been called completes its execution. Whenever a recursive method is called and before the previous stack frame is removed, a new stack frame must be added to store all of the information required to return the program to its current position after the recursive call finishes including parameters that were passed into the method, the address to return the Instruction Pointer to upon completion, and all of the local variables that are used within the function. In the initial implementation of the Markovian Jensen-Shannon divergence (MJSD) algorithm, the Cluster method used over 30 local variables requiring the stack frame to hold over 3000 bytes per call just for these local variables. By keeping the variables that must be stored in the stack frame to a minimum in the new implementation, only 16 bytes are needed to store local variables in the stack frame.

3.2 Methods Used to Increase Speed

When tests were run on the initial implementation of the MJSD algorithm the segmentation step was the most time-intensive procedure, making it the first to be considered for parallel processing. In order to preserve data integrity, multiple processors should not
access the same data items at the same time, thereby limiting the degree of parallelism to the number of genomic segments that are candidates to be segmented. Initially the Jensen-Shannon divergence is calculated for the entire input, meaning that only one processor can be used for this first step. As the genome is segmented more processors can be added, multiplying the speed at which segmentation can be accomplished. To take advantage of this opportunity for gains in runtime speed, parallel processing was implemented on the segmentation step of the MJSD algorithm.

3.2.1 Parallel Processing

In order to enable parallel processing in the segmentation step, certain parts of the logic needed to be isolated so that they could be called by multiple threads concurrently. The method that calculates the best segmentation point of a sequence and the method that splits segments were isolated into private methods FindSegmentationPoint and SplitSegment respectively. A SegmentationEngine class was created to contain all of this logic that only pertained to segmentation. A ThreadPool was chosen to manage the parallelism requiring one more method to be created – the ProcessCandidate method was created that accepts a start index and end index to be processed and acts as a wrapper for the FindSegmentationPoint and SplitSegment methods. The signature of the ProcessCandidate method is specifically structured as a work item in such a way that a ThreadPool can queue the call and assign it to a processor when one becomes available.

With the logic isolated and the queue in place to accept work items, all that remained was to queue the entire genome as the first work item to start the process. The FindSegmentationPoint method returns an index of the best segmentation point, or -1 if no
appropriate segmentation point can be found that creates a divergence greater than the confidence specified by the user. If the segment can be split, the SplitSegment method queues two new work items with the left and right side. Otherwise the segment is added to a new cluster in the global Clusters list and processing is complete on that segment of the genome.

Once a segment is processed that cannot be split and no other work items remain in the queue, the segmentation step is complete. The result is a list of clusters, each containing a single segment of the input genome that are homogeneous within and heterogeneous between [2]. Using this method of queueing work items ensures that candidates placed into the queue contain segments of the genome that are not being processed by any other processors at the same time, thus maintaining data integrity.

3.3 Methods Used to Increase Accuracy

The goal of increasing the accuracy of the algorithm was pursued on two fronts: reducing the data that is lost during entropy calculations, and offering new options to the user that were not available in the initial implementation. By giving the user additional options to alter how the algorithm runs, we allow the user to customize the steps to obtain a more accurate result. Selections were added so that the user can choose best match ordering to be used for the order that clusters should be compared, the amount that global oligomer weights should play in the algorithm, and whether the statistical significance should be calculated for each step individually, thus preparing the algorithm to serve not only as a tool for the analysis of genomic data, but also as a tool to study the effects that all of these decisions play.
3.3.1 Reducing Data Loss

Because of space constraints in the original implementation, all of the oligomer frequencies for all clusters were held in one array of integers, and the data of a certain cluster were accessed by retrieving a portion of this array using indexes for each cluster. In the entropy comparison step these values were normalized by dividing the frequency by the number of segments in a cluster. This was done so that a cluster with ten thousand segments and a segment with five hundred segments would still be seen as similar if they contain oligomers in a similar relative frequency. In order to save memory space for calculations, these normalized values were then saved over the actual oligomer frequencies in the array. As these quotients were being saved in an array of integers, any remainders were lost and only the whole part of the quotient was saved. This calculation is shown in this formula:

\[ F_{\text{Norm}}(A) = \left\lfloor \frac{F(A)}{n(A)} \right\rfloor \]

where \( F_{\text{Norm}}(A) \) is the normalized version of the oligomer frequencies in Cluster \( A \) and \( n(A) \) is the number of segments in Cluster \( A \).

