University of Nebraska - Lincoln

DigitalCommons@University of Nebraska - Lincoln

Computer Science and Engineering: Theses, Dissertations, and Student Research

Computer Science and Engineering, Department of

USE OF CLUSTERING TECHNIQUES FOR PROTEIN DOMAIN ANALYSIS

Eric Rodene

University of Nebraska - Lincoln, erodene@cse.unl.edu

Follow this and additional works at: http://digitalcommons.unl.edu/computerscidiss

Part of the Biochemistry, Biophysics, and Structural Biology Commons, Bioinformatics Commons, Computer Engineering Commons, and the Computer Sciences Commons

USE OF CLUSTERING TECHNIQUES FOR PROTEIN DOMAIN ANALYSIS

by

Eric T. Rodene

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfillment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professors Stephen D. Scott and Etsuko N. Moriyama

Lincoln, Nebraska

July 2016

USE OF CLUSTERING TECHNIQUES FOR PROTEIN DOMAIN ANALYSIS

Eric Rodene, M.S.

University of Nebraska, 2016

Advisors: Stephen Scott, Etsuko Moriyama

Next-generation sequencing has allowed many new protein sequences to be identified. However, this expansion of sequence data limits the ability to determine the structure and function of most of these newly-identified proteins. Inferring the function and relationships between proteins is possible with traditional alignment-based phylogeny. However, this requires at least one shared subsequence. Without such a subsequence, no meaningful alignments between the protein sequences are possible. The entire protein set (or proteome) of an organism contains many unrelated proteins. At this level, the necessary similarity does not occur. Therefore, an alternative method of understanding relationships within diverse sets of proteins is needed.

Related proteins generally share key subsequences. These conserved subsequences are called domains. Proteins that share several common domains can be inferred to have similar function. We refer to the set of all domains that a protein has as the protein's domain architecture.

We present a technique which clusters proteins sharing identical domain architecture. Matching a domain to a protein is determined with a confidence estimate (*e.g.*, the E-value). The confidence with which a domain is matched to the sequence varies widely. By using a threshold for what is considered an acceptable match, domains with weak similarities can be ignored. By changing this E-value threshold, the clustering patterns and relationships between proteins can be analyzed. Clusters may merge or split

as their domain architecture shifts based on this threshold. By studying the relationships between clusters from one iteration to the next as the threshold is made more stringent, phylogeny-like networks can be constructed. This technique clusters together proteins with identical domain architecture, and also illustrates relationships among clusters with similar architecture.

This technique was tested on the multi-domain Regulator of G-protein Signaling family. The output is consistent with the known functional subdivisions of this protein family. This technique is also considerably faster than typical alignment-based phylogenetic reconstruction on this family. Use of the technique at the proteome level was also tested using bacterial proteome data from *Bacillus subtilis*.

Table of Contents

List of Figures

List of Tables

CHAPTER 1
INTRODUCTION

1.1 Overview

With the rapid expansion of data made available from next-generation sequencing, many new protein sequences have been identified, and even whole proteomes of various organisms are available to study. Unfortunately, knowledge of the three-dimensional structure and function of most of these newly-identified proteins generally tends to be limited. There are some options for studying such proteins, with the goal of finding other sequences whose functions are already known and that the proteins under study are similar to. One common approach would be to use the Basic Local Alignment Search Tool (BLAST) [1], which is made available through the National Center for Biotechnology Information (NCBI) website [2]. BLAST uses local alignment searches against a database of proteins or nucleotide sequences to find highly similar hits. The assumption is that sequences with high similarity are likely to be closely related and to share similar functions as well.

Another approach is to use alignment-based phylogenetic methods. Often used multiple sequence alignment methods include ClustalW [3], MUSCLE [4], and MAFFT [5]. Phylogenetic reconstruction can be performed using methods such as neighbor-joining [6], maximum parsimony [7, 8], and maximum likelihood [9, 10, 11]. A traditional approach to understanding the relationships within a set of proteins would be to perform an alignment and then use the alignment as the basis for generating a phylogenetic tree. However, this approach requires that the sequences under study align with one another. There would need to be at least one significant region within each of the sequences that is highly conserved. Without such regions of similarity, no useful

alignment is possible. As a result, this approach is only applicable to closely-related sequences, and would fail if it was attempted on a diverse set of proteins, such as what is encountered in an organism's proteome. It is therefore necessary to employ an alternative method for this purpose.

Many proteins are composed of smaller subsequences called domains. A domain is an amino acid sequence that may be present in a protein and which tends to be highly conserved between different related proteins, even though the rest of the protein sequences are overall more divergent. The amino acid sequence of a protein can determine how it is folded into its three-dimensional structure, due to the chemical interactions between each amino acid molecule. The presence of conserved domains in a set of proteins can be used to infer that they may share some similarities in three-dimensional structure. The structure of a protein determines its function. As a result, it is possible to infer structural similarity and related function between various proteins from the presence of shared domain sequences.

Alignment-based phylogenetic methods of clustering proteins are well known, and have been in common use for years. However, they typically rely on analyzing the whole protein sequences, or extracted subsequences. Tribe-MCL is another clustering method that does not rely on global alignment of protein sequences [12]. It has been used for proteome-level protein clustering. However, this method does not consider overall domain architecture, instead only focusing on the strongest domain match present in a given protein. While such a domain match may be diagnostic for including a protein within a particular family of related proteins, this does not consider the modifying effects that the presence or absence of any other domains may have on its overall function. As

discussed above, it may be possible for some proteins with similar domain architecture to be functionally similar, even though their overall sequences may be more divergent. By using a domain-based method of protein clustering that focuses on the full domain architecture, it may be possible to identify similarities in function that might not be obvious from alignment-based methods, due to this divergence.

Protein domain clustering has previously been investigated by Enright, *et al.* [12] and Shah [13]. Shah's work focused on using a biclustering algorithm called Bimax [14] to group proteins into clusters based on the similarity of their domain architecture. Bimax uses a simple binary matrix to construct its clusters. However, the strength of the domain's matching also factors into the relationships between proteins. Therefore, it is not sufficient to say that a domain is simply present in a protein. By somehow incorporating domain similarity information, in addition to domain architectures, into the clustering method, it would be possible to see not only whether or not a domain is simply present in a protein of interest, but also how the strength of its matching can affect how it is clustered with other proteins.

1.2 Objectives

The primary objectives of this thesis are to develop a domain-based tool for clustering that can be used as an alternative to more typical alignment-based methods of phylogenetic reconstruction or Tribe-MCL. In particular, such a tool is not only useful for analyzing the relationships between members of an organism's proteome, without the need of a universally shared subsequence, but it also conducts analysis on the functional relationships between smaller datasets more rapidly than a maximum likelihood phylogenetic analysis can be conducted on the same dataset.

The overall contributions that have been made by this thesis are:

- The creation of a Matlab implementation of Bimax.

- The creation of a Matlab implementation of a bit vector clustering method. This implementation iterates by decreasing the accepted domain similarity (E-value) threshold. The output shows the cluster membership of proteins and domains present in the cluster architecture.

- Visualizations of bipartite matchings for each iteration are generated. A final "tier"-based graph showing the "inheritance" of proteins between clusters in adjacent iterations is also generated, as well as Newick trees stored in plain text files displaying phylogeny-like relationships between the clusters as derived from this "tier" chart.

The program was tested using the mouse regulator of G-protein signaling (RGS) protein dataset and *Bacillus subtilis* proteome. The output of the program was evaluated based on how successfully the resulting trees grouped together proteins with related function.

1.3 Organization of the Thesis

The rest of the thesis is organized as follows:

Chapter 2: Background and Related Work

Section 2.1: Background

This section focuses on the background and primary motivation behind this thesis, as well as some of the early steps undertaken.

Section 2.2: Related Work

This section provides a summary of some other algorithms that have been developed for biclustering methods, as well as one for use in protein domain analysis.

Chapter 3: Methods and Pipeline Overview

This chapter goes over the fine details of what methods the bit vector approach uses, as well as a step by step overview of how it goes about clustering the input proteins and building the resulting phylogeny-like trees.

Chapter 4: Results and Discussion

4.1: Datasets Used

This section covers a brief overview of the RGS and *B. subtilis* datasets used in the study.

4.2: Evaluation Methods

This section introduces how the quality of the resulting trees was assessed.

4.3: RGS Data

This section covers the output of the bit vector approach on the RGS data, as well as the output of several runs using maximum likelihood and neighbor-joining on the same data.

4.4: Discussion of RGS Data

This section discusses the key findings of the RGS data output, as well as discussing comparisons to the performance of alternative algorithms.

4.5: *B. subtilis* Data

This section describes a small selection of the trees provided in the output of the bit vector program on the *B. subtilis* data.

4.6: Discussion of *B. subtilis* Data

This section describes some of the key findings of the *B. subtilis* data output.

Chapter 5: Conclusion and Future Work

This chapter discusses the main conclusions of this thesis, and how the proposed bit vector program compared to other methods.

5.1: Future Work

This section provides a discussion of some proposed improvements that could be made to the bit vector approach.

CHAPTER 2
BACKGROUND AND RELATED WORK

2.1 Background

The main goal of this thesis is to further the work of Shah [13]. Shah's protein-domain clustering method focused on biclustering techniques using a C implementation of the algorithm Bimax [15]. Her technique was tested using a sampling of the multi-domain Regulator of G-protein Signaling (RGS) family of proteins from *Mus musculus*, as well as multiple proteomes from both prokaryotes and eukaryotes.

Originally, it was intended to find and test an alternative algorithm which can allow the strength of the domain match to be considered, as opposed to a simple binary presence-absence test (as Bimax does). In Oghabian, *et al.* [16], several biclustering algorithms were compared. Although most of the algorithms discussed were designed for gene expression biclustering, it was hoped a way might be found to run a selected algorithm using protein domain data as input instead. The above review assessed various qualities of the sampling of algorithms under investigation, including how successful they were in differentiating different sample types, how well the groups of genes in the algorithms' results are annotated with similar gene ontology categories, how well the algorithms were able to differentiate genes known to be specific to particular sample types the authors used in the study, and also running time of the algorithms. The review determined that techniques such as Plaid and SAMBA were the most useful and reliable methods assessed in the study.

Based on the results of the aforementioned study, SAMBA was originally considered for the alternative algorithm. It is not a standalone algorithm, and comes bundled in a larger package of gene expression and microarray analysis methods known

as EXPANDER. This package was designed by Shamir, *et al.* [17], who also wrote an overview of the SAMBA algorithm itself [18]. EXPANDER is freely available online [19]. Ultimately, after several tests on the input data based on the RGS proteins, it was found that the clusters generated by SAMBA did not match the clusters that Shah obtained using Bimax closely enough to be of use for this project. For example, several distinct clusters in Bimax's results were often combined together using SAMBA. As a result, the use of SAMBA was abandoned.

2.2 Related Work

Biclustering methods as well as protein domain analysis tools have been previously investigated by a number of authors.

Király, *et al.* [20] developed a biclustering using bit-tables. This method is very similar to the technique used by Bimax and the bit vector approach described in this thesis. The bit-table method proposed by Király is given as a Matlab implementation, and uses matrix and vector multiplication in order to discover the biclusters efficiently. The tests presented indicate the algorithm outperforms Bimax in all cases.

BicOverlapper [21] is a gene expression analysis tool that visualizes key aspects of the analysis process, such as the expression data, profiling, and annotation. It integrates several techniques into one convenient package. Its main contribution is to provide useful visualizations based on results of biclustering algorithms on gene expression data.

Another biclustering method is Bi-Force [22]. This technique uses a weighted bicluster editing model, and was compared against other biclustering algorithms (FABIA, QUBIC, Cheng and Church, Bimax, Spectral, xMOTIFs, and ISA) on synthetic and real-

world gene expression datasets. It generally performed favorably against the other algorithms, although Spectral was shown to be consistently faster. Generally, however, the quality of results that Bi-Force reported was considered to be better that that of the other algorithms.

BicSPAM [23] is a biclustering technique that was proposed with the intention of being a more robust order-preserving biclustering algorithm than other methods previously available. It was evaluated based on its ability to capture bicluster symmetries, handle noise, and scalability. The authors report that BicSPAM surpasses the issues found in other order-preserving methods, and was shown to be both flexible and robust in terms of noise and expression profiles.

A tool called Furby [24] is presented as a visualization technique for biclustering results. The technique offers an overview of the results of gene expression biclustering, showing what data forms the clusters together, and also provides the ability to set thresholds to form "fuzzy" clusters into "hard" clusters that can be studied with other methods, such as bar charts.

