COMPARATIVE Q-GRAM ANALYSIS OF GENE PROMOTER REGIONS

BY

BLAIR FOSTER JR.

BACHELOR OF COMPUTER SCIENCE

UNIVERSITY OF NEW BRUNSWICK

2002-2003

BLAIR FOSTER JR.

SUPERVISOR:

DR. PATRICIA EVANS

FACULTY OF COMPUTER SCIENCE

UNIVERSITY OF NEW BRUNSWICK

ABSTRACT

COMPARATIVE Q-GRAM ANALYSIS OF
GENE PROMOTOR REGIONS

by Blair Foster Jr.

Supervisor:
Dr. Patricia Evans
Faculty of Computer Science

A current barrier for gene expression experiments is determining why two different genes often have expression similarities. The expression of a gene is regulated by promoters and repressors that bind to the DNA sequence directly before the gene. Given a pair genes whose expression patterns are very similar, the genes may be unrelated; one gene could promote the other; or both genes could be promoted by the same things. This last case is known as co-expression. Finding similarities in the genes' promoter regions is necessary to determining when co-expression may occur and separating groups of similarly expressed genes. If some parts of the promoter regions are common only to the genes with similar expression patterns, then they can be conjectured as promoter or repressor binding sites.

Q-gram analysis of the gene promoter regions involves taking substrings of a fixed length q and determining which genes share this substring in their respective promoter region. These q-grams will form a network, a weighted graph with gene regulatory sequences as vertices. Each vertex will be connected to all other vertices and the weight of the edge between each pair will be determined by the number of common q-grams. A pair wise analysis of the q-grams will be performed to determine the edge weights.

Once the network is formed, further investigation can be used to determine various graph partitions, and extract the sequences of highest similarity. The end result is to have a better understanding of why genes have expression similarities, and if promoter regions are the cause of these similarities.

Table of Contents:

CHAPTER 1

INTRODUCTION

1.1 Overview

The manipulation and matching of string patterns is an intensely studied area in computer science and has many applications across a broad range of areas of study. Molecular biology is an excellent example because it focuses on genetic sequences, which are represented by simple strings [17]. These sequences are continually being compared for matches and manipulation. Evolutionary changes in a particular species are often times identified in such a manner. While this may seem like a very simplistic approach, it is a necessary and useful one. It is simplistic because genetic sequences contain a large amount of structure and should be thought of as more than just sets of strings. It is necessary because each sequence demands a certain amount of genetic structure. If two sequences are similar, then it is conjectured that their structures will also be similar, and from this we can conclude that the function of each respective gene in these genetic sequences is also very similar. It is useful because we can identify most of the evolutionary changes in a particular species in this very simplistic and understandable fashion, which can then be investigated further on a much more detailed level [17]. This is all to say that there is a great amount of usefulness in comparing a large number of sequences with each another.

It is important that the distinction between a string and a sequence is identified [17]. A *sequence* is defined as a successive listing of symbols from an alphabet; in the case of genetic sequences the alphabet is the set {A, C, G, T}. A *string* can be defined as a contiguous sequence. While a

string can be a sequence, not all sequences are strings [17]. That is to say that given a group of symbols from the genetic alphabet such as ATCGAT, a sequence in itself, we can see that ACAT is a sub-sequence (**A**T**C**G**AT**), but is not a string. At the same time, TCGA is a sub-string (A**TCGA**T) and it is also a sub-sequence. The symbols TAC would not be a sequence as they are not found in correct, left to right, succession within the given group of symbols.

Given a large set of sequences it becomes necessary to examine sub-sequences and sub-strings of the sequences to determine commonalities amongst them [17]. These commonalities can be as simple as having a single common symbol, to having an entire set of sub-strings and sub-sequences in common, to being completely identical. In the case of Molecular biology, which this research will focus on, the alphabet on which these sequences are defined is very small. This creates a problem of too much repetition across very long sequences, and because of this, it is typically easier to focus on different regions of these genetic sequences, draw some preliminary conclusions, then eventually work to include the entire genetic sequence. The work contained herein focuses primarily on two things: the promoter region (the 'starting' area) of genetic sequences; and string comparisons of these promoter regions. The largest problem that presents itself here is how to efficiently deal with the large number of genes that must be compared. [17]

1.2 Genetic Sequences

Deoxyribonucleic acid (DNA) is the genetic coding from which organisms are constructed [17]. DNA consists of a long sequence of four acid base types: adenine; guanine; cytosine; and thymine. These four base types are often represented by their first letter; A, G, C and T

respectively. The four base types are linked together through a sugar-phosphate backbone creating a single sequence called a DNA strand. Each DNA strand is attached to its complementary strand through base pairing. The complement of A is T, while the complement of G is C. The DNA strands link together to form what is often referred to as the DNA double helix. Ribonucleic acid (RNA) is closely related to DNA with the exception that thymine (T) is replaced by uracil (U). [17]

Genes are also comprised of the same four acid bases and are sub-sequences of a DNA strand [17]. Within each gene itself we find even smaller sub-sequences defined to be different regions of the gene, each with its own properties. Genes consist of regulatory regions, which regulate various characteristics of the gene; the protein coding regions, which contain the necessary code segments, exons, and non-coding segments, introns, for the gene to create proteins. Regulatory regions further consist of promoter binding regions, and transcription anchor regions [17]. While the main focus for this research is in one specific area, the promoter binding regions, it is worthwhile to mention each area because they are all closely related.

I will mention why there is a need to study promoter regions more closely and briefly touch on existing methods for analyzing the promoter regions for a set of genes within a genome. I will then present a different approach to the problem of promoter sequence matching.

CHAPTER 2

BACKGROUND

2.1 Promoter Binding Regions and Gene Expression

As previously mentioned, promoter binding regions (promoters) are regulatory regions within genes, and control how and when a gene will create a protein. This routine process in which a gene creates a protein is called gene expression. It can be said then that promoters directly affect gene expression levels which is a measurement of gene expression. Gene expression consists of two primary stages: transcription and translation.

During transcription groups of enzymes and proteins bind to promoter regions [8]. These enzymes and proteins are referred to as RNA polymerase. The enzymes and proteins contained within the RNA polymerase are strictly regulated by the promoter itself. Each promoter contains a different sequence of DNA, or template, which is accessible only by the matching enzymes and proteins. The RNA polymerase begins by unwinding the DNA helix at the promoter region. The RNA polymerase then traverses down the gene, unwinding the DNA at its head and rewinding the DNA at its tail, to the transcription anchor. The transcription anchor is a small portion of DNA which exists just before the gene coding regions which signals the RNA polymerase to begin transcription of the DNA. Once the transcription anchor is located, the RNA polymerase begins to make a copy of the 'coding' DNA strand as it is unwound from its complement strand [8]. As each DNA base is synthesized to its RNA equivalent, it is added to the RNA chain being constructed. This process is called elongation. Once the RNA polymerase has reached the terminating region,

transcription of that gene has been completed and the result is a strand of RNA called messenger RNA (mRNA). While at the terminating region, the RNA polymerase collapses and the initial enzymes and proteins are released with the mRNA. The mRNA copy is then used in translation, which is the next primary phase in gene expression [8]. It is important to note that promoter regions have a huge impact on transcription because they determine which enzymes and proteins will transcribe the gene. Since transcription is the first stage in gene expression, it is clear that promoters have a large impact on the entire gene expression process.

Translation, which occurs near the end of the gene expression process, is the act of coding the RNA segments into the amino acids which make up the resulting protein. I will not describe translation in any further detail as it is difficult to relate its process to promoter regions.

2.2 Gene Co-Expression

Given two genes, gene A and gene B which exist in the same genome, there are three possibilities that can be explored with respect to their gene expression processes [5]. First, the genes can be unrelated, meaning they have relatively nothing in common. Second, gene A can be promoted by gene B, meaning that the resulting protein of gene B directly affects the expression of gene A, or gene B can be promoted by gene A. Lastly, gene A and gene B may be promoted under similar conditions. Known as co-expression, this is the central focus of this research. By examining the promoter regions of a set of genes, it is believed that co-expression can be determined based upon the similarities within the promoter regions. If two promoters have many similarities, then it can be said that the RNA polymerase that binds to these promoter regions will also be similar and thus gene expression could be triggered simultaneously and the gene expression levels will likely be very similar. Identifying those genes

that may be co-expressed is important in understanding which genes may be reliant on each other for important and necessary functionality within the organism, or which genes work together to form a common functionality. [5]

While examining promoter sequences is, once again, a very simplistic approach to identifying possible co-expression, it is the first logical and necessary step. Doing so will, if nothing else, potentially cluster the genes into a smaller subset that are more likely to be co-expressed and should be investigated further.