Once the comparison was finished these normalized frequencies needed to be returned to the actual frequencies. In order to do this, the normalized version was multiplied by the number of genomic segments in the Cluster:

\[ F(A) = n(A) F_{\text{Norm}}(A) \]

If two clusters were combined, the oligomer frequency needed to be stored for the combined cluster. To arrive at these new frequencies, the oligomer frequencies were multiplied by the number of segments in each cluster and summed:

\[ F(A + B) = n(A) F_{\text{Norm}}(A) + n(B) F_{\text{Norm}}(B) \]
Saving these values as integer data types for each iteration led to data loss whenever two clusters were compared, which happened millions of times during the execution of the algorithm. In the refactored version of the algorithm, each Cluster object holds an array of integers with the frequency of each oligomer. During the comparison process the values are normalized by the length of the segments, not by the number of segments to allow for different sizes of segments to be found similar, even if they contain similar relative oligomer frequencies. These normalized versions are saved to a local variable of type double and the actual count remains saved in the Cluster object. In this way the accuracy of the algorithm is greatly improved because there is no data loss as a result of saving a normalized version of the oligomer frequency over the actual count each time the entropy is calculated and the actual oligomer frequency counts are preserved for the entirety of the algorithm.

3.3.2 Best Match Clustering

When clustering the segmented genomic sequence, there are a variety of clustering techniques that may be employed. Originally the algorithm utilized a simple method where clusters are compared with each other in the order they appear in the input genomic sequence. If the clusters are similar enough to be combined according to the confidence interval, they are combined and recalculated before any more comparisons are made.

One drawback of this direct method is that it may combine clusters that meet the confidence requirement, but may not be the best match available. A result of this is that by the time two distant clusters are compared, they may not be similar enough to combine even though they may have been the most similar clusters initially. Another result of this is that the
first cluster usually increases in size much more rapidly than others and the representation of true dispersion is skewed.

As an alternative to the first-order method, a user-selectable option was added to the non-contiguous clustering step that compares the first cluster with all of the others and combines that cluster with the best match first. Adding this option did increase the runtime of the algorithm because it meant that more comparisons had to be made, but it provides for a more accurate representation of which segments are most similar.

3.3.3 Global Oligomer Weights

In much the same way that text indexing systems like search engines use both the number of times that a word appears in a document and the uniqueness of that word to determine the document’s importance, it may be the case that genomic analysis can benefit from placing a value on different oligomers over others. This would likely be more valuable in larger order Markov models because a term length of 5 or 6 would allow for enough diversity in frequency to have a significant effect.

To implement this, a parameter was added to the algorithm that takes in a parameter with input values 0 through 1 to indicate the weight that the user would like to place on Global oligomer weights. An input of 0 for this parameter will result in the Global oligomer weights having no effect. An input of 1 means that Global oligomer weights are applied directly to the frequency of each oligomer in the calculation of entropy.

To calculate the Global oligomer weights for each oligomer, the algorithm employs an Inverse Collection Frequency method where the raw oligomer frequency is multiplied by \(\log\left(\frac{N}{n}\right)\) where \(N\) represents the total frequency of the oligomer in the whole genomic sequence, and \(n\)
represents the number of times that the oligomer appears in the current cluster [19]. This results in rare oligomers taking more of an effect in the similarity calculations than common ones.