Finally, DoMosaics [25] is a protein domain analysis program that is intended for comparison and visualization of domain architectures. Its primary contribution is that it combines domain annotation, homology search, analysis of domain architecture evolution, and visualization into a single convenient tool.

CHAPTER 3
METHODS AND PIPELINE OVERVIEW

It was initially decided to use Bimax, but to test varying the maximum threshold of the permissible E-values in the input data. The E-value, or expect value, is a confidence estimate widely used in bioinformatics tools. Examples include BLAST [1] and HMMER [26, 27]. The E-value is used to describe the likelihood of seeing positive hits of equal or better score due to random chance in a database of a given size. The closer the E-value is to 0, the more significant the match is. As a result, smaller E-values are more desirable than larger ones. By varying the accepted E-value threshold of a protein-domain matching, it would be possible to see how the clusters change as low-scoring domains are systematically deleted and Bimax is re-run on the modified data. To this end, a new implementation of Bimax was written, which was ultimately embedded inside a larger script allowing the method to be systematically called on increasingly more stringent input data. All code has been written in Matlab. This programming language was selected primarily due to the streamlined way it handles matrices, as well as ease of importing / exporting data to and from files.

Upon completion of this implementation of the core method, several test input matrices were created. This data was then passed to both the Matlab implementation of Bimax, as well as the original C implementation [15]. The results for each test were compared to ensure the Matlab implementation was indeed operating correctly. For both implementations, the clusters reported were identical for each test case. With the core method functioning as expected, further scripting was done to produce a pipeline. Data taken from HMMER results of a selection of proteins is fed into a script, which then imports this data into a cell array. Certain columns are accessed, including the protein

name, domain it is matched to, and the E-value of the match. This data is used to

construct a simple $m \times n$ binary matrix, where m corresponds to the number of unique

proteins under investigation, and n corresponds to the number of unique domains present

in the entire set of m proteins. Thus each row of the matrix corresponds to one protein,

while each column corresponds to one domain.

If the value of a cell i, j is set to 0, it indicates that domain j is not present in

protein i, according to the results from HMMER at the current E-value threshold. It

should be noted that an E-value of 1.0 was selected as the maximum permissible value,

values higher than this threshold were not considered. Otherwise, if the cell's value is 1,

it indicates HMMER has reported domain j is present in protein i. This matrix is then

exported to a plain text file, which is then used as the input for the implementation of

Bimax. After each iteration of a matrix through Bimax, the sorted list of E-values is used

to delete all entries with an E-value of the same order of magnitude as the poorest-score

left in the list. It accomplishes this by iterating over the input matrix and changing any

entries at the deletion threshold from 1 to 0. For example, if the current highest (and

therefore poorest score) E-value is 2.3×10^{-8}, then all entries with E-values at the order

of magnitude of 10^{-8} are similarly removed. The updated matrix is then passed back to

Bimax and the new set of clusters is found. This process repeats until either all

remaining E-values are at the same order of magnitude (and so all such entries would be

deleted if the algorithm were to iterate one more time), or the current order of magnitude

reaches a user-defined threshold, used as a termination condition.

The results reported by Bimax are inclusion maximal, that is, in the event a

protein has a set of domains A, B, C, for example, it will be clustered not only with all

proteins possessing all three domains as their total domain architecture, but will also be clustered with the set of proteins containing domain *A*, the set of proteins containing domain *B*, the set of proteins containing domain *C*, the set containing domains *A* and *B*, and so on. This means an individual protein could potentially participate in more than one cluster.

The current version of the clustering implementation has been used to compare the results by this method with those by Shah's [13]. In the process of this comparison, it was determined that because Shah focused on the complete domain architecture of each protein, rather than only subsets of a protein's domains (as seen in results from inclusion maximal clusters), it was determined that post-processing of the Bimax results would be required to eliminate clusters that did not involve all possible domains for some of the member proteins, with the intent to focus only on clusters representing the entire domain architecture of the member proteins. Rather than do this, it was instead decided to create an alternate implementation that used bit vectors instead, and cluster proteins together only if they shared all of their domains, and not just subsets of them. The main reason for this decision was it was believed it would be a more efficient approach to simply cluster the proteins in non-inclusion-maximal groupings that only focus on the total domain architecture for each group as the criterion for inclusion, rather than the presence or absence of individual domains without regard to the overall architecture, as Bimax does.

The core approach is as follows. Each protein is assigned a bit vector derived from the binary matrix discussed above. The number of bits is determined by the total number of unique domains present in the entire set of proteins being clustered. Suppose the entire protein selection has 50 domains. Then the binary string or bit vector will

consist of 50 bits. The index of each bit in the string will correspond to a particular domain in the selection. As discussed previously, a value of 0 indicates that particular protein does not have that domain present, according to the HMMER data provided as input and the current E-value threshold. A value of 1 indicates the domain is present for that particular protein. A simple comparison of the strings of each protein will then allow proteins sharing identical strings to be clustered together.

The comparison process is as follows: protein 1 is compared with protein 2, 3, 4, and so on until the mth protein. Then protein 2 is compared to protein 3, then 4, and so on to the mth protein. Then protein 3 is compared to protein 4, 5, and so on. Each pair of proteins is compared exactly once, so the number of comparisons is reduced with each new target protein under comparison. In the event that a string being compared to the target string is found to be identical, the proteins are placed in the same cluster, and the bit vector is deleted from the list, guaranteeing they are not unnecessarily compared again, as their proteins cannot participate in any additional clusters. As a result, the time complexity of this core method is $\Theta(m^2)$.

In addition to what is described above, the process is repeated with increasingly stringent E-value thresholds. The E-value threshold is here defined as the maximum permissible E-value for inclusion in the computational process. Any E-values higher than that threshold are not considered. For example, on the first iteration, only E-values strictly less than 1.0 are of interest. This has been arbitrarily chosen as a threshold for which E-values are appropriate for inclusion. Values of 1.0 or higher are deemed too weak to warrant attention. It should be noted that the HMMER input data has two possible E-values, the conditional E-value and the independent E-value. The independent

E-value (or i-Evalue) is defined as the significance of the sequence in a search of the entire database, if the domain the E-value is associated with were the only domain identified [28]. The conditional E-value (or c-Evalue), on the other hand, measures the statistical significance of each domain given that the target sequence has already been decided to be a true homolog. Thus it is the expected number of additional domains that could be found with the same domain score due to random chance. The i-Evalue, which is more regularly used in HMMER, was used in this work. For each subsequent iteration, the order of magnitude of the highest remaining E-value is located. For instance, if this E-value were to be 2.1×10^{-8}, then the order of magnitude is 10^{-8}. All E-values remaining that are at that same order of magnitude are removed from the matrix during each iteration. For example, if Protein X were to possess domains A, B, and C, suppose Domain C was matched with an E-value of 2.1×10^{-8}, and domains A and B were matched with much stronger E-values several orders of magnitude smaller than this value. After the iteration in which all E-values with the order of magnitude 10^{-8} are removed, Protein X would no longer have the domain architecture A, B, C, but would instead now have the architecture A, B, as Domain C has now been removed by the threshold cutoff. In this way, the threshold of what E-values are deemed acceptable is constantly decreased (made more stringent) as the algorithm progresses. This has the effect of changing which bits are determined to be 1, as the permissible threshold of inclusion changes, and therefore also has the effect of deleting weakly-scoring edges (representing the protein-domain matchings) in the resulting bipartite graph with each iteration.

As stated before, the method has a core time complexity of $\Theta(m^2)$ for its bit vector comparison, given *m* bit vectors (one for each protein). However, this process is repeated for each E-value iteration, and so the time complexity would be $\Theta(em^2)$, where *e* represents the number of E-value iterations, determined by the values present in the HMMER data. However, technically the way the code is implemented, for each of the *n* domains, each of the *m* bit vectors requires iterating over each individual bit to generate the binary strings on which the comparisons are based. As a result, the true time complexity is actually $\Theta(em^2n)$ to cluster the proteins over all of the E-value iterations. Furthermore, the construction of the final trees requires iterating over each of the *m* proteins and performing pairwise comparisons to determine which proteins have the same Newick strings as they are being constructed. This must be repeated for each of the *e* iterations until the final Newick string is constructed for each protein. This procedure therefore also has a time complexity of $\Theta(em^2)$.

The principal output of the algorithm is a set of graph files that can be viewed using Gephi [29, 30], an open-source program intended for network visualization, which also provides capabilities such as exploratory data analysis, link analysis, and biological network analysis. The basic layout of the graph is to have the protein nodes lined up in the left-hand column, and the domain nodes lined up in the right-hand column, with the edges connecting the two columns. Each edge represents that a given protein has the connected domain according to the HMMER data and the E-value threshold of that iteration. In other words, if the binary matrix for that iteration has a 1, an edge is present between that protein and that domain. In addition, the edge weight w_{ij} of the *i*th protein connecting with the *j*th domain is found according to the formula:

$$w_{ij} = 10 \cdot \left[\frac{-\log_{10}(E_{ij})}{100}\right]$$

where E_{ij} is the E-value of the ith protein matching with the jth domain (taken from the HMMER input). The value of 100 is generated from $-\log_{10}(1.0 \times 10^{-100})$, where 1.0×10^{-100} is arbitrarily chosen as the "best" E-value possible. In the event E happens to be smaller than 1.0×10^{-100}, it is mapped back to this value, and so the resulting weight will be 10 (from $10 \cdot \left(\frac{100}{100}\right)$, or $10 \times 1 = 10$). The value is multiplied by a factor of 10 to allow for easy viewing in the Gephi visualizations. If the value is left as a decimal between 0 and 1, the edge weight thickness will not be easily distinguished in the resulting visualization.

The files to be used in Gephi are generated in GEXF format. GEXF stands for Graph Exchange XML Format. As its name suggests, the content of the file is XML code which specifies the properties of the nodes and edges. By using this format, the algorithm's output could be changed to precisely specify the properties of each node – particularly position and color [31]. This allows the generation of the Gephi graphs to be automated, thereby increasing the efficiency of the process. The current version presents each cluster of protein nodes by grouping all nodes of a given cluster next to each other with extra white space above and below that cluster to visually separate it from the others.

Plain text files are also generated displaying the final trees found by the algorithm. Tree outputs are expressed using the Newick format, which is regularly used for phylogenetic trees. The string "$((A,B),C)$" is a simple example of Newick format. It indicates that A and B are sister taxa, and that together they form a branch which itself is

sister to the branch containing *C*. **Figure 3.1** provides a visualization of this tree. These files can conveniently be imported into a tree visualization program to be displayed. There is both an un-labelled version, without branch lengths, node labels, or domain architecture presented, and also a labelled version that does show these features. TreeView [32] was used to generate the final tree visualizations.
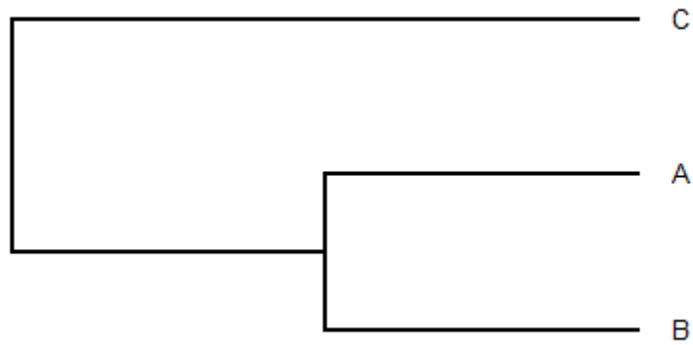
Figure 3.1: A visualization of the Newick tree ((*A*,*B*),*C*).

The basic procedure of the algorithm described above is summarized in **Figure 3.2**.

1. Read input data.
2. Reformat data to a tab-delimited file for importing into Matlab.
3. Read reformatted file.
4. Extract relevant fields (domain name, protein name, i-Evalue).
5. Build a list of proteins and domains present in the data.
6. Output protein and domain lists to file.
7. Construct a binary matrix indicating which domains of the set are present in which proteins.
8. Create a list of nodes for import into Gephi.
9. Iterate until no entries remain, decreasing the E-value threshold:
   a. Delete poor E-values in the binary matrix (if applicable).
   b. Run the bit vector procedure.
   c. Output edge data (for Gephi) and clusters to file.
10. Generate data for tier chart showing inheritance of proteins of each cluster between adjacent iterations.
    a. If cluster remains unchanged between iterations, a single edge is drawn between nodes.
    b. Else if cluster is the result of the merging of two or more previous clusters, draw edges converging on the new merged cluster.
    c. Else if cluster splits into two or more clusters between iterations, draw edges to the descendant clusters.
11. Build Newick trees from the tier chart data.
12. Output Newick trees to file.
13. Output tier chart.