2.3 String Comparison Algorithms

The problem of finding substring patterns within larger sets of strings has been studied by computer scientists for many years and many algorithms to solve these pattern finding problems have been discovered [19]. While many of these algorithms were written with databases and text searching in mind, these algorithms are certainly applicable to genetic sequences.

Similar to any well-defined algorithm, string pattern matching algorithms can be adapted to their problem environment, often yielding even greater efficiency by exploiting restrictions to the problem [19]. One obvious restriction which can, and will be, exploited in this research is the fact that genetic sequences are defined over a very small four letter alphabet. This fact quickly singles out algorithms which are most effective over small alphabets, a few of which I will mention, along with the reasons why these algorithms are not the best approach to the problem at hand.

2.4 Using Q-grams

The idea of using q-grams, substrings of a fixed length q, was first introduced [6] as a filtration technique for text and database searching. The idea was that a q-gram could be any particular word used to filter out areas of text. However, when applied differently, q-grams could also identify text sequences which had words in common [11]. Being able to identify common pieces in sequences using q-grams gives the ability to measure the similarity of the sequences, because it is expected that similar sequences will share a large number of q-grams. [16]

Knowing that q-grams can be used to measure similarity between two sequences is one reason for choosing this method. The other reason is that q-grams can effectively encapsulate entire promoter regions given that q is large enough. This is very valuable as it means promoters compare exactly to other promoters with a minimal amount of non-promoter data on either end of the q-gram.

2.4.1 Q-grams and String Algorithms

When dealing with q-grams, the pattern in the string searching algorithm limited to a fixed length (q). Determining a useful value for q depends entirely on data being used. An appropriate q value for genetic sequences is one which is large enough to encapsulate some genetic function from the sequences. This, of course, is different for every genome. A study of genetic sequences and their related functions would be encouraged before placing a value on q. However, large values for q would certainly restrict the amount of similarity across a set of genes. Focusing on smaller values would, while allowing for more error, encourage similarity.

The methods I used to determine an appropriate value for q are a bit simplistic because studying gene functions is well outside the bounds of this research. Instead, I have decided to run many tests using various values for q and present the results for each value. I will be using values of q which are relatively small. The idea, as mentioned earlier, is to try and encapsulate the promoter, or most of it, within the q-grams. Because of this, choosing a value for q that is too large would mean not only getting the promoter, but the other non-promoter data around it. The danger here is that the similarities may become too great as the q-grams are small, and random similarities may be found instead of promoter regions.

2.5 Measuring the Value of an Algorithm

In order to determine the level of effectiveness of an algorithm to a problem, it is often necessary to understand the efficiency of the algorithm [18]. Efficiency can be viewed both in terms of the time it takes to run the algorithm and the amount of space required. This is measured using asymptotic complexity. Asymptotic complexity allows algorithms to be measured for their efficiency based upon the size of their input. A high asymptotic complexity translates to a very poor algorithm for large input sizes, while a small asymptotic complexity translates to a very efficient algorithm for large input sizes [18]. This is extremely important in this research, as the number and size of the sequences which can be compared is very large. It is clear then that the desired algorithm will require a low asymptotic complexity, especially as it relates to the running time of the algorithm.

Asymptotic complexity can be denoted in many ways, but typically it is denoted by one of three Greek alphabet symbols: omicron (**O**), referred to as 'Big-Oh' [13]; omega (**Ω**), referred to as 'Big-Omega' [14]; and theta (**Θ**), referred to as 'Big-Theta' [15]. 'Big-Oh' is used to

denote an asymptotic upper bound for an algorithm, which is to say that its asymptotic complexity can not exceed a constant multiple of this value. I will be using 'Big-Oh' notation, since it is the most understood and commonly used notation.

CHAPTER 3

ALGORITHM DESIGN

3.1 Dealing with the Problem Efficiently

The focus of this research is to determine if there is a relationship between promoter regions of a set of genes. This resulting relationship might better explain the high levels of similarity across the gene expression data. This requires a large number of comparisons to be performed across all the input sequences looking for matching q-grams. For each q-gram that is found in one sequence, a search of each remaining sequence is needed to determine if they also contain this q-gram. Any pair of sequences that contain this q-gram must be noted as having some level of similarity; if one does not contain the q-gram, the pair is noted as having some level of difference. The number of occurrences of a particular q-gram in a sequence is how the measurement of similarity or difference is done. Given this information, it is easy to see that this requires a large amount of computation. Consider that there are at most k sequences, with each sequence containing at most n bases. For each sequence at most n-(q-1) q-grams can be extracted. Each q-gram from each sequence must then be searched for in all remaining sequences. Combining these steps gives an algorithm which resembles the following:

For each sequence j do the following:

1) Extract the next substring of length q (q-gram)
2) For each sequence m, where m is not j do the following:
    1) Search for the q-gram from j in m

As can be seen, this yields a time complexity of $O(k(n-(q-1))(k-1)$(complexity of search algorithm)). So clearly a fast searching algorithm must be found, however, this will still provide a very inefficient algorithm. I will discuss a pair of fast searching algorithms which could be used for this approach. I will also discuss a completely different solution which could be used, and finally, explain a much more efficient method that I have devised for the problem.

3.2 String Searching Algorithms

3.2.1 Knuth-Morris-Pratt (KPM)

The KPM algorithm is an extremely well-known and efficient algorithm for string searching [19]. Given a string pattern of length m, and sequence of length n, the KPM algorithm will find all occurrences of the pattern within the sequence in $O(n + m)$ time and $O(m)$ space [4]. However, given the problem as stated above (with $m = q$), this leaves us with a total time complexity of $O(k(n-(q-1))(k-1)(n+q))$ which itself has an upper bound complexity of $O(k^2n^2)$. [19]

3.2.2 Boyer-More (BM)

The BM algorithm is yet another well-known and efficient algorithm [19]. It is very popular and is used by many applications that have "search and substitute" functionality. Given a string pattern of length m, a sequence of length n, and an alphabet size of σ, BM will find all occurrences of the pattern within the sequence in $O(nm)$ time and $O(m + \sigma)$ space. However, given the problem as stated above (with $m = q$), this leaves us with a total time complexity of $O(k(n-(q-1))(k-1)(nq))$ which itself has an upper bound complexity of $O(k^2n^2q)$. [19]

3.2.3 Variations on BM and KPM

While there are many variations of both the BM and KPM algorithms [2], I will not discuss those as they do not provide any significant amount of improvement in terms of asymptotic complexity.

3.3 Q-gram Alignment Based on Suffix Arrays (QUASAR)

The QUASAR algorithm [1] takes a much different approach to searching for q-grams in a given set of sequences. Originally designed for database searching, the algorithm works on the principle that if two sequences are relatively the same (that is to say the amount of editing required to make one sequence identical to the other is reasonably small) then it follows that the two sequences will share a large number of q-grams. This then allows a large amount of pruning of the sequences to be searched. Using suffix arrays, sequences are initially grouped together based upon a scoring scheme. Following this, q-grams can be searched for within each grouping separately. While this sounds a bit redundant (grouping similar sequences together only to further show that they are similar), it is a valid approach to exact pattern matching because the initial grouping deals strictly with approximation and does not identify the exact matches.

The running time of this algorithm is measured in two parts. It initially takes $O(k\log_2 k)$ time to perform the suffix preprocessing on k sequences. The searching then requires $O(k)$ time for each q-gram. Given that there can be a maximum of n-(q-1) q-grams from each sequence of maximum length n, the total searching time is $O(k^2 n)$. Of course this time complexity is the worst case and considers that all sequences are similar to each other. [1]

Clearly this approach is much more efficient than the previously mentioned methods. However, it is a very complex method and requires preprocessing of the sequences beforehand. More importantly, the QUASAR technique cannot handle common sequences where the q-grams are unordered. This is actually a problem with most existing q-gram methods as well; these methods are all concerned with the order in which the q-grams appear in the sequence. This makes most q-gram methods useless for this problem because promoters are unordered and the desired technique needs to account for this. I will proceed to demonstrate a much more simplistic q-gram method which applies specifically to the problem at hand and makes use of a few obvious restrictions to allow for an even faster solution.