![Global oligomer weight example](image)

**Figure 3.3.** Global oligomer weight example.

In the simplified example shown in Figure 3.3, Cluster 1 and Cluster 3 are the most similar clusters if all terms are weighted the same. While A, C, and G are quite common, T is very rare in this genomic sequence and it is only shared by Cluster 1 and Cluster 2. Taking Global oligomer weights into account, common oligomers would not be weighted as heavily as rare ones when determining the similarity between two clusters allowing new similarities to be discovered.

While this approach has yet to be proven effective, by allowing the user to select the magnitude for Global oligomer weights, the possible applications of the algorithm are
multiplied. This option allows researchers to compare not only result sets with or without the use of Global oligomer weights, but to also investigate these effects in an incremental manner. While our preliminary experiments did show promise for this method, this method has yet to be fully vetted and putting the decision in the hands of the user paves the way for more productive research to be performed using this new method.

3.3.4 Checking Statistical Significance

Some algorithms rely on the confidence threshold that is set by the user to determine if two segments or clusters are sufficiently similar or dissimilar. It is possible, when using the Jensen-Shannon divergence, to use weight parameters that are proportional to the length of the Markov level segments such that an analytic expression can be derived to calculate $P(D^m)$, the statistical significance of the divergence (7,9). As the size of $L$ increases, the probability distribution of $D^m$ can be approximated as:

$$P(D^m \leq X) \approx X^2_v (2L \ln 2 |X|)$$

The chi-square distribution function $X^2_v$ function here has $v = k^m(k - 1)$ degrees of freedom. The probability distribution of maximum value of $D^m$ over all possible binary partitions of a given sequence can also be approximated through a chi-square distribution function as shown by Grosse et al. (7) and Arvey et al. (9):

$$P(D_{max}^m \leq X) \approx \{X^2_v [2L(\ln 2)X|\beta]\}^{N_{eff}}$$

using values for $\beta$ and $N_{eff}$, the fitting parameters, that were calculated by Azad et al. by fitting the above analytic expression to the empirical distributions obtained via Monte Carlo. The new implementation allows the user to specify if the statistical significance should be used as an additional requirement to the confidence factor before clusters can be combined.
3.4 Additional Benefits of an Object Oriented Architecture

By structuring the code in such a way that it is easy to understand, mistakes are less likely to happen and are more easily located. This clarity lends itself to greater understanding and a much more straightforward software development experience. For example, in the original MJSD implementation, because of the procedural nature of the program, whenever clusters were compared, the program incremented and decremented indexes that pointed to specific places in the large array that held all oligomer frequencies for all clusters. In the new implementation, each cluster was created as an object that holds a list of genomic segments, a saved value for the last calculated entropy, and the oligomer frequency contained in the segments. Segments contain the start index and end index from the input sequence. Segments also provide a length property that returns the length of the genomic segment.

Since each cluster holds all of the information needed to compare it to any other segment, the order of comparison can be more easily manipulated. Instead of being forced to iterate through the clusters in the order that they naturally occur, the program can be easily altered to compare the clusters in any order, such as from smallest to largest if the user wants to evaluate combining smaller clusters first, or in order by their calculated entropy so that clusters that are more likely to combine would be compared first. This in turn makes the program more easily maintained because it is always clear which cluster is being compared, since the program references the Cluster Number instead of incrementing or decrementing the index pointer.

The program is also more easily extendable because each object can be moved around and examined by itself, without being forced into a particular order. In a similar way, the
clustering methods have been grouped into a ClusteringEngine, and the segmentation methods are now held in a SegmentationEngine. If any changes need to be made to these methods, it is clear where the changes should be made and there is no need to search through the code for copied sections that may need to be changed as well. In the future this will allow for this algorithm's functionality to be expanded beyond its current use.

The result of these architectural changes is that the program is more sustainable when issues do come up, and more extendable so that changes can be made to the user interface or specific parts of the logic with minimal effect on the unchanged parts of the program.