Figure 3.2: Basic pseudocode of the clustering procedure.

CHAPTER 4
RESULTS AND DISCUSSION

4.1 Datasets Used

The bit vector program was tested on both the RGS and *B. subtilis* data from HMMER. The RGS dataset from *Mus musculus* was composed of 66 proteins with a total of 54 domains, as identified by HMMER. The *B. subtilis* proteome data consisted of 3,973 proteins. HMMER identified a total of 4,737 individual domains.

4.2 Evaluation Methods

Clustering patterns were evaluated based on how the clusters correlated with protein function. Here we define a protein's function as its role in the organism. This is usually given as the protein's description or name associated with its accession number on databases such as NCBI. For example, NP_061357.3 is a type of kinase (specifically, a G-protein coupled receptor kinase). This distinguishes it from other protein functions, such as hemoglobin, which binds to and transports oxygen in the bloodstream, or cytochrome, which participates in the electron transport chain to produce ATP. The results were evaluated for how well the individual trees encompass the proteins of the associated functionality. For example, Tree 5, presented in **Figure 4.3.4**, encompasses all of the proteins (in red) possessing the regulator of G-protein signaling (RGS) function (as opposed to the RGS domain or the RGS protein family itself). Furthermore, there are no proteins of this functional type found in any of the other RGS family trees.

It would be expected that in a given dataset, all proteins with a given functional class, such as kinases or axin proteins, would be grouped within a particular tree, rather than some members being clustered in separate trees. It would also be expected that all

proteins within a tree would possess a similar functional class, rather than the tree consisting of proteins with a variety of unrelated functions. Furthermore, it would be expected that members of one branch of a tree would exhibit more similarity in function to each other than they would to members of other branches, although the members of separate branches would still share a related function in some way. An example might be a two-way split in a tree with kinases of one type in one branch, and another kind of kinase in its sister branch. Each of the different kinds of kinases in this example would be expected to cluster together within their own branches, rather than being scattered in separate branches.

The homogeneity of each of the trees can be calculated. Trees that consist entirely of proteins with the same function have 100% homogeneity. If a tree were to consist of ten total proteins, and two of them had functions that differed from that of the other eight, then the tree's homogeneity would be 80%, as eight out of the ten proteins share the same function. The homogeneity (h) calculations follow the formula $h = x_{in}/x_{tot}$. x_{in} represents the number of in-group members of any given tree, defined here as the largest subset of included proteins sharing the same function (the dominant function of that tree), and x_{tot} represents the total number of proteins in the tree. Thus, larger percentage homogeneity means a given tree contains fewer outliers not belonging to the tree's dominant protein function.

Similarly, it is important to assess how disjointed or discontinuous a protein function is. If a tree were to have all proteins in the set with a given functionality contained within it, then that functional type would not be disjointed at all. However, if a tree were to consist of eight proteins of a given function, but another tree contained two

more proteins with the same function, then that functional type would be said to be 20%

disjointed, as two of the ten proteins do not occur within the tree where the majority of

the member proteins are placed. The discontinuity (d) calculations follow the formula

$d = y_{out}/y_{tot}$. y_{out} represents the total number of member proteins of any given

function that are found outside the tree where the majority of members of that same

function are found, and y_{tot} represents the total number of proteins of that function.

Thus, smaller percentage discontinuity means a given function has more of its members

contained within the same tree.

The average Jaccard index, described in [14], is an assessment of the similarity

between the results of two clustering methods. In this case, we compare the trees

(including those consisting of only one cluster) generated from the RGS data from the bit

vector approach to the clusters found in the maximum likelihood trees. A maximum

likelihood cluster is defined as a branch having a node with at least a 70% bootstrap. In

such a case, all proteins within that branch are considered to be clustered together. We

refer to set B as the set of bit vector trees, and set M refers to the set of maximum

likelihood clusters. The Jaccard index was compared between the bit vector approach

and the maximum likelihood approaches using both the MUSCLE and MAFFT

alignments. The formula used is as follows:

$$S(B, M) = \frac{1}{|B|} \sum_{B_1 \in B} max_{(M_1) \in M} \frac{|B_1 \cap M_1|}{|B_1 \cup M_1|}$$

4.3 RGS Data

In addition to the Newick tree visualizations shown in **Figures 4.3.8-4.3.15**, the

program also outputs a "tier"-based chart, as shown in **Figures 4.3.1a-4.3.1d**. The

construction of this chart also serves as the basis for constructing the Newick trees. It should be noted that this chart does not necessarily result in the creation of one large tree, but can instead (as it did with the RGS data) result in several smaller trees with no overlap between their contents. Many of the cluster groups connected by edges in the chart may never merge with each other, and so they remain separate in the final trees.
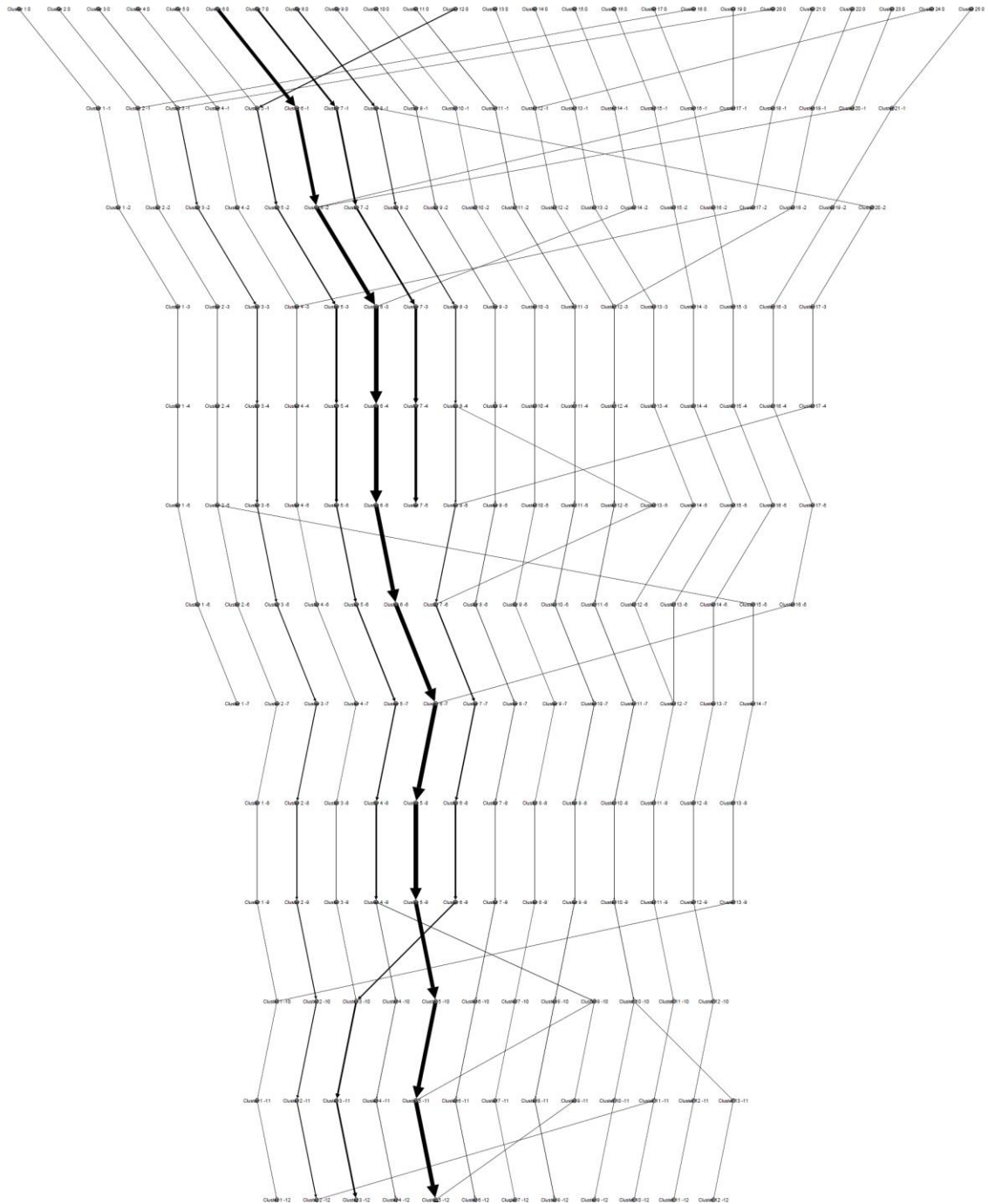
Figure 4.3.1a: First portion of the tier chart for the RGS data, from iteration 0 to iteration -12.
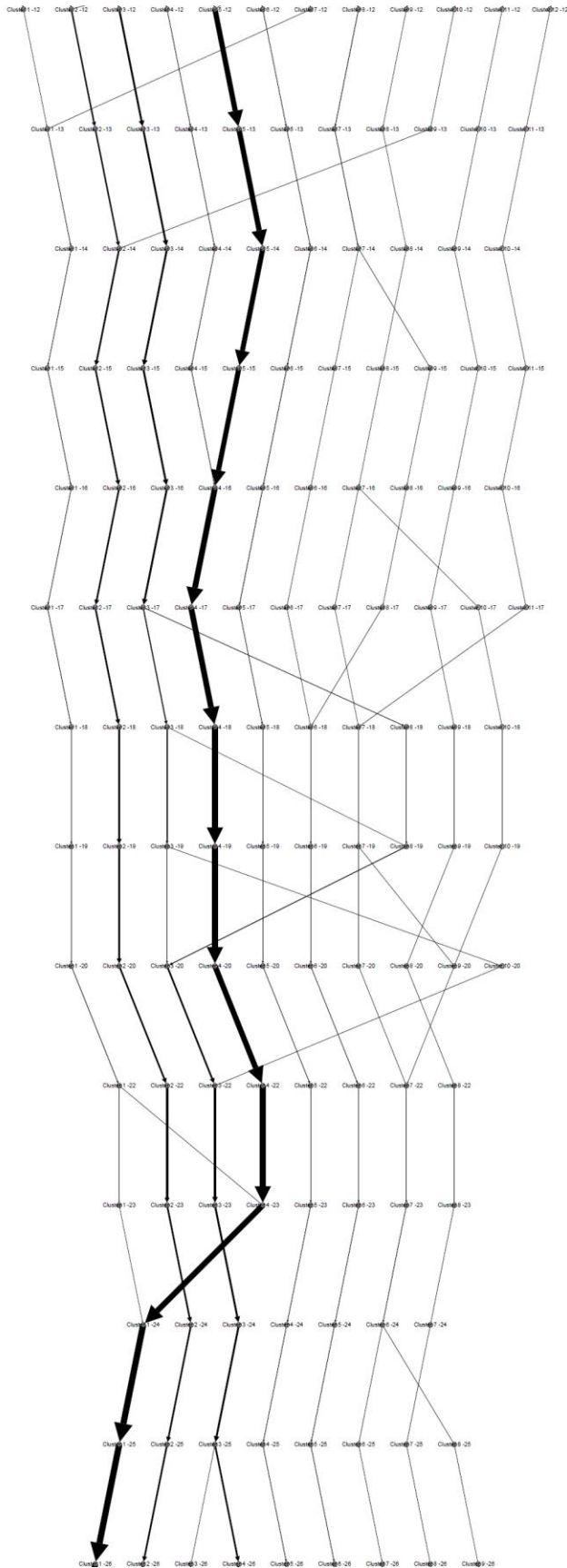
Figure 4.3.1b: Second portion of the tier chart for the RGS data, from iteration -12 to iteration -26.
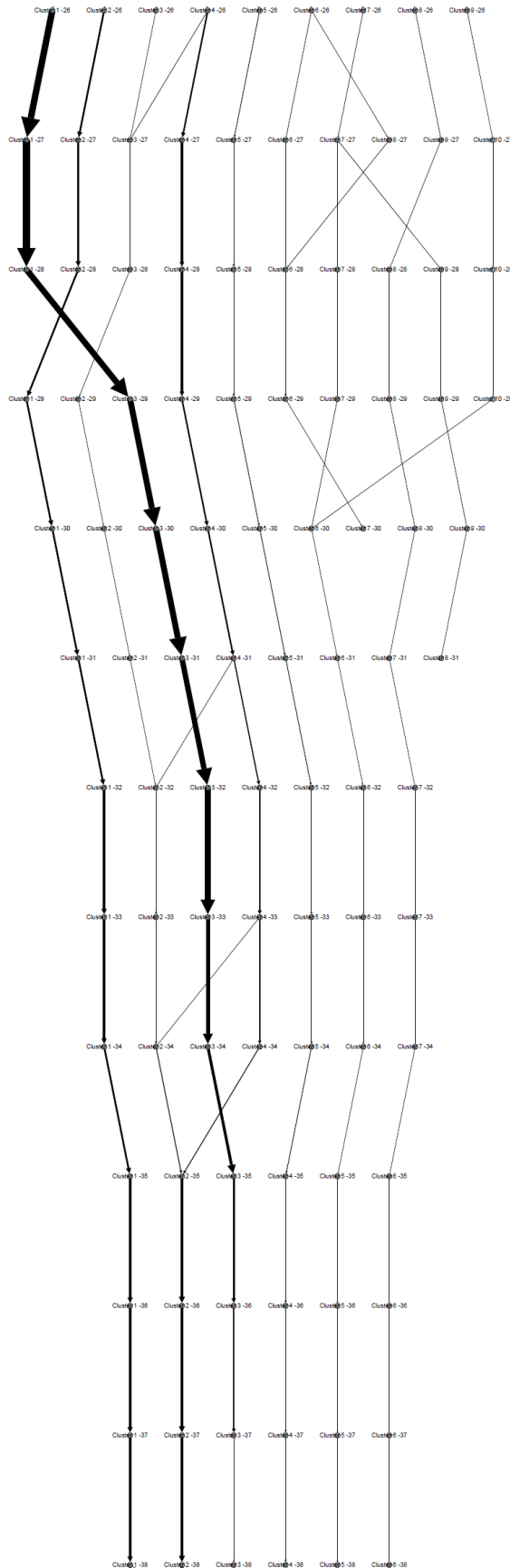
Figure 4.3.1c: Third portion of the tier chart for the RGS data, from iteration -26 to iteration -38.
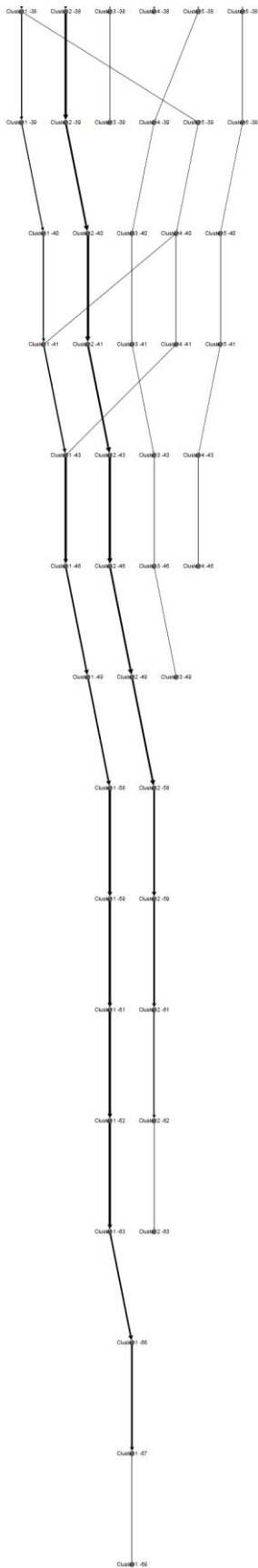
Figure 4.3.1d: Final portion of the tier chart for the RGS data, from iteration -38 to iteration -69.