3.4 Using Q-grams Differently

As mentioned earlier, looking for q-grams using string searching algorithms can be very time consuming. One of the reasons for this is that character-to-character comparisons need to be made for each searched q-gram. One way I propose to speed this up is described in the following paragraph.

If each of the characters in our four letter alphabet correspond to two bits in a bit string, that is {A = 00, G = 01, C = 10, T = 11}, each q-gram will be labeled with an integer between 0 and 4^q. Next, a table of the number of possible q-grams (4^q rows) by number of sequences (k columns) can be constructed. This table will be used to store the number of occurrences of a particular q-gram within each sequence. By looking at each sequence in turn, it is then possible to process each sequence's q-grams by extracting them, converting the q-gram to its integer value, and updating the table. An additional column will be used to indicate if a particular q-

gram was present in any of the sequences. This last column will be used later to speed up the comparison of the sequences. Building this table (M) is done as follows:

For each sequence (k) do the following:

1) Extract the first q-gram (first q characters)

2) Convert the q-gram to its integer value (qval)

3) Update the q-gram column, M[qval, last column] += 1

4) Update the sequence column, M[qval, k] += 1

5) Shift the q-gram characters and grab the next character in the sequence to form the next q-gram

6) Repeat from step 2 until all characters in the sequence have been used.

This process can be accomplished in linear time (O(kn) for a maximum sequence length of n).

Once the table of q-gram occurrences has been built, the next step is to perform a pair wise comparison of the sequences. This comparison will be determined by looking at the number of occurrences of a particular q-gram within the pair of sequences. To do this, a positive score value (pos), and a negative score value (neg) will be needed. A tolerance value (tol) must also be used to specify the allowable difference between the q-gram value pairs. The scores from this comparison will be stored in a separate table (T). The scoring shall be described in the next paragraph.

For each q-gram (qval) that was found (M[qval, last column] > 0), do the following:

For each pair sequences i and j (j > i):

1) If M[qval, i] = 0 or M[qval, j] = 0, then T[i, j] = T[i, j] + neg, otherwise;

2) If |M[qval, i] – M[qval, j]| < tol , then T[i, j] = T[i, j] + pos, otherwise;

3) T[i, j] = T[i, j] + ((MIN{M[qval,i], M[qval,j]}/MAX{[qval,i], M[qval,j]}) * pos)

Analyzing this algorithm, the complexity is found to be $O((4^q\text{-}c)k(k\text{-}1))$, where c is the number of q-grams that were not found within the sequences. This complexity itself has an asymptotic upper bound of $O(4^q k^2)$. It must then be determined if this algorithm is actually better than $O(k^2n^2)$. While it may not be immediately obvious, I will proceed to show that it is. When considering genetic sequences, the number of characters can become extremely large (I consider sequences of length 1000). Even in this case, where I only look at regulatory regions of the genes, the sequences are still very large. Also, when you consider that promoters are expected to be relatively short (I examine $q \leq 8$), 4^q becomes smaller than n^2. However, even if q begins to become large, the number of q-grams actually found in these sequences will decrease, and as such the number of q-grams ignored (c) will begin to increase. As such, even though the possibility of encountering more q-grams increases, the number of q-grams actually used for comparison decreases.

CHAPTER 4

IMPLEMENTATION AND TESTING

4.1 Implementation

Implementation of the above algorithm was written as an application in the C programming language [7] (Appendix 1). The application allows for user specification of q-gram length (q), number of input sequences (k), the desired number of output sequences, a tolerance value (tol), as well as positive (pos) and a negative (neg) scoring values. All of these values must be specified upon runtime. A flat file of the gene sequences is required, and it is expected that this file is formatted with one sequence per line. This file is also specified by the user.

The reason for making the application flexible is to allow various tests to be performed and to be able to compare the effects each value has on the results. It should be noted that keeping the positive and negative score values approximately the same absolute size is not necessary, however doing so keeps the weighting of the comparisons less inflated.

The output from this application includes the number of q-grams used for sequence comparisons (4^q - c), as well as two files that will contain a user specified number of the highest and lowest scoring pairs of sequences.

4.2 Testing

To test the algorithm, I decided to use various lengths of q-grams to investigate what effect the size of the q-grams would have on the output. Values of 5, 6, 7, and 8 where chosen as lengths for the q-grams. The reason for choosing these values was based upon discussion with Dr. Evans. If I were to use values that are below 5, it is expected that a lot of random similarities will occur regardless of promoter similarity. A maximum q-gram length of 8 was chosen not only from discussions with Dr. Evans, but also due to the limited availability of hardware capable of supporting the memory requirements.

I also used two different cases for positive and negative scores. In the first case, I used a positive score value of 1, and a negative score value of 0. As such if, for a pair of sequences, one sequence has a particular q-gram where the other sequence does not, I will not penalize this. In the second case, the positive score value remains 1, however the negative score value was set to -1. This was done so that in the above case of a poor sequence comparison, the score will be penalized. This keeps the sequence comparison score equally in favor of both positive and negative scoring, where as in the first case, I allow for positive scoring only. In both test cases the tolerance value (tol) was 0.

The input data that I tested consisted of the entire S. cerevisiae (yeast) genome. The availability of both sequence data [3] and expression data [9], as well as the small size of this genome (approximately 6200 genes) were the compelling reasons for choosing this input data. For each gene from the genome, sequences of length 1000, upstream of the transcription anchor, were extracted. These were then formatted into a large flat file.

Below are the results from the first case of testing as mentioned earlier. The scatter plot graphs (Figures 1 – 4) show the 100 highest pairs of q-gram similarity, as well as the 100 lowest pairs of q-gram similarity for the corresponding value of q. The pairs are plotted according to their q-similarity score vs. their difference in expression values.