3.4.1 Sustainability

With any project, there will always be a need to come back to the code that was written and make changes or additions. The more clear the code is, the easier it will be to come back to it later and understand exactly what a section of the code is doing. While naming conventions have no bearing on the compiled binaries that are run by the computer, they do allow developers to avoid bugs and fix them quickly when they are found. Throughout the program, names of methods and variables have been changed to reflect the function that they serve. For example, the Cluster method in the original implementation contained these local variables:

```c
int n,i1=0,i2=0,i3=1,i5=1,i7=1,i0=1,n1=0,p,q=1,n2=0,l0=0,r=group[h-4];
double enta=0.0,entb=0.0,entab=0.0,weight1=0.0,weight2=0.0,a,b,c,d;
double jsdiv=0.0,chi_stat=0.0,dof=0.0,signif3=0.0,neff,beta,sx;
int dstrub_oligos[256]={},dstrub_oligos1[256]={},dstrub_oligos2[256]={};
```

In the new implementation, these became:

```c
/////////////////////////////////////////////////////////////////////////
// Variables for comparing CurrentCluster with NextCluster
/////////////////////////////////////////////////////////////////////////
private double Confidence;
private double weightCurrent;
private double weightNext;
```

23
Cluster CombinedCluster;

/////////////////////////////////////////////////////////////////////////
// Variables for calculating statistical significance
/////////////////////////////////////////////////////////////////////////
private double jensenShannonDivergence;
private const double a = 1.13049;
private const double b = -2.44732;
private const double c = 0.002304;
private const double d = 1.025173;
private double beta;
private double effectiveLength;
private double degreeOfFreedom;
private double chi_stat;
private double significance;

Likewise the code that iterates through the clusters to be considered for combining was refactored from the original version:

fragments[i0+2*(q-1)]=fragments[i0];
fragments[i0+2*(q-1)-1]=fragments[i0-1];
group[i0+2*(q-1)]=group[i0];

i3+);
}    h3=h3+2*(i7-1);
if(i7>1)
{
    cluster(hash,h3,group);
}

To a more understandable format in the new version:

/////////////////////////////////////////////////////////////////////////
// Iterate through the clusters contiguously, combining when appropriate
/////////////////////////////////////////////////////////////////////////
for (int i = 0; i < Globals.Clusters.Count - 1; i++)
{
    if (ClustersCanBeCombined(Globals.Clusters[i], Globals.Clusters[i + 1]))
    {
        CombineClusters(Globals.Clusters[i], Globals.Clusters[i + 1]);
        CombinedClustersOnThisStep = true;
        i--;    // If we combined this cluster we should check it again
    }
}
3.4.2 Extendibility

The well organized structure of the new implementation allows a programmer who is unfamiliar with the code and tasked with updating the program to quickly identify the area where changes need to be made without spending time searching through parts of the code that do not need to be changed. The existing classes can also be reused elsewhere without having to refer to any low-level implementation. This was demonstrated when the best match clustering procedure was added as an option to the non-contiguous clustering step. Since all of the logic for comparing and combining clusters was isolated, only a small method that specifies the order of comparison was added to enable this new functionality.

3.4.3 Ease of Making Changes to the User Interface

Initially the program was only intended to be executed from the command line or terminal. While many users are familiar with using command line programs, this type of interface still requires the user to make sure that all parameters are specified in the correct order and format, which often leads to not only confusion but errors.

Adding a user interface could be done in several different ways including a website, a desktop computer application, an online API that other programs can call, or an app that can be run on a mobile phone or smart watch. While suggesting that someone may want to run this program on a smart watch seems far-fetched now, it would have been just as implausible to suggest it be run from a mobile device only a few years ago, yet the email possibilities on mobile phones make it likely that this program may be used in this way now. By separating the logic of the program from the user interface code, the way that the program is displayed to the user can be updated with very little effort as user needs and technology change.
3.4.4 Ease of Making Changes to the Logic

Since the logic has been isolated into classes that perform only one function, the SegmentationEngine is concerned only with the process of segmentation and defers the decision of entropy calculations to the EntropyEngine. In this way, if a different divergence measure needs to be implemented, the segmentation logic does not need to change as the new calculations made in the EntropyEngine would be used for all of the program. If the algorithm needed to be adapted to use a different type of data that is not limited to the genomic symbols A T C G, only the parts of the program that deal with the actual characters would need to change, and the rest of the program could be reused without any alteration.