Each "tier" or row of nodes in the chart represents the individual clusters found in a particular iteration. The cluster nodes are labelled in the format "Cluster *X Y*", where *X* is the cluster number for that iteration, and *Y* represents the order of magnitude of E-value that was deleted at the start of the iteration, or 0 for the initial iteration. For example, if the current order of magnitude is 10^{-5}, then Cluster 3 would have its node labelled as "Cluster 3 -5". Edges are drawn between nodes in adjacent iterations only if a cluster in one iteration "inherits" at least one protein from a cluster in the previous iteration. In addition, edges will be thicker if more proteins are inherited between the connected nodes. We refer to a lineage as the set of edges connecting all participating cluster nodes, and continuing to a final node at some point (a dead end). Dead ends of a lineage indicate that no proteins in its final cluster remain with any acceptable domains according to the threshold for the next iteration. Similarly, some proteins participating in a lineage will also periodically be removed along the way, even though other proteins remain accepted by the threshold and still continue to subsequent iterations.

Parts of a lineage may split, merge, or remain unchanged from one iteration to another. For example, a cluster may lose one or more domains that have thus far prevented its member proteins from being included in another cluster. Once these domains are lost due to the threshold, the architectures of the two clusters will be identical, and they will merge together. Similarly, a single cluster may split apart. This happens because one or more of its proteins may lose a domain before any other proteins in the same cluster lose the same domain. As a result, the architectures are no longer identical, and so a new cluster must be formed to contain the diverging proteins. In cases where no splitting or merging occurs, an edge will connect a cluster node with the

updated version of the same cluster for the next iteration. This does not necessarily mean the domain architecture or protein membership is identical between the two nodes, it just means that nothing happened to result in the cluster merging with another cluster or splitting into two or more new clusters. Some proteins could have been removed due to E-value threshold, and even one or more domains could have been deleted. If this occurs, as long as all of the surviving member proteins have the same domain deleted at the same threshold, no discrepancy in architecture will occur among the member proteins, and so no splitting will occur, and as long as the updated architecture does not match that of any other cluster, no merging will occur either.

There are several examples of the above behavior in the RGS tier chart. **Figure 4.3.2** provides an example of merging, **Figure 4.3.3** gives an example of splitting, **Figure 4.3.4** shows an example where neither splitting nor merging occurs, and **Figure 4.3.5** shows a dead end in a lineage.
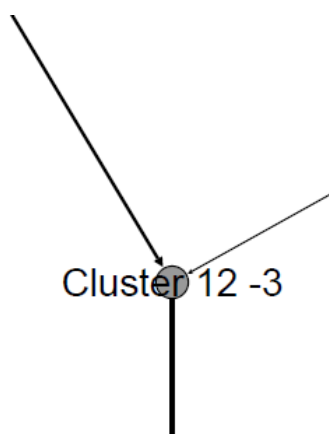
Figure 4.3.2 (left): Cluster 12 -3 is formed when Cluster 12 -2 and Cluster 18 -2 merge together.

Cluster 12 -3
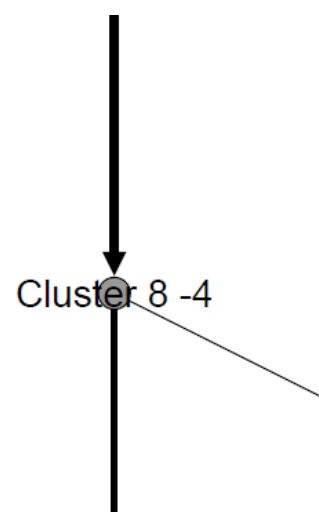
Cluster 8 -4

Figure 4.3.3 (right): Cluster 8 -4 splits, with some proteins forming Cluster 8 -5 and others forming Cluster 13 -5.
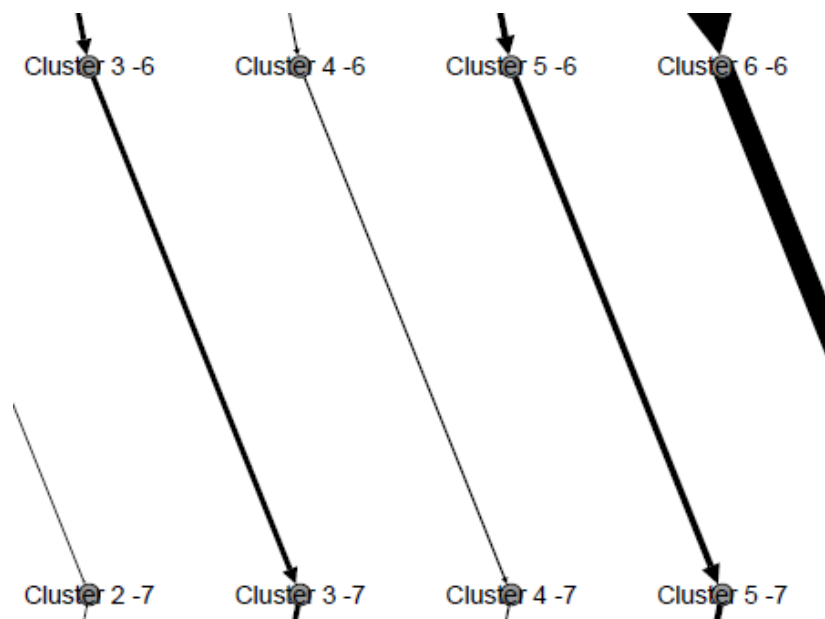
Figure 4.3.4: Examples of no change between iterations on three separate lineages. Differences in edge weight can also be seen.
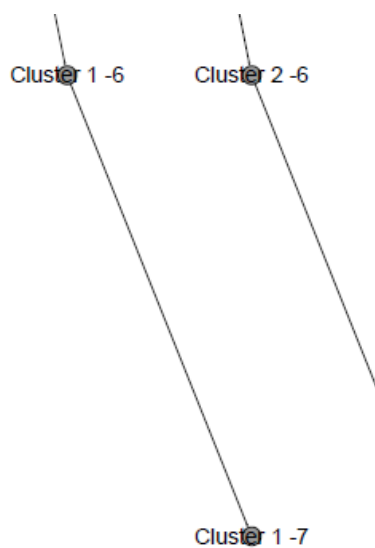
Figure 4.3.5: Cluster 1 -7 is the final node of this lineage, even though other lineages continue to subsequent iterations.

The behavior of each lineage in the tier chart is important, because this is the basis for how the Newick trees are formed. The leaves of each tree are formed from the participating clusters in the initial tier of the chart, and dead ends indicate the presence of the tree's root. Lineages that never split or merge form singleton trees, for example. Merges are probably the most important, because these form the basis for nodes in the

Newick tree. For example, the merging of two clusters forms a two-way split in the resulting tree. **Figure 4.3.6** shows another view of this behavior between Cluster 5 0 and Cluster 12 0. The Newick tree formed from this places Cluster 5 and Cluster 12 on the same branch as sister groups (see **Figure 4.3.14**). It is also possible for more than two clusters to merge at once, which results in a polytomy, such as between clusters 6, 19, and 23 in **Figure 4.3.14**. **Figure 4.3.7** shows the point in the tier chart where this merging occurs.
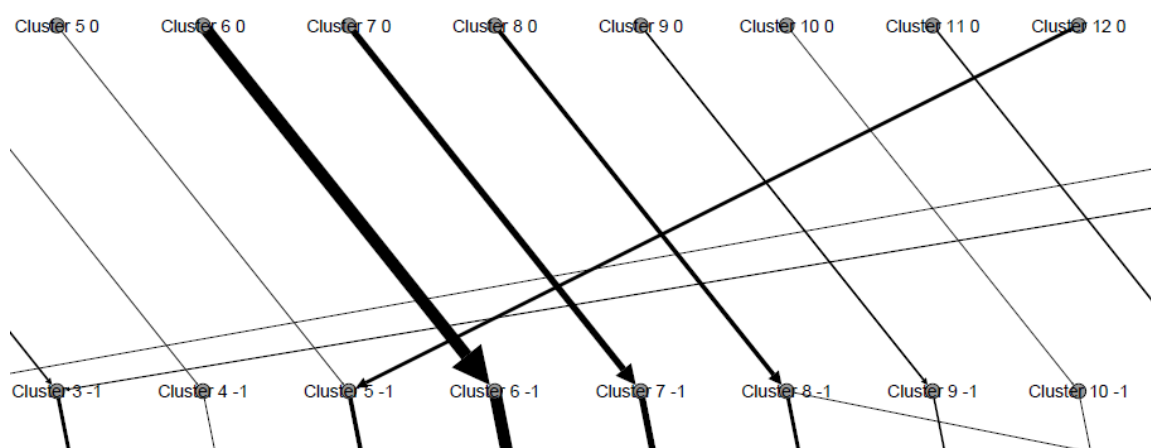
Figure 4.3.6: Illustration of Cluster 5 0 and Cluster 12 0 merging to form Cluster 5 -1. This results in a two-way split in Tree 5.
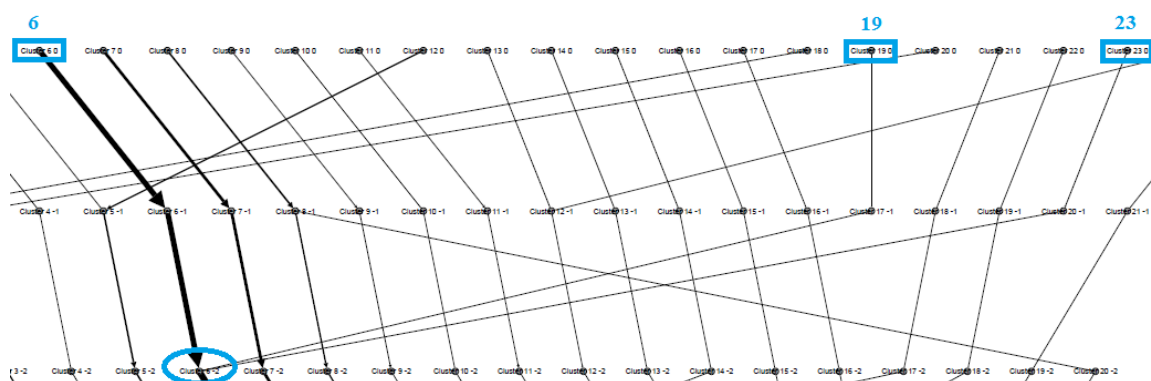
Figure 4.3.7: Clusters 6, 19, and 23 all merge together at Cluster 6 -2 (circled) in the tier chart. This results in a three-part polytomy in Tree 5.