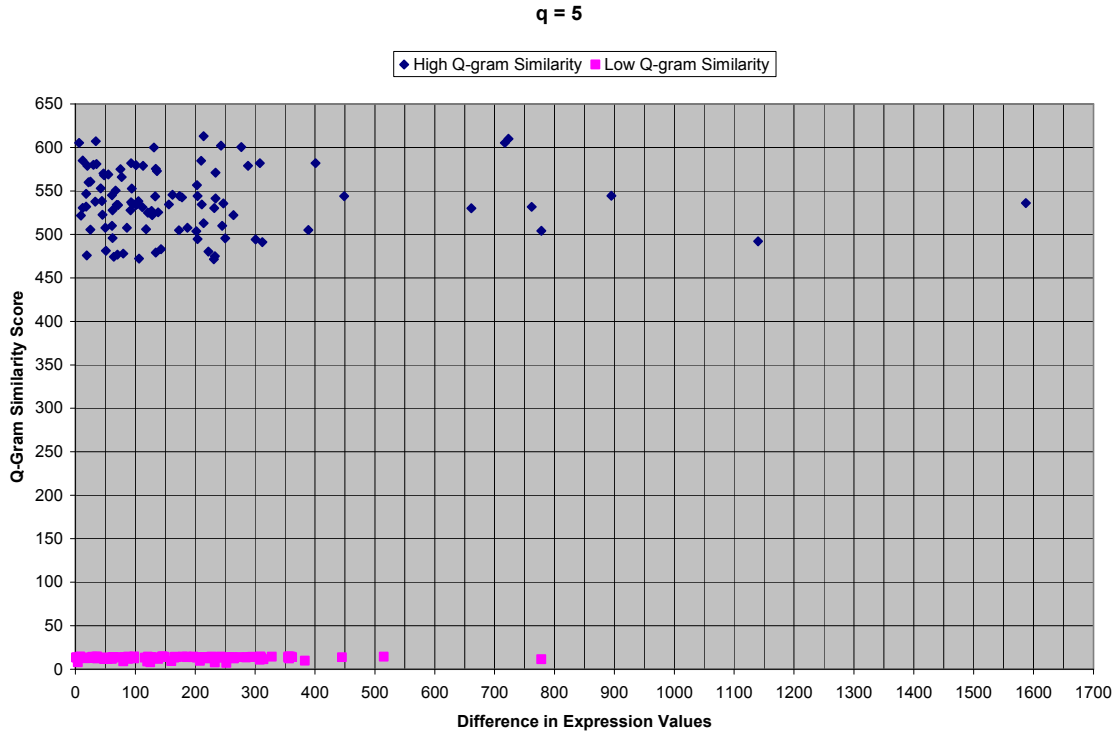

Figure 1 : q = 5, pos = 1, neg = 0

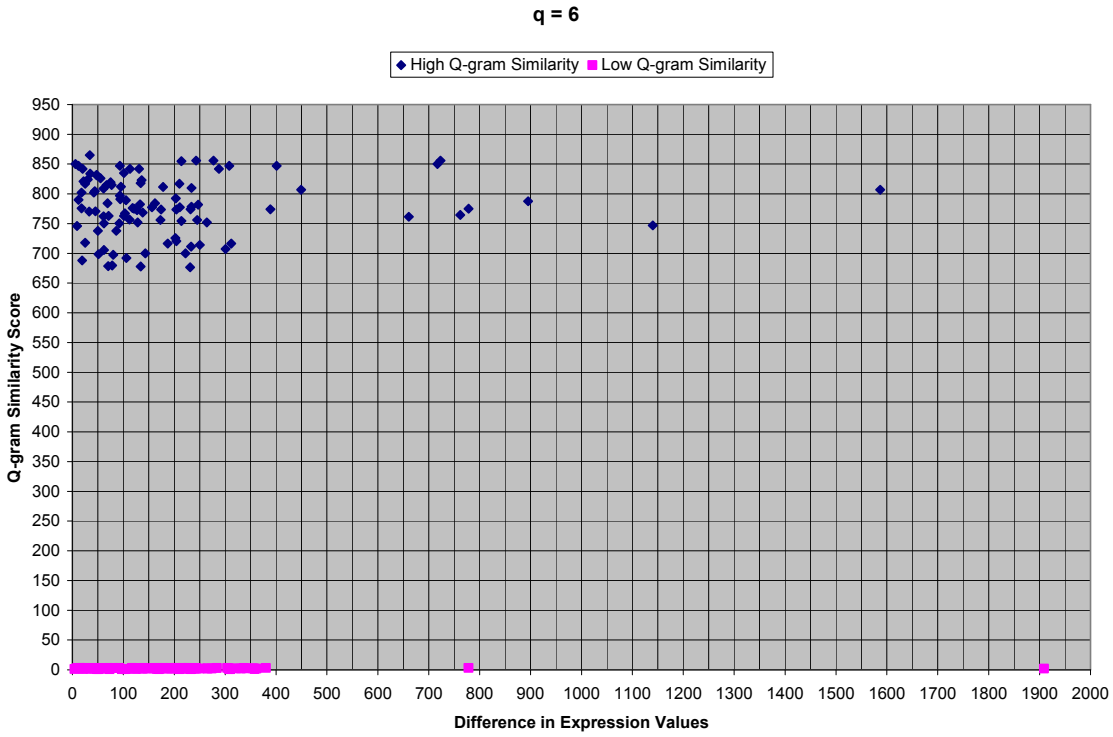

Figure 2 : q = 6, pos = 1, neg = 0

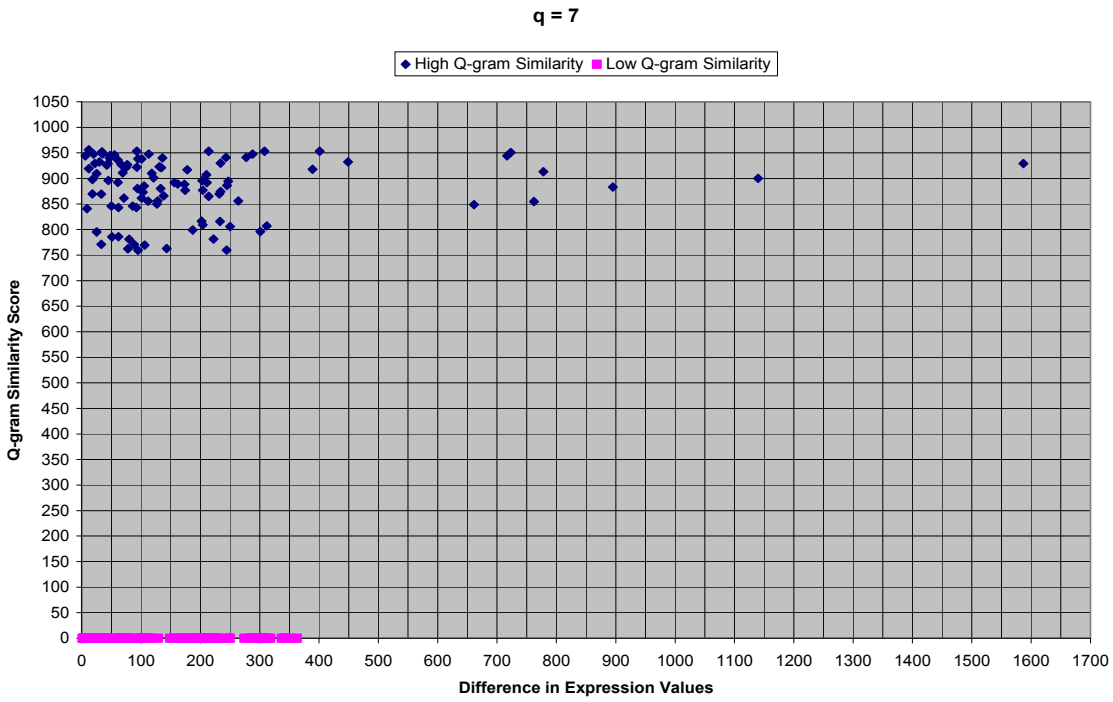

Figure 3 : q = 7, pos = 1, neg = 0

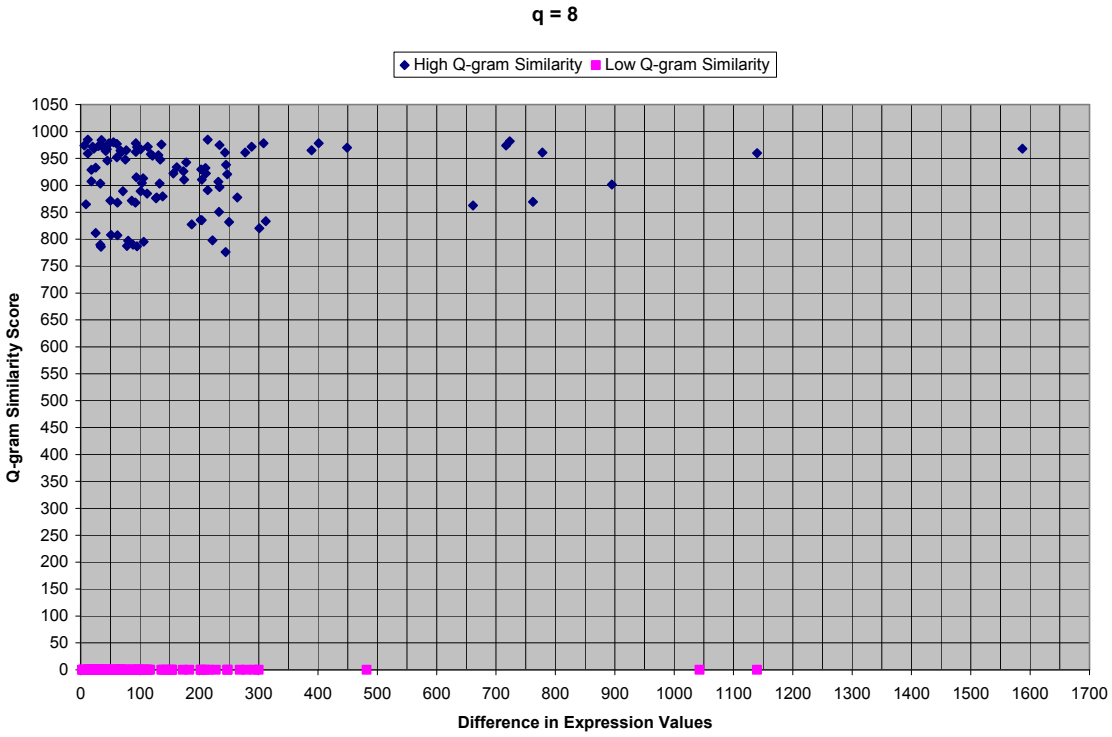

Figure 4 : q = 8, pos = 1, neg = 0

The second case of testing produced the following results (Figures 5 – 8). Note that in this case of testing, I am using a value for neg that will penalize the q-gram comparison when necessary.