The original algorithm contained four steps, including an additional Clustering step that also allowed non-contiguous clusters to be combined. Since the accuracy has been increased, there was no need to have this last step. Because there is clear separation between the clustering logic and all other parts of the program, the last Clustering step could be removed without affecting any other logical part of the algorithm. By containing all of the logic within objects that do one and only one thing, steps can be added and removed without disrupting the flow of the algorithm. The number and type of steps could even be user-definable without requiring very much development.
CHAPTER 4

EXPERIMENTAL RESULTS AND DISCUSSION

4.1 Increased File Size

The original program was limited to files no larger than 200 mb, which is a problem because genomic data can frequently rise up to more than 1000 mb in a single file. The solution to this problem has been to break the genomic sequence up into parts, and then run each part separately. Unfortunately this approach can introduce inaccuracies, since the whole genome is not compared for clustering and there is no way to find similarities between segments at the beginning and the end of the sequence. Once all the recursive methods were removed and stack frame sizes were minimized, the program was able to run files up to 2000 mb and the larger files did not need to be broken into smaller ones; thus allowing the files to be processed as a whole. These improvements removed file size as a barrier to running a full genomic sequence through the algorithm and allowed all segments to be considered for clustering with all other segments within the sequence.

4.2 Parallel Processing Speedup

Two methods were used to measure the speedup of the algorithm. First files of various sizes were processed using both the old and new implementations, and the runtimes were then compared. Because of the file size limitations of the original implementation, only files that were less than 200 mb were used. Second, to test the effect that the parallel processing contributed to the reduced runtime, files of various sizes were run using the new implementation with the segmentation step running in parallel, and then again with the number of processes restricted to one so that the program ran sequentially.
The new and old implementations of the algorithm were tested on the same server using the same settings in order to compare any runtime advantages.

Figure 4.1. Runtime comparison.

Figure 4.1 shows the new, parallel version of the program had a significant improvement in runtimes versus the original version for all file sizes and Markov levels. Runtime decreased by as much as 71%, and the new implementation processed files, on average 51% faster. While it is assumed that parallelism in the Segmentation step was primarily responsible for the faster processing times, there were some time costs incurred by implementing the parallel logic, so a comparison of the new and old implementations was the result of several factors.

In order to test the effects of the parallelization on the speed of the segmentation step, the new version of the program was run on various file sizes with the program running in parallel and in sequential mode. Apart from the segmentation step, the algorithm was the same for both tests, so any speedup in the result would only be affected by the parallelism in the Segmentation step. These tests were run on a system with a quad-core processor and all
four processors were used for the parallel tests. For all test files, the output cluster files
produced by the parallel and sequential runs match exactly. However, comparisons of the
parallel versus the sequential process of the segmentation step indicate that the parallel
version resulted in a maximum speedup of nearly 4, and an average speedup of 2.9, over the
sequential version as shown in Figure 4.2.

Figure 4.2. Segmentation step comparison.

Since only the segmentation step was run in parallel the speedup of the entire program
is not as drastic as when comparing the segmentation step alone. Runtime analysis of the
entire algorithm in Figure 4.3 showed a maximum speedup of 3 and an average speedup of 2.3
for the whole program when using four processors.
4.3 Accuracy

Tests were run using an artificial *E. Coli* genome with one, five, and ten percent of the file made up of additional donor genomes. The goal of this test was to see if the algorithm was able to distinguish the *E. Coli* genomic data from the donors. Using the new implementation, one cluster was produced with most of In all of the tests the algorithm produced multiple clusters containing the *E. Coli* genomic data, but they were easily distinguishable from the clusters that were made up of donor data.