The protein lists in **Tables 4.3.1**, **4.3.3**, **4.3.5**, and **4.3.7** associated with each of the figures below show the proteins contained within each cluster of the initial, least stringent iteration. The individual proteins have been color-coded to display their function (as indicated by their name), and each cluster also has its domain architecture listed as it appears in the initial iteration. Protein indices correspond to that used internally by the bit vector algorithm, and are based on their order of occurrence in the HMMER data file. A key is provided to give the specific functions. The clusters have been arranged according to their placement in the various trees (**Figures 4.3.8-4.3.15**), and are listed by branch order from top to bottom. The cluster labels within each of the figures have also been outlined with the color of the dominant function of the proteins contained within.

In addition to the versions of the trees without node labels or branch lengths, there are also versions of each of those trees that do use branch lengths and node labels (**Figures 4.3.9**, **4.3.11**, **4.3.13**, and **4.3.15**). The labels use the internal indices for each of the domains. The associated tables (**Tables 4.3.2**, **4.3.4**, **4.3.6**, and **4.3.8**) provide keys to those domains. The branch lengths are calculated based on the number of iterations a branch survives before being merged with another branch. Each iteration adds 0.1 to the branch length. For example, **Figure 4.3.9** exhibits a difference in the length between the branch containing clusters 14 and 16 versus the branch containing cluster 3 and 20. Clusters 3 and 20 merge in the second iteration, while clusters 14 and 16 merge in the eighth iteration. As a result, clusters 14 and 16 both have longer branches before joining to form a node.

The protein functions generally match with the clusters grouped in each tree, with most or all of the proteins within a given tree sharing the same or related functions. For example, **Figure 4.3.8** and **Table 4.3.1** indicate that Tree 1 is composed exclusively of guanine nucleotide exchange factors. **Figure 4.3.10** and **Table 4.3.3** shows that Tree 2 is composed entirely of kinases. **Figure 4.3.12** and **Table 4.3.5** presents Tree 3 as being composed only of sorting nexins. **Figure 4.3.14** and **Table 4.3.7** illustrates that Tree 5 is composed of all the RGS proteins, plus a few outliers, either kinases or related proteins. **Table 4.3.9** shows similar consistency with the membership of the singleton clusters. Clusters 9 and 11 are composed of nucleotide exchange factors and axins, respectively. Cluster 7 appears to be composed of a group of poorly-understood proteins, including some that are either predicted, or have not been characterized.

The results of the homogeneity and discontinuity calculations are provided in **Tables 4.3.10** and **4.3.11**.

It should be noted that the RGS family is characterized by proteins possessing either the RGS or RGS-like domain. Cluster 9 is the only exception, but it should be noted that the original HMMER data file actually does include the RGS domain for each of its member proteins, but as the i-Evalue was given as greater than 1.0, this was ignored. This also seems to be one of the reasons Cluster 9 was not included within Tree 1 with the other guanine nucleotide exchange factor proteins. Overall, the domain architecture between the Tree 1 clusters and Cluster 9 seems to be too divergent for them to have been placed together by this method. It should also be pointed out that the Cluster 9 members are all labelled as DBS proteins, while none of the Tree 1 members are. This may be another reason for this distinction in the clustering.

Cluster 8 is composed entirely of kinase proteins. NP_036011.3, a rhodopsin kinase precursor at first glance appears to be an outlier among the G protein-coupled receptor kinases. However, it should be noted that NCBI lists EDL22144.1 as an identical protein to NP_036011.3. EDL22144.1 is described as G protein-coupled receptor kinase 1 from *Mus musculus*, and so NP_036011.3 appears to not actually be an outlier in this case. The sister branch to Cluster 8 in Tree 2 is composed of clusters 4 and 21. Each contains a single beta-adrenergic receptor kinase. As such, Tree 2 is broadly composed of two classes of kinases, and each class is contained within its own branch of the tree.

Tree 5 is composed of an almost uniform selection of RGS proteins. However, Cluster 6 contains three outliers: a G-protein-coupled receptor kinase, a precursor to an A-kinase anchor protein, and a beta-adrenergic receptor kinase. Cluster 6's domain architecture consists solely of the RGS domain itself. Indeed, the entries of these three proteins in the HMMER data file only list the RGS domain as being matched to them. This is unexpected, as compared to the architectures seen in clusters 4, 8, 17, and 21. This may be an indication that the domain architecture does not necessarily dictate a protein's function in all cases, or, more likely, possibly that the data provided by HMMER did not accurately reflect the total architecture of these three proteins. There could have been prediction errors possibly brought about by removed exons. This could have removed some of the domains as well. Because both of the kinases are isoforms, or alternate versions of the same protein produced by the same gene, this is a strong possibility. Also, NP_064305.2 (the A-kinase anchor protein) is reported by NCBI to have a total of three domains, including a binding domain of A-kinase anchor proteins.

NP_001030608.1 (the beta-andrenergic receptor kinase) is listed by NCBI as an obsolete version of NM_001035531.1, which has been removed due to insufficient support.

NP_001074212.1 (the G-protein-coupled receptor kinase) is indeed listed by NCBI as possessing only the RGS superfamily domain, in contrast to NP_062370.2 (one of the proteins from Cluster 8), which has other domains as well.

Key:

regulator of G-protein signaling   sorting nexin

RNA-binding protein   axin

G-protein-coupled receptor kinase   beta-adrenergic receptor kinase

guanine nucleotide exchange factor   PM1-like

A-kinase anchor protein   Slx-like

rhodopsin kinase precursor   predicted genes / proteins
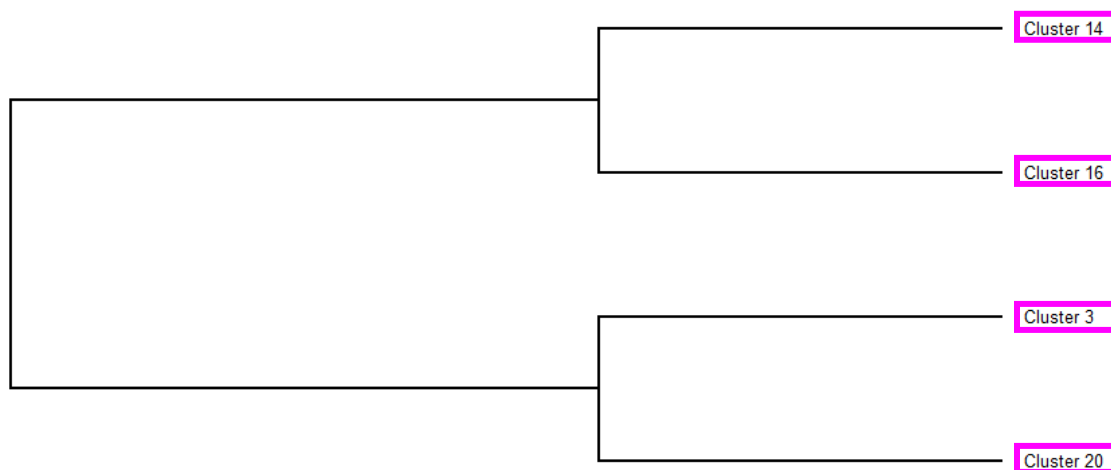
Figure 4.3.8: The version of Tree 1 from the RGS data without node labels or branch lengths.

Table 4.3.1: The clusters present in Tree 1. Domain architecture is given on the right of each cluster header.

Cluster 14	RGS-like, RGS, RhoGEF, PDZ, PDZ_2, DUF3135
22: NP_001003912.1 - rho guanine nucleotide exchange factor 11	
Cluster 16	RGS-like, RhoGEF, PDZ, PDZ_2, OmpH, AAA_23
31: NP_081420.2 - rho guanine nucleotide exchange factor 12	
Cluster 3	RGS-like, RhoGEF, PH_5
3: NP_001123624.1 - rho guanine nucleotide exchange factor 1 isoform c	
24: NP_001123625.1 - rho guanine nucleotide exchange factor 1 isoform c	
46: NP_032514.1 - rho guanine nucleotide exchange factor 1 isoform d	
Cluster 20	RGS-like, RhoGEF
42: NP_001123623.1 - rho guanine nucleotide exchange factor 1 isoform b	
45: NP_001123622.1 - rho guanine nucleotide exchange factor 1 isoform a	
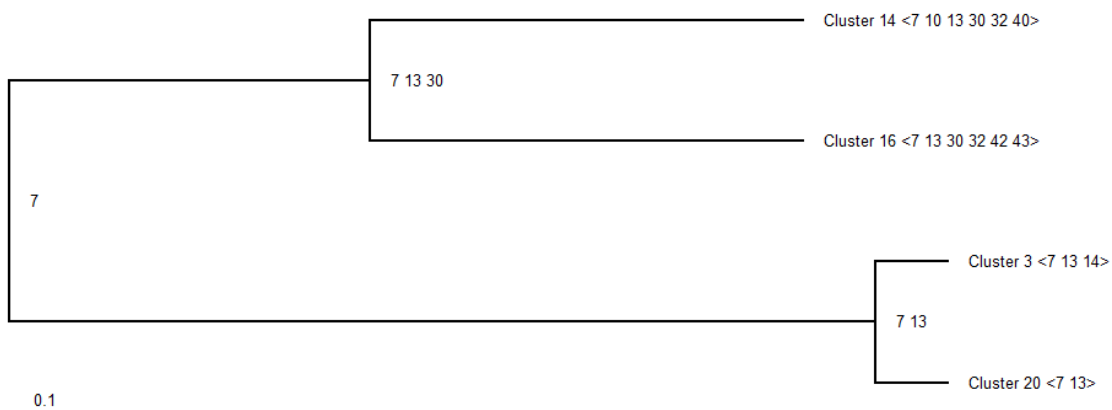
Figure 4.3.9: Tree 1 with branch lengths, node labels, and domain architecture displayed.

Table 4.3.2: Key to the domain indices present in **Figure 4.3.9**.

Index Number	Domain
7	RGS-like
10	RGS
13	RhoGEF
14	PH_5
30	PDZ
32	PDZ_2
40	DUF3135
42	OmpH
43	AAA_23
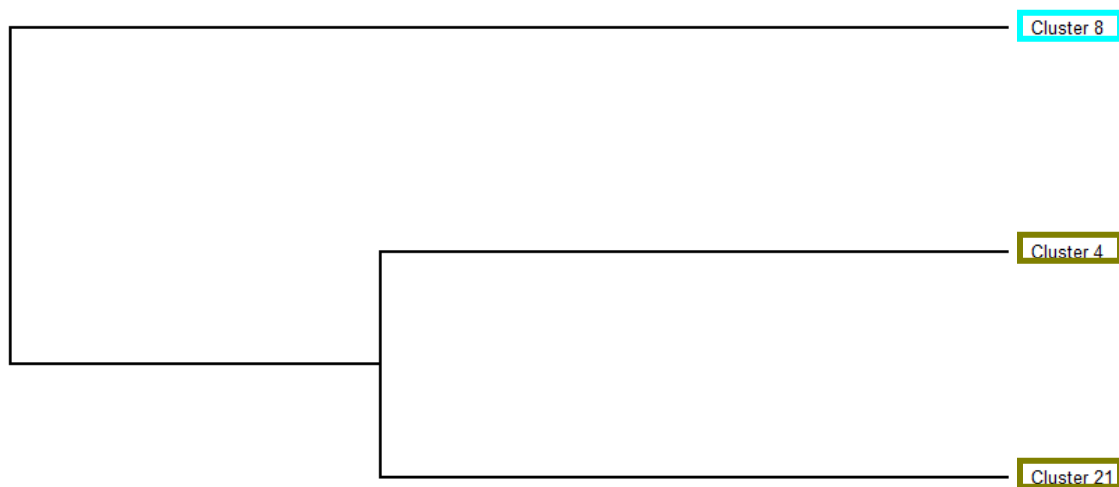
Figure 4.3.10: The version of Tree 2 from the RGS data without node labels or branch lengths.