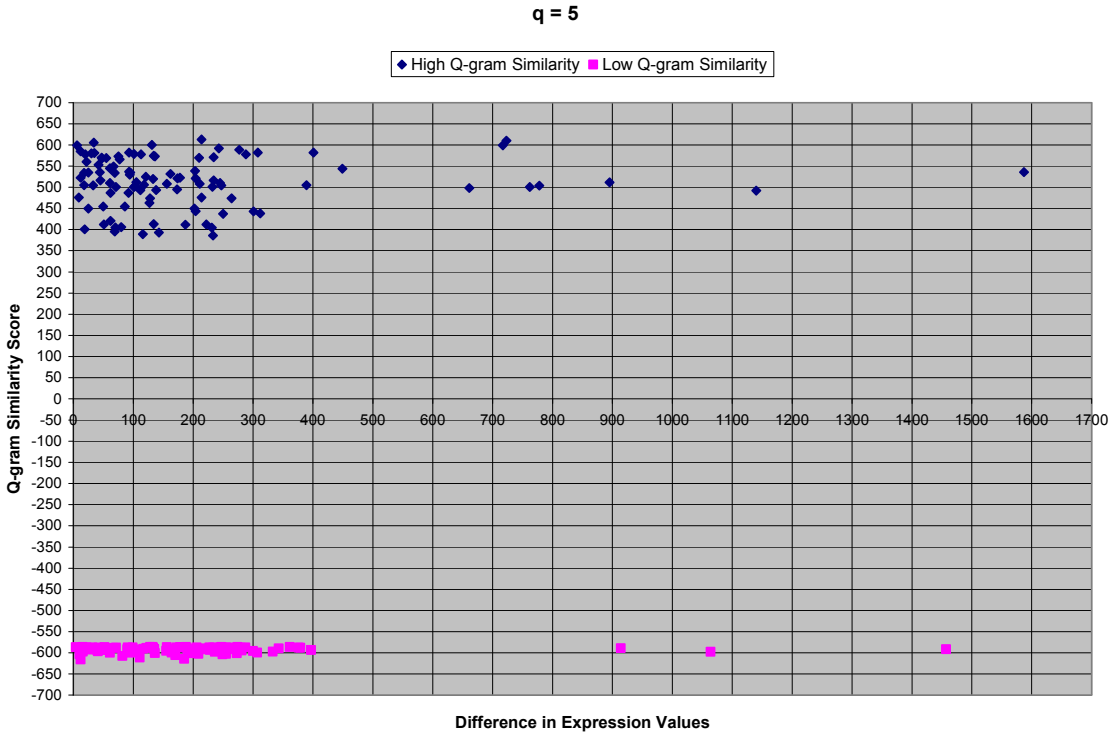

Figure 5 : q = 5, pos = 1, neg = -1

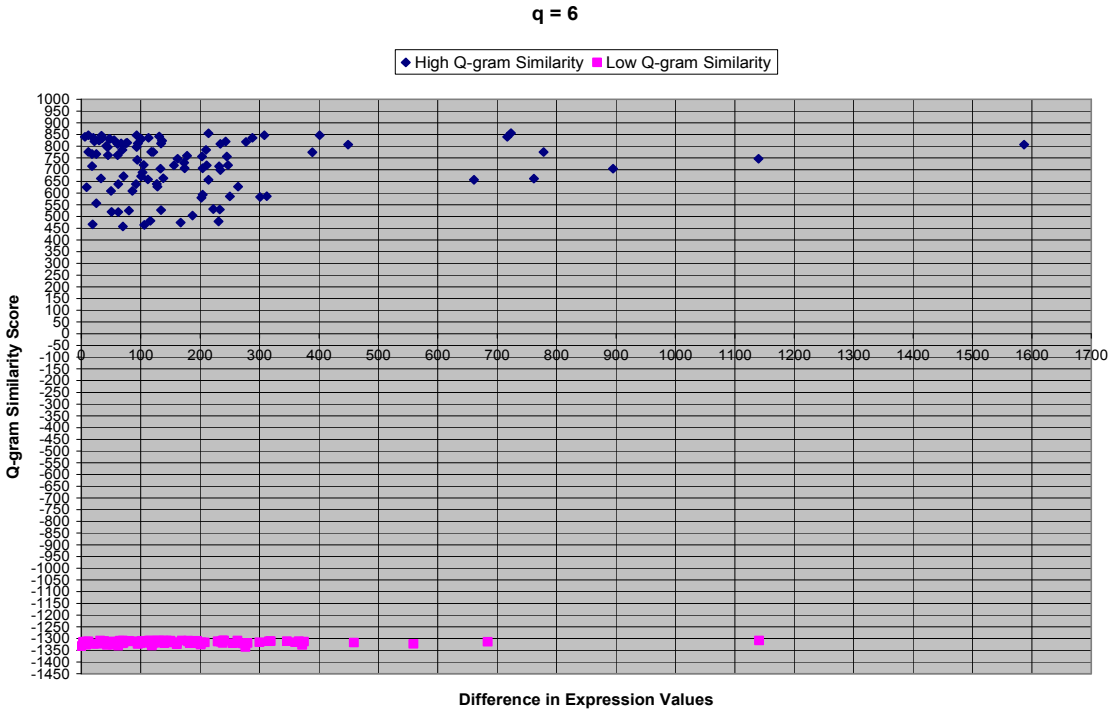

Figure 6 : q = 6, pos = 1, neg = -1

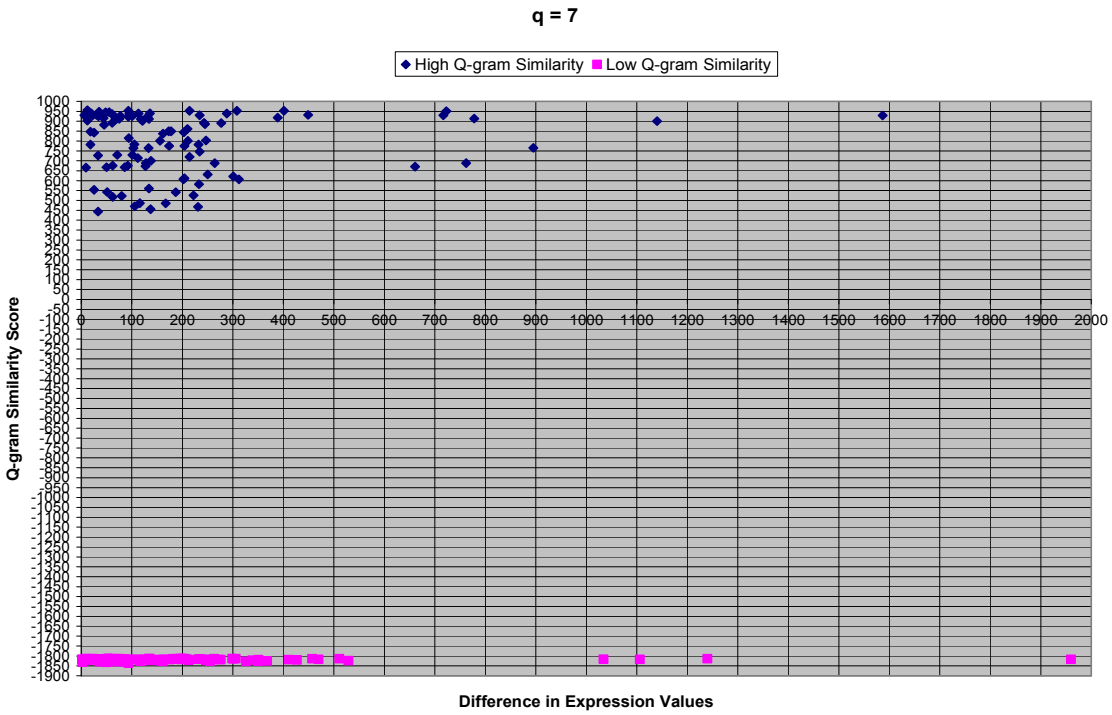

Figure 7 : q = 7, pos = 1, neg = -1

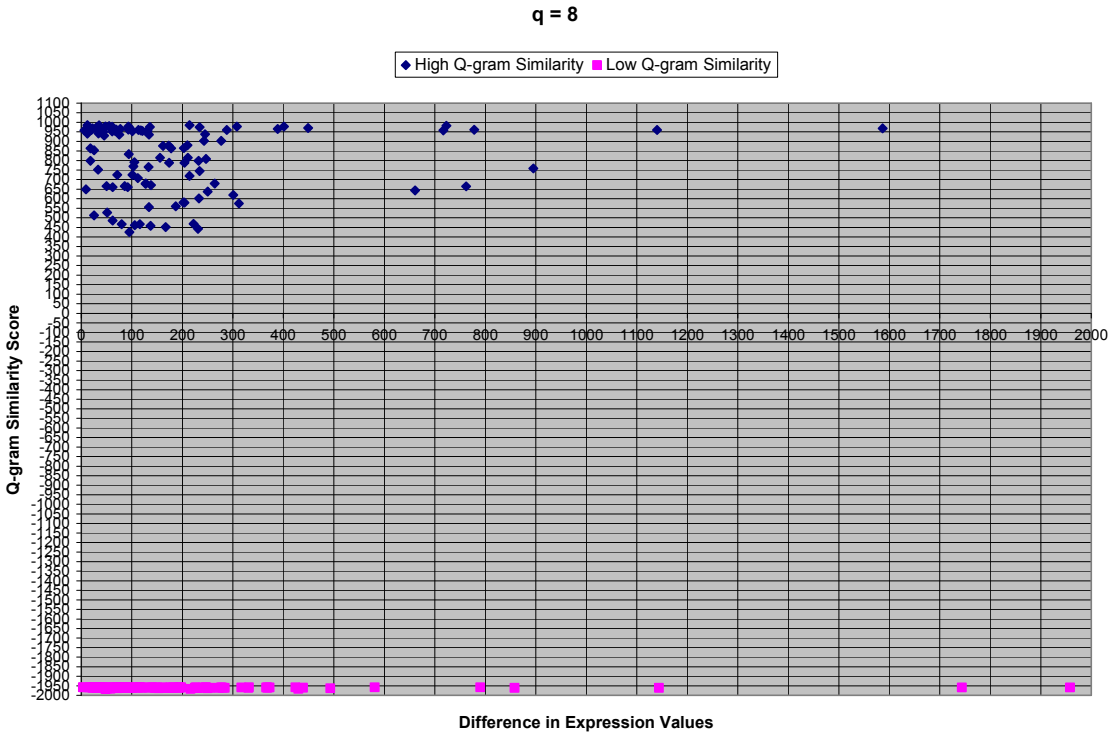

Figure 8 : q = 8, pos = 1, neg = -1

4.3 Results Analysis

When looking at the results, it is obvious that four different scenarios appear. Initially, plots can have low q-gram similarity and low difference in expression values. If this result is seen, then it is possible to conjecture that since the q-gram similarity is low, but the expression similarity is high (their difference is low), then some other factor outside of the promoter regions is causing the co-expression of these genes.

Secondly, plots can have low q-gram similarity and high difference in expression values. In this case it could be conjectured that since the q-gram similarities are low, the promoters are most

likely very different. As a result a high difference in expression values is expected, and these genes should not be co-expressed.

Thirdly, plots can have high q-gram similarity and low difference in expression values. This scenario is clearly the desired case as it could be conjectured that the high similarity in the promoter regions could have a direct effect on the co-expression of these genes. This is important as it corroborates the arguments of this research and I will focus primarily on this scenario.

Finally, plots can have high q-gram similarity and high difference in expression values. This last scenario is the least desired one. In this case there is another reason, other than promoters, which lead to very different expression levels for the genes, and it can be conjectured that promoters have very little effect on expression levels. This goes against the argument of this research; promoter similarity could help explain expression similarity.

The remaining question concerns an acceptable level of expression difference to consider the genes co-expressed. When studying the expression data used, it was found that the average expression value for a gene was 297, where the smallest value was 171, and the largest value was 6999. Clearly then, the expression values between a pair of genes can potentially be radically different. The only effective way to establish bounds on co-expression is to perform a gene clustering technique to establish sets of co-expressed genes [5]. Clustering this expression data is outside the scope of this research, however, reviewing data compiled by some of my own previous research [4] reveals that dense clustering of the S. cerevisiae genome can and does occur with genes having expression values that differ by as much as 100-150. This range makes sense because it is expected that genes whose expression values are say ~350

to be paired with genes whose expression values are also ~300-400, giving a total window of ~100.

Within the first case of testing, it can be seen how the length of the q-gram affects the results. When q is 5 (Figure 1) there is a lot less density within the plots than can be seen when q is 8 (Figure 4). Also it is expected that when q is smaller, there will potentially be a larger difference in expression values since there is a greater potential for more random q-gram similarities. This is evident when q is 5, as many more gene pairs of high q-gram similarity fall outside the acceptable level of expression similarity. Increasing to q to values of 7 and 8, there is a much denser grouping of the plots falling within the 0 – 150 range of expression value difference. More importantly, the distance of the plots that fall outside this range decreases as well. This means that as q is increased, and genes with higher promoter similarity are found, genes with higher expression similarity are also found.