*Figure 4.3. Total runtime comparison.*
Figure 4.4 shows that the algorithm was able to group 99.99% of the E. Coli genomic data into one cluster. The two donor clusters contained 99.97% of all donor data, showing that the algorithm was able to distinguish with great accuracy between the E. Coli and donor data.
When processing a file containing ten percent donor data with the *E. Coli* genome, the new implementation produced clusters shown in Figure 4.5 where 99.8% of the *E. Coli* genomic data was correctly grouped into one cluster and 10 donor clusters produced. Most of the donor clusters were very pure, containing mostly data from only one donor. This is an improvement from the original program which either produced more clusters than desired or donor clusters of slightly less purity.

*Figure 4.5. Cluster distribution using ten percent donor data.*
CHAPTER 5

CONCLUSION

In this paper, the Markovian Jensen-Shannon divergence (MJSD) algorithm [2] was augmented so that it could address some of the common issues that plague these types of algorithms. The original MJSD algorithm takes a genomic sequence and uses the Jensen-Shannon divergence measure to segment it into homogeneous pieces that are then clustered according to the oligomer frequency that they contain. The issues of file size limitations, long runtimes, and inaccurate results were noted as problems that occurred in the original algorithm. An object oriented architecture was then used, parallel processing was introduced, and recursive methods were replaced with iterative ones. These changes were done in order to address the problems discovered in the original program.

File size limitations were addressed by refactoring recursive methods into iterative method calls. As a result of using iterative methods, the number of stack frames being stored in memory was dramatically reduced. The size of the stack frames was also reduced by minimizing the number and size of local variables that needed to be included because of the use of recursive methods. Both of these changes resulted in the ability to process larger files, increasing the maximum file size from 200 mb to over 2000 mb.

Long runtimes were also reduced by introducing parallel processing to the segmentation step, which was the most time consuming step in the original implementation. In order to allow parallel processing of the genomic sequence, the logic that finds the best segmentation point and the logic to split segments needed to be isolated from the logic that controls the data flow. Parallelism was accomplished by wrapping these methods in a work item that could be
placed on a queue and assigned to processors as they become available. This resulted in an average speedup of 2.3 over the entire program.

Accuracy issues were addressed by reducing the data that was lost in each entropy calculation and by introducing several new options that are available to the user. The addition of Global oligomer weights gives the user the ability to specify the degree a rare oligomer should be weighted higher than a common one when calculating similarity between two segments of genomic data. Best match clustering gives the user the option of specifying whether the program should produce the fastest result or one that requires additional time, but provides the best match for the cluster being analyzed before any clusters are combined. All of these changes tend to produce more accurate results as well as provide more user controlled options, which should increase the usefulness of the program for future research.

In addition to addressing the issues of file size, runtime, and accuracy, refactoring of the code into an object oriented architecture produced benefits for the next programmer who will maintain or extend the functionality. Names of methods and variables were made more descriptive and were grouped into objects that served a single function. This should enable future developers to make changes to one part of the program without worrying that other parts will cease to work. Also, this isolation of responsibilities within the program means that the SegmentationEngine can be reused in a different program without requiring any alterations. In the same way, if a different entropy calculation needs to be implemented, only the EntropyEngine requires changes before it is used in the new calculation.

The idea of using Global oligomer weights in genomic analysis is an exciting idea that should be pursued in future research until it has been shown whether it can be effective in this
domain. This algorithm can also be expanded to perform inter-contiguous clustering in which there are many different genomic sequences that require segmentation and clustering, and the clusters obtained for each sequence must be compared inter-contiguously with each other to analyze which clusters combine with one other. Numerous options were theoretically possible for future research on the MJSD algorithm and because of the work described in this research, many of these options are now attainable.
REFERENCES