Table 4.3.3: The clusters present in Tree 2. Domain architecture is given on the right of each cluster header.

Cluster 8	RGS, Pkinase, Pkinase_Tyr, Kinase-like
13: NP_062370.2 - G protein-coupled receptor kinase 4 isoform 1	
20: NP_036011.3 - rhodopsin kinase precursor	
26: NP_001033107.1 - G protein-coupled receptor kinase 6 isoform a	
38: NP_001106182.1 - G protein-coupled receptor kinase 6 isoform c	
62: NP_061357.3 - G protein-coupled receptor kinase 5	
66: NP_036068.2 - G protein-coupled receptor kinase 6 isoform b	
Cluster 4	RGS, Pkinase, Pkinase_Tyr, PH, PH_11
4: NP_796052.2 - beta-adrenergic receptor kinase 2 isoform 1	
Cluster 21	RGS, Pkinase, Pkinase_Tyr, PH, PH_11, Kinase-like, Kdo
49: NP_570933.1 - beta-adrenergic receptor kinase 1 isoform 2	
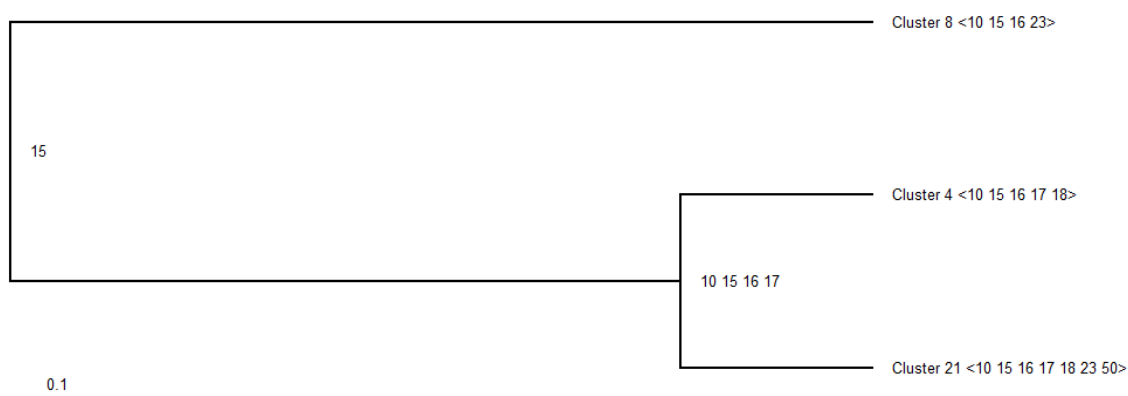
Figure 4.3.11: Tree 2 with branch lengths, node labels, and domain architecture displayed.

Table 4.3.4: Key to the domain indices present in **Figure 4.3.11**.

Index Number	Domain
10	RGS
15	Pkinase
16	Pkinase_Tyr
17	PH
18	PH_11
23	Kinase-like
50	Kdo
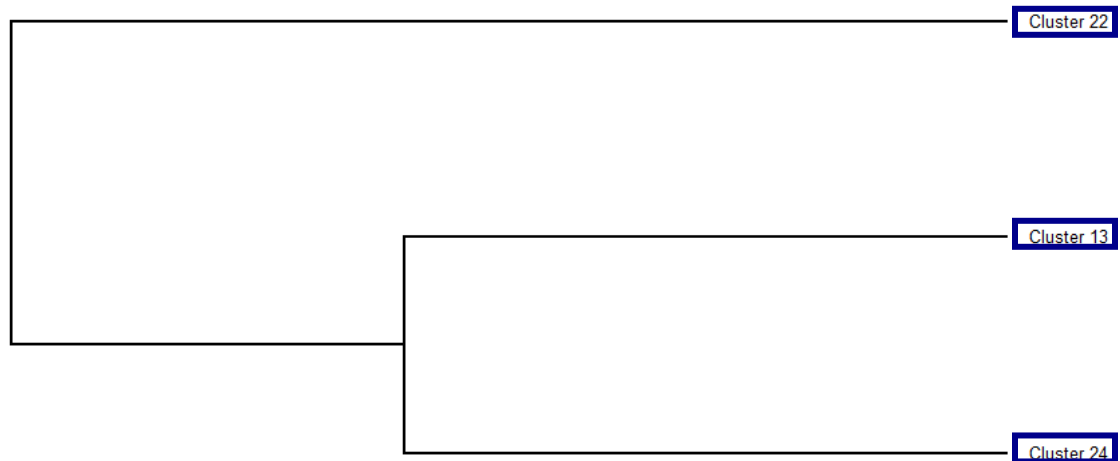
Figure 4.3.12: The version of Tree 3 from the RGS data without node labels or branch lengths.

Table 4.3.5: The clusters present in Tree 3. Domain architecture is given on the right of each cluster header.

Cluster 22	RGS, PXA, Nexin_C, PX, End3, Isy1
51: NP_997096.2 - sorting nexin-25	
Cluster 13	RGS, PXA, Nexin_C, PX, COX5A
19: NP_001014973.2 - sorting nexin-13	
Cluster 24	RGS, PXA, Nexin_C, PX
59: NP_766514.2 - sorting nexin-14	
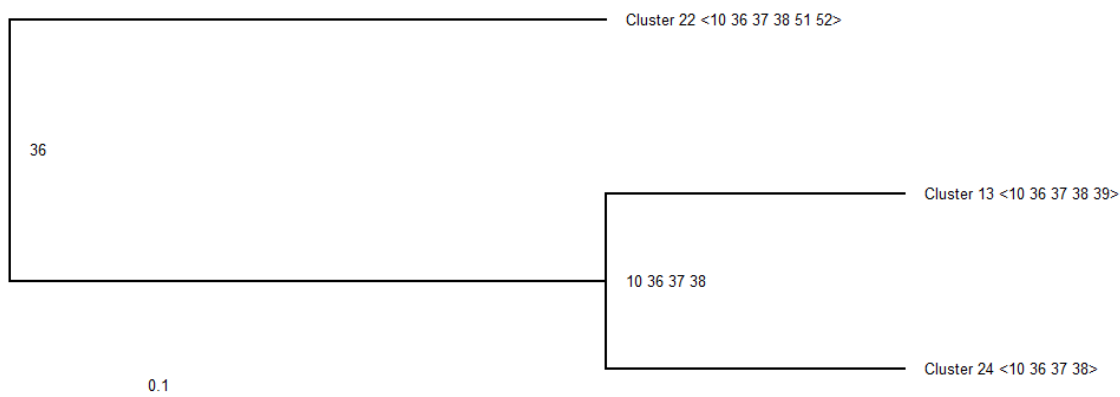
Figure 4.3.13: Tree 3 with branch lengths, node labels, and domain architecture displayed.

Table 4.3.6: Key to the domain indices present in **Figure 4.3.13**.

| Index Number	Domain
10	RGS
36	PXA
37	Nexin_C
38	PX
39	COX5A
51	End3
52	Isy1
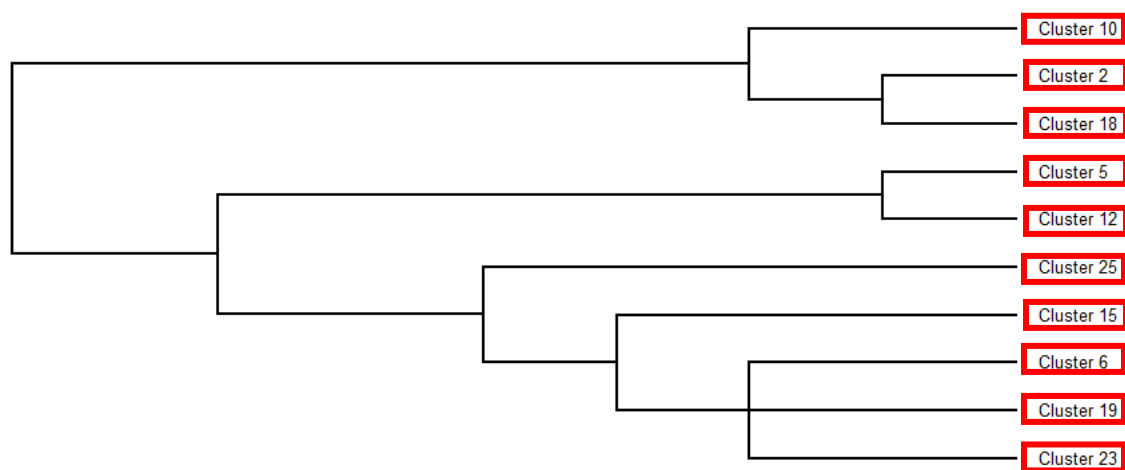
Figure 4.3.14: The version of Tree 5 from the RGS data without node labels or branch lengths.

Table 4.3.7: The clusters present in Tree 5. Domain architecture is given on the right of each cluster header.

Cluster 10	RBD, RGS, GoLoco, PDZ, PID, PDZ_2, TUG-UBL1
15: NP_775578.2 - regulator of G-protein signaling 12 isoform A	
Cluster 2	RBD, RGS, GoLoco, PSD4
2: NP_058038.2 - regulator of G-protein signaling 14	
Cluster 18	RBD, RGS, GoLoco, TUG-UBL1
36: NP_001156984.1 - regulator of G-protein signaling 12 isoform B	
Cluster 5	RGS, DEP, G-gamma, DUF1203
5: NP_056627.1 - regulator of G-protein signaling 6	
Cluster 12	RGS, DEP, G-gamma
18: NP_001074538.1 - regulator of G-protein signaling 11	
30: NP_001185932.1 - regulator of G-protein signaling 7 isoform 2	
41: NP_001159406.1 - regulator of G-protein signaling 9 isoform 2	
58: NP_035398.2 - regulator of G-protein signaling 9 isoform 1	
64: NP_036010.2 - regulator of G-protein signaling 7 isoform 1	
Cluster 25	RGS, PDZ, PDZ_2, TFIIA
60: NP_599018.3 - regulator of G-protein signaling 3 isoform 2	
Cluster 15	RGS-like, RGS, FliL
25: NP_080694.1 - regulator of G-protein signaling 10	
Cluster 6	RGS
6: XP_921002.3 - PREDICTED: regulator of G-protein signaling 21	
9: NP_080722.1 - regulator of G-protein signaling 19	
10: NP_694811.1 - regulator of G-protein signaling 13	
11: NP_056626.2 - regulator of G-protein signaling 1	
12: NP_001182677.1 - regulator of G-protein signaling 22	
17: NP_033089.2 - regulator of G-protein signaling 5	
23: NP_001171266.1 - regulator of G-protein signaling 20 isoform 1	
28: NP_001074212.1 - G protein-coupled receptor kinase 4 isoform 2	
33: NP_033088.2 - regulator of G-protein signaling 4	
39: NP_075019.1 - regulator of G-protein signaling 18	
40: NP_067349.2 - regulator of G-protein signaling 20 isoform 2	
43: XP_894544.3 - PREDICTED: regulator of G-protein signaling 21	
47: NP_064305.2 - A-kinase anchor protein 10, mitochondrial precursor	
50: NP_035397.2 - regulator of G-protein signaling 16	
55: NP_033087.2 - regulator of G-protein signaling 2	
57: NP_001030608.1 - beta-adrenergic receptor kinase 2 isoform 2	
61: NP_080656.2 - regulator of G-protein signaling 8	
Cluster 19	RGS, Spexin
37: NP_001230152.1 - regulator of G-protein signaling protein-like	
Cluster 23	RGS, DUF4226
53: NP_001155294.1 - regulator of G-protein signaling 17 isoform 1	
63: NP_064342.1 - regulator of G-protein signaling 17 isoform 2	
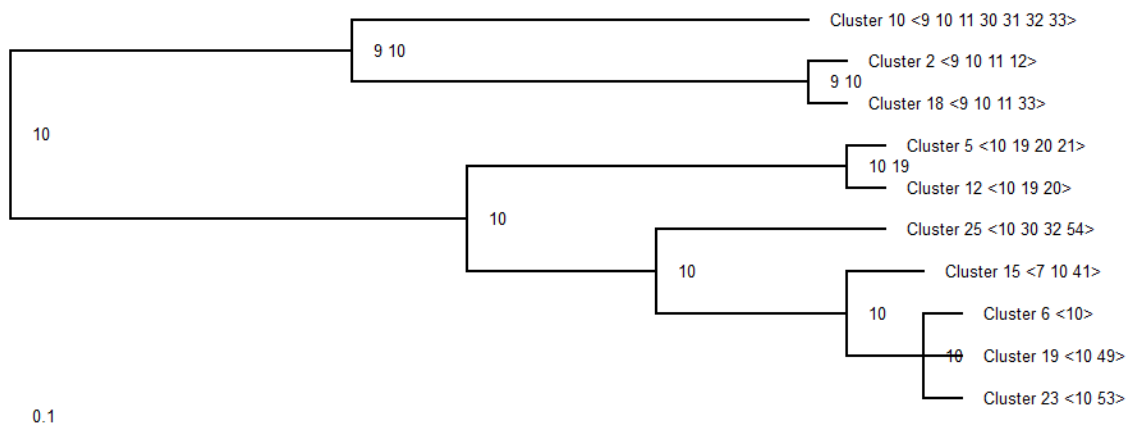
Figure 4.3.15: Tree 5 with branch lengths, node labels, and domain architecture displayed.