Also worth noting is that the number of plots which fall extremely far outside the 0 – 150 range stays very consistent, and very few. This is desired because it minimizes the cases where the promoters are very similar yet the expression value of one of the genes is extremely high and the other is extremely low.

Switching attention to the second case of testing, where poor comparisons are penalized, slightly different results appear. While the increase of q still admits the same increase in the grouping of plots, the effect is much more noticeable. The density of the plot groupings becomes very intense when q is 7 or 8. This is most certainly due to the addition of penalizing sequences which admit very different q-grams. In effect, these poor scoring sequences get filtered out as the value of q is increased. There is also an increase in the number of plots

which fall into the acceptable (0 – 150) range. Once again this is due to the filtering effect of including the ability to penalize sequences. Yet another consistency throughout the tests is the number of plots which fall outside the acceptable range. This number stays very consist, and once again, very low across all values of q. The overall consensus from these tests is that many genes in this genome with similar promoter regions also exhibit very similar expression.

CHAPTER 5

CONCLUSION AND RECOMMENDATIONS

5.1 Conclusion

Gene promoter regions are only a small piece of the puzzle known as gene expression. But by understanding and studying this piece, it brings us closer to understanding how and why gene are expressed and co-expressed. This in turn helps to further understand which genes are needed for particular functions of each organism and the affects of evolution on those genes and functions.

Using q-gram analysis of the promoters is a valuable approach to determining promoter similarities across a set of genes, and improves identification of co-expressed genes. The results confirm this, as there is some positive relationship between the promoter similarity and co-expression of a pair of genes. It must be noted, however, that not all co-expression can be explained simply by examining promoters.

5.2 Recommendations

There are several areas of improvement which could be further investigated. One of which includes using known data about the promoters of a genome so that random similarities among the sequences can be minimized. The idea here is that the algorithm will only consider specific sets of known promoters and ignore all the other "garbage" which is not a known promoter. If very little information is known about the promoters of the genome, then it is

also possible to use a gene expression clustering algorithm to limit the number of genes that need to be examined. For instance, if a known group of highly clustered genes which exhibit very similar expression values, this could limit our input set to only those genes, thus examining only their promoters. A slight downside to this might be that it limits the ability to identify genes with high expression similarities across different clusters since the input data forces all co-expressed genes to be present in the same cluster.

Aside from the above mentioned filtration techniques, q-gram differences, and not similarities, could also be examined. The idea behind this is to identify pairs of genes which have such different q-grams that only the remaining pairs need to be investigated for co-expression. This is essentially the complement of the approach taken by this research.