Table 4.3.8: Key to the domain indices present in **Figure 4.3.15**.

Index Number	Domain
6	zf-CCCH
7	RGS-like
9	RBD
10	RGS
11	GoLoco
12	PSD4
19	DEP
20	G-gamma
21	DUF1203
30	PDZ
31	PID
32	PDZ_2
33	TUG-UBL1
41	FliL
49	Spexin
53	DUF4226
54	TFIIA

Table 4.3.9: A listing of the singleton tree clusters from the RGS data. Domain architecture is given on the right of each cluster header.

Cluster 17	RGS-like, AKAP7_NLS, AKAP7_RIRII_bdg, Corona_NS2A, 2_5_RNA_ligase2, tRNA_lig_CPD
34: NP_061217.3 - A-kinase anchor protein 7	
Cluster 9	RhoGEF, PH, CRAL_TRIO_2, Spectrin, SH3_1, SH3_9, MCPsignal, SH3_2
14: NP_835177.2 - guanine nucleotide exchange factor DBS isoform 1	
29: NP_001152957.1 - guanine nucleotide exchange factor DBS isoform 3	
52: NP_001152958.1 - guanine nucleotide exchange factor DBS isoform 2	
Cluster 11	RGS, DIX, Axin_b-cat_bind
16: NP_056547.3 - axin-2	
27: NP_001153070.1 - axin-1 isoform 1	
54: NP_033863.2 - axin-1 isoform 2	
Cluster 1	RRM_5, RRM_6, RRM_1, Nup35_RRM_2, PWI, zf-CCCH, RGS-like, DUF2785
1: NP_598838.3 - RNA-binding protein 26	
Cluster 7	RGS, Cor1
7: NP_001160118.1 - uncharacterized protein LOC100040867	
8: XP_986693.2 - PREDICTED: X-linked lymphocyte-regulated protein PM1-like	
21: XP_003945709.1 - PREDICTED: X-linked lymphocyte-regulated protein PM1-like	
32: NP_001160073.1 - predicted gene 16430	
35: XP_003945710.1 - PREDICTED: X-linked lymphocyte-regulated protein PM1-like	
44: XP_987134.2 - PREDICTED: X-linked lymphocyte-regulated protein PM1-like	
48: NP_001207426.1 - Slx-like	
56: NP_001207427.1 - Slx-like	
65: XP_001474919.1 - PREDICTED: X-linked lymphocyte-regulated protein PM1-like	

Table 4.3.10: Percentage homogeneity of each of the trees from **Figures 4.3.8-4.3.15**, as well as **Table 4.3.9**.

Tree Number	Homogeneity
Tree 1	100%
Tree 2	100%
Tree 3	100%
Cluster 17	100%
Tree 5	90.32%
Cluster 9	100%
Cluster 11	100%
Cluster 1	100%
Cluster 7	55.56%

Table 4.3.11: Percentage discontinuity of each of the functional types in the RGS data.

Function	Discontinuity
rho guanine nucleotide exchange factor	0%
guanine nucleotide exchange factor DBS	0%
G-protein-coupled receptor kinase rhodopsin kinase precursor beta-adrenergic receptor kinase	20%
sorting nexin	0%
regulator of G-protein signaling	0%
A-kinase anchor protein	50%
axin	0%
RNA-binding protein	0%
predicted genes / proteins	0%
PM1-like	0%
Slx-like	0%

The dataset of 66 RGS proteins was also tested using the maximum likelihood phylogenetic method. The full protein sequences were not used for this, however. The RGS or RGS-like domains for each protein were extracted based on whatever reported subsequence had the strongest i-Evalue for its protein. Two runs were conducted. MEGA was used to perform all alignment and tree building steps for the first run. Alignment was made using MUSCLE, and the maximum likelihood was conducted using the Jones-Taylor-Thornton (JTT) model. The settings are provided in **Tables 4.3.12** and **4.3.13**. The second run conducted the alignment step using MAFFT instead of MUSCLE. MEGA was still used to conduct the maximum likelihood analysis, and used the same settings as the first run. MAFFT was run using the L-INS-i option with default settings.

Table 4.3.12: Settings used for MUSCLE alignment of the RGS proteins.

Gap Open Penalty	-5
Gap Extend Penalty	-0.01
Hydrophobicity Multiplier	1.2
Maximum Memory	4095 MB
Maximum Iterations	8
Clustering Method (all iterations)	UPGMA
Lambda	25

Table 4.3.13: Settings used for maximum likelihood testing of the RGS proteins.

Bootstrap Replications	1000
Gamma Categories	5
Treatment of Gaps	Partial Deletion
Site Coverage Cutoff	50%
ML Heuristic Method	Subtree-Pruning-Regrafting level 5
Initial Tree	NJ/BioNJ
Branch Swap Filter	Very Strong
Number of Threads	6

The MUSCLE alignment's running time was insignificant, finishing within seconds. However, the maximum likelihood run took over 24 hours to finish, which is considerably slower than the bit vector clustering method on the same dataset. **Table 4.3.14** provides the details of this efficiency comparison. Also, bear in mind that the bit vector program assumes the presence of a pre-generated HMMER dataset file as input. The program's tasks not only include the clustering and tree building, but also reading and reformatting of the HMMER input files, iterating over the E-value thresholds (this will vary based on the total number of unique orders of magnitude less than 1.0 in the HMMER file's E-values), generating the tier chart, retrieving the unique Newick trees from the data, and outputting all data to files.

Table 4.3.15: Comparison between the running times of the bit vector method and the maximum likelihood run with MUSCLE alignment.

Dataset Information	
No. Proteins	66
No. Domains	54
No. E-value Iterations	52
Running Time	
Bit Vector	**Maximum Likelihood**
~23 secs.	27 hrs. 17 mins.

Figures 4.3.16a and **4.3.16b** display the tree generated by the maximum likelihood analysis using the MUSCLE alignment. Branches have been color-coded to match the coding used in **Figures 4.3.8**, **4.3.10**, **4.3.12**, and **4.3.14**, and **Tables 4.3.1**, **4.3.3**, **4.3.5**, **4.3.7**, **4.3.9** and **4.3.11**, and some branches have been labelled, where the relationships correspond well with the output of the bit vector approach. Aside from the outliers mentioned in Tree 5, both methods agree with the placement of the Tree 2 proteins together. Similarly, the maximum likelihood tree also places the Tree 1 and Cluster 9 proteins in separate branches. The bit vector approach also recognized this division. Clusters 7 and 11 are also preserved in the maximum likelihood tree, although Tree 3 and some of the single-cluster trees were not maintained, instead splitting their members across different branches. Aside from a pair of outliers, the maximum likelihood tree places the red proteins from Tree 5 all in the same branch. Some of its subdivisions more or less correspond with some of the clusters found by the bit vector approach, as well as the relationships between them. For example, clusters 2, 10, and 18 are placed together within a sub-branch, as are clusters 5 and 12. Cluster 6 is largely preserved, but is interspersed with some proteins from other clusters.
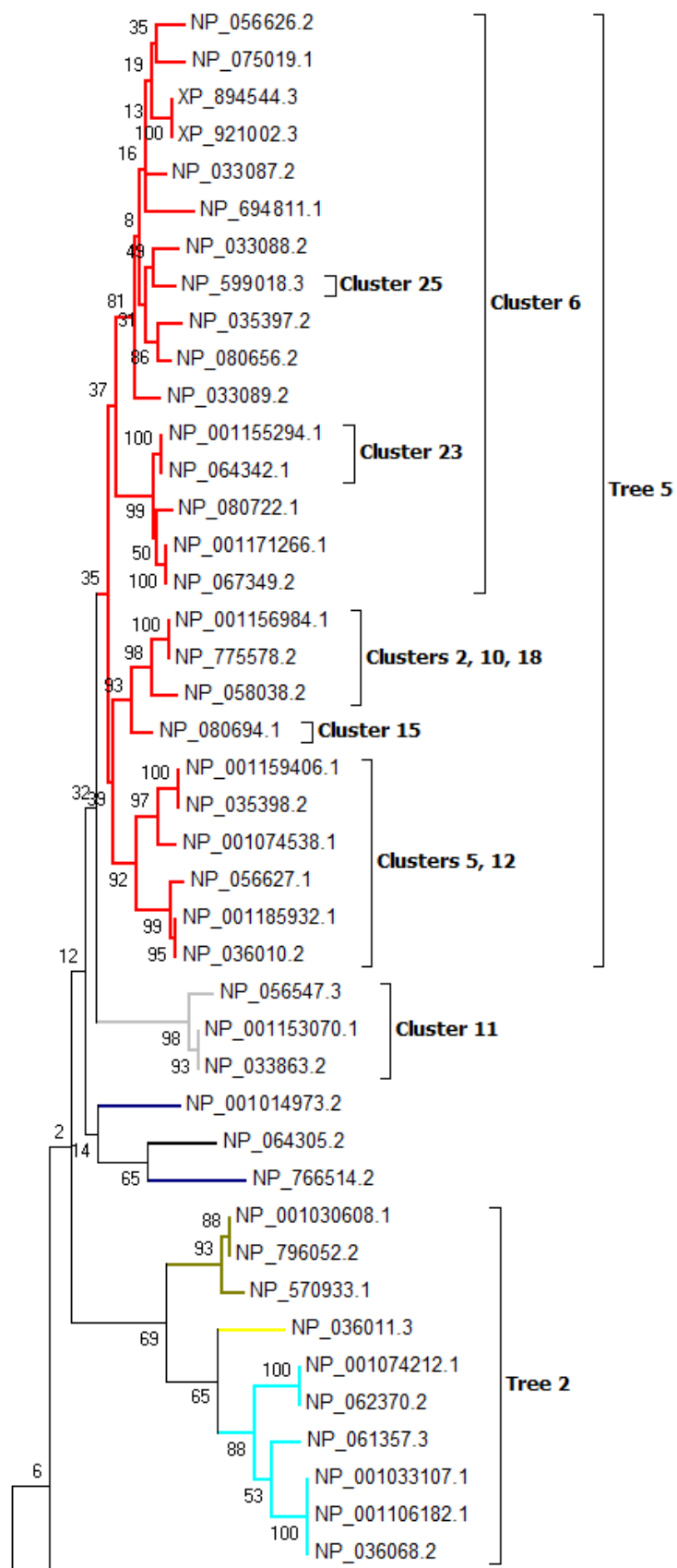
Figure 4.3.16a: First portion of the maximum likelihood tree for the RGS data using a MUSCLE alignment. Branch colors correspond to function, as seen in the previous tree visualizations.

Figure 4.3.16b: Second portion of the maximum likelihood tree for the RGS data, overlapping partly with **Figure 4.3.16a**. Branch colors correspond to function, as seen in the previous tree visualizations.

Figures 4.3.17a and **4.3.17b** present the maximum likelihood results for the

second run, using the MAFFT alignment. The tree has branches color coded in the same

way as the first run, and notable features are also labelled as before. Some notable

similarities with the tree in **Figures 4.3.16a** and **4.3.16b** are the placement of

NP_001182677.1 and NP_001230152.1 within a branch separate from the rest of the Tree

5 proteins, agreement on the placement of some of the Tree 5 proteins together in

branches matching their cluster groupings from the bit vector results (specifically,

clusters 2, 10, and 18, clusters 5 and 12, and Cluster 23), and the preservation of trees 1

and 2 and clusters 9 and 11. Some notable differences between the trees include the

placement of the Cluster 1 protein within the branch containing the Tree 1 proteins, and

the embedding of Cluster 11 within the branch containing the Tree 5 proteins.