APPENDIX 1 – IMPLEMENTATION CODE

```
//Q-Gram Application to score gene sequences.
//Blair Foster
//Last Modified - April 02, 2003
#include <stdio.h>
#include <math.h>

//Some useful global variables
int pattern_length;
int num_sequences;
char *pattern;

//for matrix
int rows;
int cols;
char **matrix;

//for files
char *fileName;
FILE *inputFile, *outputFile;

//seq. scoring
float **score_matrix;
int *total_qgrams; //not currently used
int score,scorediff, scorepos, scoreneg;

//finding similarity
int current_seq;
int smallest,highest;
int sim_num;

struct pairs{
int seq1;
int seq2;
float val;
};
struct pairs *highlist;
struct pairs *lowlist;


void initLists()
{
        int i;
        smallest = 0;
        highest = 0;
        highlist = (struct pairs *) malloc(sizeof(struct pairs)*sim_num);
        lowlist = (struct pairs *) malloc(sizeof(struct pairs)*sim_num);
        for(i = 0; i < sim_num; i++)
        {
        highlist[i].val = 0;
        lowlist[i].val = 1000000; //some reasonably 'high' value!
        }
}

void tryListInsert(int i, int j, int which)
{
        int k;
        if(which == 0)
        {
                if(score_matrix[i][j] > highlist[smallest].val)
                {
                        highlist[smallest].val = score_matrix[i][j];
                        highlist[smallest].seq1 = i;
                        highlist[smallest].seq2 = j;
                        smallest = 0;
                        for(k = 0; k < sim_num; k++)
                        {
                                if(highlist[k].val < highlist[smallest].val)
                                smallest = k;
                        }
                }
        }
        else
        {
                if(score_matrix[i][j] < lowlist[highest].val)
```

```
                {
                        lowlist[highest].val = score_matrix[i][j];
                        lowlist[highest].seq1 = i;
                        lowlist[highest].seq2 = j;
                        highest = 0;
                        for(k = 0; k < sim_num; k++)
                        {
                                if(lowlist[k].val > lowlist[highest].val)
                                        highest = k;
                        }
                }
        }

}

void printSeqCompare(int which)
{
        int i,status;
        if(which == 0)
        {
                outputFile = fopen("seq_compare_data_high.txt","w");
                for(i = 0; i < sim_num; i++)
                {
                        status = fprintf(outputFile,"Sequences %d and %d\n -
                        %f",highlist[i].seq1,highlist[i].seq2,highlist[i].val);
                }
        }
        else
        {
                outputFile = fopen("seq_compare_data_low.txt","w");
                for(i = 0; i < sim_num; i++)
                {
                        status = fprintf(outputFile,"Sequences %d and %d\n -
                        %f",lowlist[i].seq1,lowlist[i].seq2,lowlist[i].val);
                }
        }
}


/*Find the desired number of
* most similar sequences
*/
void seqCompare()
{
        int i,j,k;
        initLists();
        for(i = 0; i < num_sequences; i++)
                for(j = i+1; j < num_sequences; j++)
                {
                        tryListInsert(i,j,0);
                        tryListInsert(i,j,1);
                }
                printSeqCompare(0);
                printSeqCompare(1);
}



//Initialize the matrix to all 0's
void initMatrix(char *m[], int nrows, int ncols)
{
        int i,j;
        printf("Initializing Matrix....");
        fflush(stdout);
        for(i=0; i<nrows; i++)
        {
                for(j=0; j<ncols; j++)
                m[i][j] = 0;
        }
        printf("Done!\n");
}



//Initialize the matrix to all 0's
void initFMatrix(float *m[], int nrows, int ncols)
{
        int i,j;
        printf("Initializing Matrix....");
        fflush(stdout);
        for(i=0; i<nrows; i++)
```

```
        {
                for(j=0; j<ncols; j++)
                        m[i][j] = 0;
        }
        printf("Done!\n");
}


//print out score data
void printScore()
{
        int i, j, status;
        outputFile = fopen("score_data.txt","w");
        for(i = 0; i < cols - 1; i++)
        {
                status = fprintf(outputFile,"%d\t",i);
                //printf("row %d\n",i);
                for( j = 0; j < cols - 1; j++)
                        status = fprintf(outputFile,"%.2f\t\t",score_matrix[i][j]);
                status = fprintf(outputFile,"\n");
        }
}

/* For later implementation
* We can use this to consider the number of q-grams in a sequence
* when scoring
*/
void getTotals()
{
        int i,j,temp;
        temp = 0;
        total_qgrams = (int *)malloc(sizeof(int)*num_sequences);
        for(i = 0 ; i < num_sequences; i++)
        {
                total_qgrams[i] = 0;
                for(j = 0; j < rows; j++)
                {
                        total_qgrams[i]+=matrix[j][i];
                }
        }
}

/*Score the sequences in a pairwise fashion
* but only consider q-grams that have been found
* also we don't look at a q-gram more than once.
* Saves time!
*/
void scoreMatrix()
{
        int i,j,k;
        unsigned int count;
        float percent,add;
        count = 0;
        score_matrix = (float **) malloc (sizeof(float **) * (cols-1));
        getTotals();
        for(i=0; i<num_sequences; i++)
        {
                score_matrix[i] = (float *) malloc(sizeof(float)*num_sequences);
        }
        initFMatrix(score_matrix,num_sequences,num_sequences);

        for(i=0;i < rows; i++)
        {
                if(matrix[i][cols-1] != 0)
                {
                        count += 1;
                        if(i % 100 == 1)
                        {
                        printf(".");
                        fflush(stdout);
                        }
                        for(j=0; j < num_sequences; j++)
                        {
                                for(k=j+1; k < num_sequences; k++)
                                {
                                        if(matrix[i][j] == 0 && matrix[i][k] == 0)
                                                {/*do nothing!*/}
                                        else if(matrix[i][j] == 0 || matrix[i][k] == 0)
                                        {
                                                score_matrix[j][k]+=scoreneg;
                                        }
```

```
                                        else
                                        {
                                                if(abs(matrix[i][j] - matrix[i][k]) <= scorediff)
                                                        score_matrix[j][k]+=scorepos;
                                        else
                                        {
                                        /*We want to use a percentage of scorepos since both
                                        sequences
                                        * have the q-gram, but their difference is
                                        signifigant
                                        */
                                                if(matrix[i][j] < matrix[i][k])
                                                        percent = ((float)(matrix[i][j])) /
                                                        ((float)(matrix[i][k]));
                                                else
                                                        percent = ((float)(matrix[i][k])) /
                                                        ((float)(matrix[i][j]));
                                                score_matrix[j][k]+=(scorepos * percent) ;
                                        }
                                }
                        score_matrix[k][j] = score_matrix[j][k];
                        }
                }
        }
        matrix[i][cols-1] == 0;
        free(matrix[i]);
        }
        printf("\n%d q-grams Examined!\n",count);
}

//print out matrix data
void printMatrix()
{
        int i, j, status;
        outputFile = fopen("matrix_data.txt","w");
        for(i = 0; i < rows; i++)
        {
                status = fprintf(outputFile,"%d\t",i);
                //printf("row %d\n",i);
                for( j = 0; j < cols; j++)
                        status = fprintf(outputFile,"%d\t",matrix[i][j]);
                status = fprintf(outputFile,"\n");
        }
}

//simple value assigned to the characters
int charVal(char c)
{
        if (c == 'a' || c == 'A')
                return 0;
        if (c == 'g' || c == 'G')
                return 1;
        if (c == 'c' || c == 'C')
                return 2;
        if (c == 't' || c == 'T')
                return 3;
        return -1;
}

//calculate the matrix position of the q-gram
int calPos(char string[])
{
        int i,ret,val;
        ret = 0;
        for(i=0;i<pattern_length;i++)
        {
                val = charVal(string[i]);
                if (val == -1)
                {
                        printf("Encountered an invalid character %c.... exiting!\n",string[i]);
                        exit(1);
                }
                ret = ret + pow(4,pattern_length - i - 1) * val;
        }
        return ret;
}

/*We can free the q-gram matrix after scoring.
*/
void freeMatrix()
{
```

```
//    free(matrix);
//    free(pattern);
       printf("All matrix memory has been de-allocated!\n");
       fflush(stdout);
}


//Shift input characters to prepare for next input
void shift_chars(char *string,int length)
{
       int i;
       for(i = 0; i < length - 1; i++)
       {
              string[i] = string[i + 1];
       }
}


//update the table entry
void addEntry(int row, int col)
{
       matrix[row][col]+=1;
}

/*Process each sequence.
*Grab the patterns and update the matrix!
*/
void processSeq(int seq_num)
{
       int i,temp;
       char c;
       for (i = 0; i < pattern_length;i++)
       {
              c = (char) fgetc(inputFile);
              if (c == EOF || c == '\n')
              {
                     printf("Sequence %d is too short... exiting!\n",seq_num);
                     exit(1);
              }
              pattern[i] = c;
       }
       temp = calPos(pattern);
       addEntry(temp,seq_num - 1);
       if(matrix[temp][cols-1] == 0)
              matrix[temp][cols-1] = 1;
       while(c != '\n' && c != EOF)
       {
              c = (char) fgetc(inputFile);
              if(c == '\n')
                     break;
              if(c == EOF)
                     break;
              shift_chars(pattern, pattern_length);
              pattern[pattern_length - 1] = c;
              temp = calPos(pattern);
              addEntry(temp,seq_num - 1);
              if(matrix[temp][cols-1] == 0)
                     matrix[temp][cols-1] = 1;
       }
}


//Test file and start sequence processing
void processFile()
{
       int i;
       inputFile = fopen(fileName,"r");
       if(inputFile == NULL)
       {
              printf("Error opening the input file... exiting!\n");
              exit(1);
       }

       printf("Processing sequences:");
       fflush(stdout);
       for (i=1;i<=num_sequences;i++)
       {
              //      printf(".. %d",i);
              //        fflush(stdout);
              processSeq(i);
       }
```

```
        //  printf(" done!\n");
        printf("Finished processing all sequences!\n");
}

//close our files
void closeFiles()
{
        fclose(inputFile);
        fclose(outputFile);
        printf("All files have been closed!\n");
}

int main(int argc, char *argv[])
{
        int i;
        if(argc <= 3)
        {
                printf("Parameter(s) Missing -> [int:pattern length] [int:num sequences]
                [filename]\n");
                printf("\tAdditional Paramters (sequence scoring)\n\t-> [int: score diff.]
                [int: score pos.] [int: score neg.] [int: # of most similar sequences to
                output]\n");
                return 0;
        }
        scorediff = (int)NULL;
        scorepos = (int)NULL;
        scoreneg = (int)NULL;
        sim_num = (int)NULL;
        score = -1;
        pattern_length = atoi(argv[1]);
        num_sequences = atoi(argv[2]);
        fileName = argv[3];
        printf("USING:\t--> Pattern length: %d \n", pattern_length);
        printf("\t--> Total Sequences: %d \n", num_sequences);
        printf("\t--> Input File: %s \n", fileName);
        if(argv[4] != NULL)
        {
                score = 0;
                scorediff = atoi(argv[4]);
                scorepos = 1;
                scoreneg = -1;
                sim_num = 0;
                printf("\t--> Sequence scoring difference of %d \n",scorediff);
                if(argv[5] != NULL)
                {
                        scorepos = atoi(argv[5]);
                        if(argv[6] != NULL)
                        scoreneg = atoi(argv[6]);
                        if(argv[7] != NULL)
                        sim_num = atoi(argv[7]);
                }
                printf("\t--> Positive Score = %d | Negative Score = %d
                \n",scorepos,scoreneg);
                printf("\t--> %d Most similar sequences will be extracted!\n", sim_num);
        }
        else
                printf("NOTE: Sequences will not be scored!\n");

        printf("Contructing Matrix...");
        fflush(stdout);
        cols = num_sequences + 1;
        rows = pow(4.0, (double)pattern_length);
        pattern = (char *) malloc(sizeof(char) * pattern_length);
        matrix = (char **) malloc (sizeof(char **) * rows);
        for(i=0; i<rows; i++)
        {
                matrix[i] = (char *) malloc(sizeof(char)*cols);
        }
        printf("%d X %d Matrix contructed!\n", rows, cols);
        initMatrix(matrix, rows, cols);
        processFile();
        //printMatrix();// Only need this if we wish to output our matrix for analysis
        if(score != -1)
        {
                printf("Scoring sequences...\n ");
                fflush(stdout);
                scoreMatrix();
                printScore();// Only need this if we wish to output our sequence scoring for
                analysis
                printf("Finished scoring sequences!\n");
        }
```

```
        sleep(1);
        freeMatrix();
        if(sim_num > 0)
        {
                printf("Determining %d most similar sequences...", sim_num);
                fflush(stdout);
                seqCompare();
                printf("done!\n");
        }
        closeFiles();
        return 0;
}
```

APPENDIX 2 – SUMMARY SHEET

UNIVERSITY OF NEW BRUNSWICK
FACULTY OF COMPUTER SCIENCE

CS4997 SUMMARY SHEET
2002-2003

STUDENT NAME: Blair Foster Jr.

STUDENT SIGNATURE: _____

ID #: 265932

E-MAIL: e22b2@unb.ca

PHONE: (506) 363-4680

THESIS TITLE: COMPARATIVE Q-GRAM ANALYSIS OF GENE PROMOTER REGIONS

SUPERVISOR: Dr. Patricia Evans

DATE SUBMITTED: April 14, 2003 (original – January 14, 2002)

PHASE TITLE	ESTIMATE		ACTUAL	
	PERSON-HOURS	COMPLETION DATE	PERSON-HOURS	COMPLETION DATE
Initial Research (Molecular Biology)	20		25	02/14/03
Algorithm Analysis / Design	45		30	02/25/03
Implementation (Coding)	40		65	04/02/03
Testing / Results Analysis	15		25	04/12/03
Presentation Preparation	15		10	03/30/03
Writing Final Report	35		45	04/12/03
	Total: 170		Total: 200	

Keep this Summary Sheet for updating during the life of the thesis project. Submit a copy of the updated Summary Sheet with your Plan and attach the final version of this Summary Sheet as an Appendix in your Thesis.

You must deliver copies of the PLAN directly to both the Coordinator and the Supervisor.

BIBLIOGRAPHY

[1] S. Burkhardt, A. Crauser, P. Ferragina, H. Lenhof, E. Rivals, M. Vingron. *Q-gram Based Database Searching Using a Suffix Array (QUASAR).* RECOMB (99), pgs 77-83

[2] C. Charras, T. Lecroq. *Exact String Matching Algorithms*, http://www-igm.univ-mlv.fr/~lecroq/string/index.html [as of April 1, 2002]

[3] Cold Spring Harbor Laboratory, Yeast Promoter Database, SCPD, http://cgsigma.cshl.org/jian/ [as of April 1, 2002]

[4] B. Foster. *Gene Expression Clustering.* CS5905 Project Report, pp. 9-16 (2003)

[5] L. Heyer, S. Kruglyak, S. Yooseph. *Exploring Expression Data: Idenfication and Analysis of Coexpressed Genes.* Genome Research Vol. 9 pp. 1106-1115

[6] P. Jokinen, E. Ukkonen. *Two Algorithms for Approximate String Matching in Static Texts.* Proc. Mathematical Foundations of Computer Science 1991 (ed. A. Tarlecki), Lecture Notes in Computer Science, Vol. 520, Springer-Verlag, Berlin, 1991, pp. 240–248

[7] B. Kernigan, D. Ritchie. *The C Programming Language* 2nd Ed. Prentice Hall (1988)

[8] B. Lewin. *Genes VII.* Oxford University Press, 2000

[9] NBCI, Gene Expression Omnibus, Full-genome S. cerevisiae ORF array, http://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GPL221 [as of April 1, 2002]

[10] G. Navarro, *A Guided Tour to Approximate String Matching.* ACM Computing Surveys 33(1):31-88, 2001

[11] G. Navarro, E. Sutinen, J. Tanninen, and J. Tarhio. *Indexing text with approximate q-grams.* In Proc. 11th Annual Symposium on Combinatorial Pattern Matching (CPM'2000)

[12] E. Ukkonen. *Approximate String matching with q-grams and maximal matches.* Theoretical Computer Science, 92(1):191–211 (1992)

[13] National Institue of Standards and Technology. *big-O notation.* http://www.nist.gov/dads/HTML/bigOnotation.html [as of April 1, 2003]

[14] National Institue of Standards and Technology. *Omega.* http://www.nist.gov/dads/HTML/omegaCapital.html [as of April 1, 2003]

[15] National Institue of Standards and Technology. *Theta.* http://www.nist.gov/dads/HTML/theta.html [as of April 1, 2003]

[16] A. Puri, N. Kataria. *Approximate String Join*.

[17] J. Setubal, J. Meidanis. *Introduction to Computational Molecular Biology*. Brooks/Cole Publishing Company, 1997

[18] S. Skiena. *The Algorithm Design Manual*. Springer-Verlag New York, Inc. (1998)

[19] G. Stephen. *String Searching Algorithms*. Singapore; River Edge, NJ: World Scientific, c1994.