Table 4.3.16: Comparison between the running times of the bit vector method and the maximum likelihood run with MAFFT alignment.

Dataset Information	
No. Proteins	66
No. Domains	54
No. E-value Iterations	52
Running Time	
Bit Vector	**Maximum Likelihood**
~23 secs.	24 hrs. 41 mins.
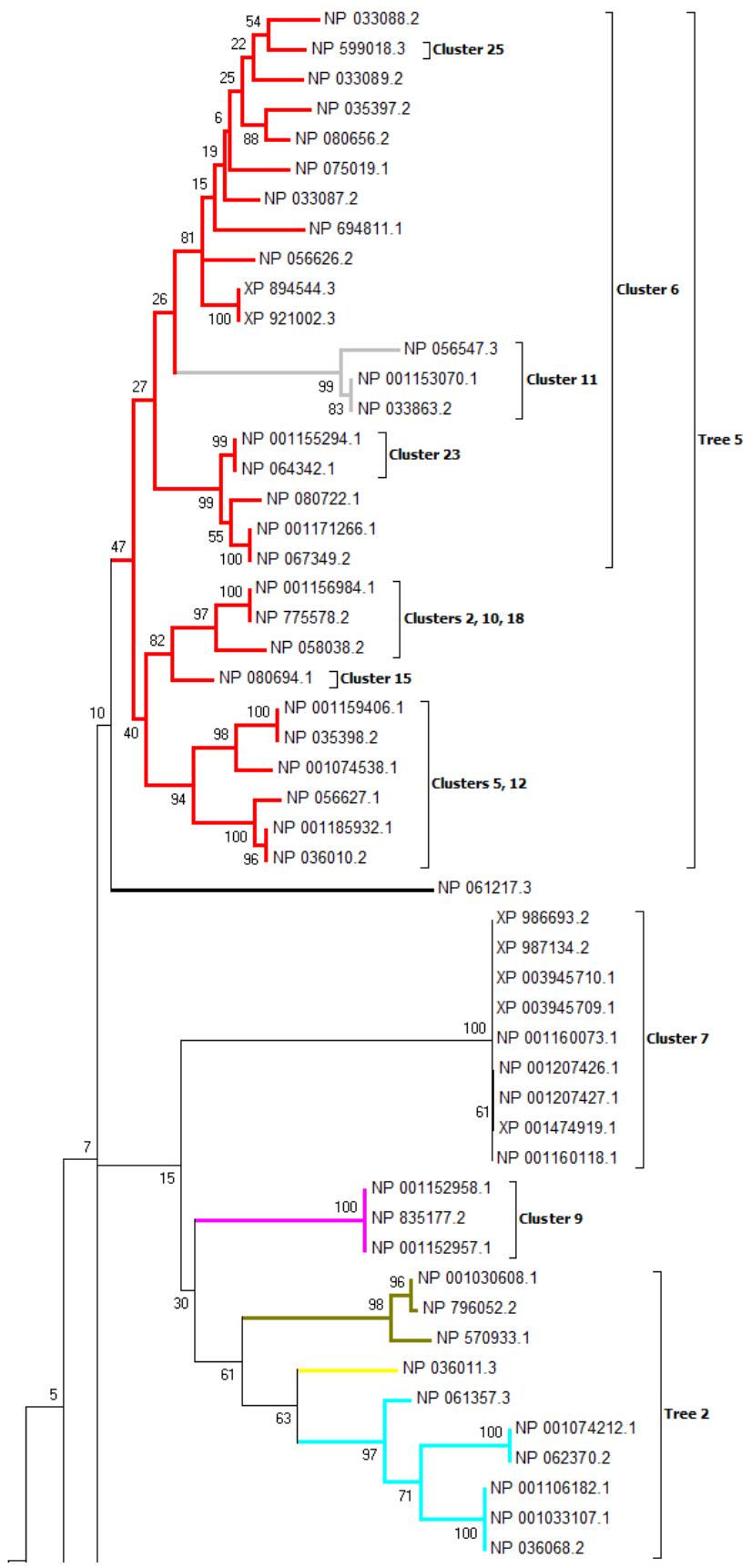
Figure 4.3.17a: First portion of the maximum likelihood tree for the RGS data using a MAFFT alignment. Branch colors correspond to function, as seen in the previous tree visualizations.
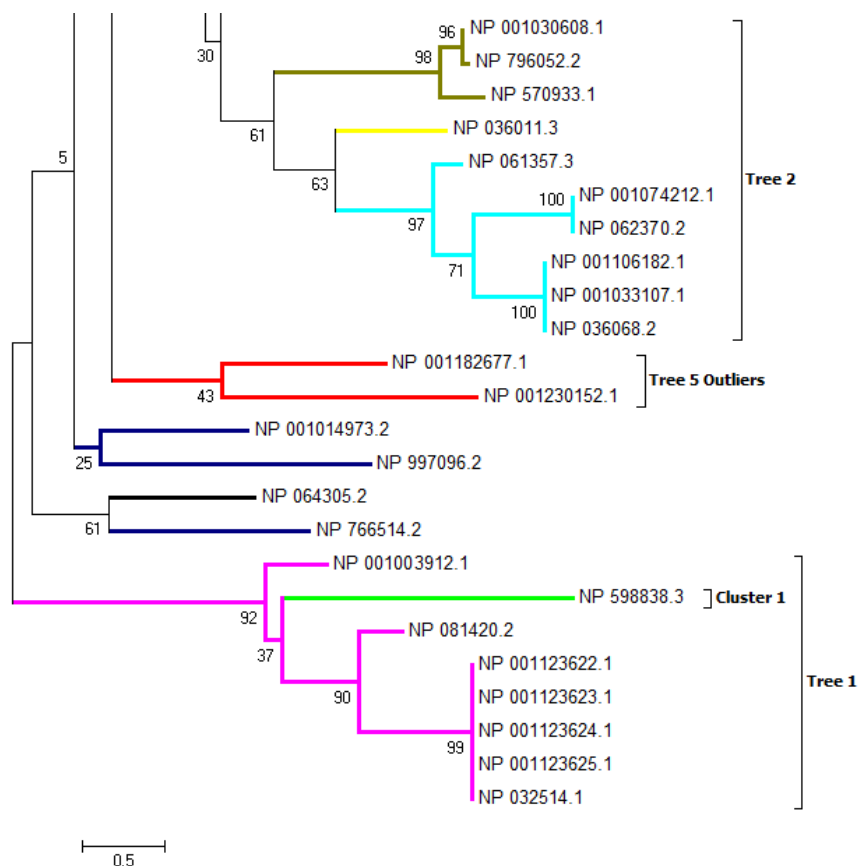
Figure 4.3.17b: Second portion of the maximum likelihood tree for the RGS data, overlapping partly with **Figure 4.3.17a**. Branch colors correspond to function, as seen in the previous tree visualizations.

The Jaccard indices are provided in **Table 4.3.17**. The indices for each of the bit vector trees is given, along with the average for the comparison (either bit vector to MUSCLE-aligned ML or bit vector to MAFFT-aligned ML). The ML clusters are also given in **Tables 4.3.18** and **4.3.19**.

Table 4.3.17: Jaccard indices of the bit vector trees as compared to the ML clusters.

Comparison to MUSCLE-aligned ML	
Tree 1	1
Tree 2	0.56
Tree 3	0.33
Tree 5	0.35
Cluster 1	1
Cluster 7	1
Cluster 9	1
Cluster 11	1
Cluster 17	1
Average	0.8
Comparison to MAFFT-aligned ML	
Tree 1	0.875
Tree 2	0.56
Tree 3	0.33
Tree 5	0.35
Cluster 1	0.125
Cluster 7	1
Cluster 9	1
Cluster 11	1
Cluster 17	1
Average	0.69

Table 4.3.18: Cluster membership for the MUSCLE-aligned ML tree. Proteins not mentioned in this list form singleton clusters.

Cluster	Members
Cluster 1	NP_056626.2, NP_075019.1, XP_894544.3, XP_921002.3, NP_033087.2, NP_694811.1, NP_033088.2, NP_599018.3, NP_035397.2, NP_080656.2, NP_033089.2
Cluster 2	NP_001155294.1, NP_064342.1, NP_080722.1, NP_001171266.1, NP_067349.2
Cluster 3	NP_001156984.1, NP_775578.2, NP_058038.2, NP_080694.1
Cluster 4	NP_001159406.1, NP_035398.2, NP_001074538.1, NP_056627.1, NP_001185932.1, NP_036010.2
Cluster 5	NP_056547.3, NP_001153070.1, NP_033863.2
Cluster 6	NP_001030608.1, NP_796052.2, NP_570933.1
Cluster 7	NP_001074212.1, NP_062370.2, NP_061357.3, NP_001033107.1, NP_001106182.1, NP_036068.2
Cluster 8	NP_001152957.1, NP_001152958.1, NP_835177.2
Cluster 9	NP_001207426.1, XP_001474919.1, NP_001207427.1, XP_987134.2, XP_003945710.1, XP_003945709.1, NP_001160118.1, NP_001160073.1, XP_986693.2
Cluster 10	NP_001003912.1, NP_081420.2, NP_032514.1, NP_001123623.1, NP_001123625.1, NP_001123622.1, NP_001123624.1

Table 4.3.19: Cluster membership for the MAFFT-aligned ML tree. Proteins
not mentioned in this list form singleton clusters.

Cluster	Members
Cluster 1	NP_033088.2, NP_599018.3, NP_033089.2, NP_035397.2, NP_080656.2, NP_075019.1, NP_033087.2, NP_694811.1, NP_056626.2, XP_894544.3, XP_921002.3
Cluster 2	NP_056547.3, NP_001153070.1, NP_033863.2
Cluster 3	NP_001155294.1, NP_064342.1, NP_080722.1, NP_001171266.1, NP_067349.2
Cluster 4	NP_001156984.1, NP_775578.2, NP_058038.2, NP_080694.1
Cluster 5	NP_001159406.1, NP_035398.2, NP_001074538.1, NP_056627.1, NP_001185932.1, NP_036010.2
Cluster 6	XP_986693.2, XP_987134.2, XP_003945710.1, XP_003945709.1, NP_001160073.1, NP_001207426.1, NP_001207427.1, XP_001474919.1, NP_001160118.1
Cluster 7	NP_001152958.1, NP_835177.2, NP_001152957.1
Cluster 8	NP_001030608.1, NP_796052.2, NP_570933.1
Cluster 9	NP_061357.3, NP_001074212.1, NP_062370.2, NP_001106182.1, NP_001033107.1, NP_036068.2
Cluster 10	NP_001003912.1, NP_598838.3, NP_081420.2, NP_001123622.1, NP_001123623.1, NP_001123624.1, NP_001123625.1, NP_032514.1

4.4 Discussion of RGS Data

The bit vector program was shown overall to accurately cluster the RGS proteins

into groups based on functionality. Although there are some outliers, such as the divide

between the Tree 1 proteins and Cluster 9, or the kinases in Cluster 6 of Tree 5, these still

have come about solely as a result of the domain data made available to the program.

The Tree 1-Cluster 9 division is due to a very weak matching of the RGS domain

to the Cluster 9 proteins. It is present in the HMMER data, but for each protein it was

over the 1.0 E-value threshold, and so was ignored. Overall, the domain architectures in

these clusters are too dissimilar to be included within a single tree. The RhoGEF

(guanine nucleotide exchange factor) domain is the only domain in common, with the

Cluster 9 proteins also possessing several other domains that do not appear in the Tree 1

proteins. Ultimately, what prevents these trees from merging is the complete absence of the RGS-like domain in Cluster 9, while this domain in Tree 1 is easily the strongest match out of the entire sampled RGS dataset, surviving to an order of magnitude of 10^{-69} before finally being deleted. As the RGS-like domain was the strongest match within Tree 1, and one of the most important defining features of the proteins of that group, its absence from Cluster 9 meant that there could be no merging of the two groups. As such, they remained separate throughout the entire set of iterations. Also bear in mind the presence of the DBS label in the Cluster 9 proteins, and its absence in any of the Tree 1 clusters, as stated in **Section 4.3**. This indicates that the two groups actually do not have identical function, even though they are functionally associated with each other.

The presence of outliers in Cluster 6 is, again, a product of how the bit vector program operates. As each of the Cluster 6 proteins only possess the RGS domain according to the HMMER data that was used, they are clustered together regardless of what their stated function may be. That said, it may be the case that the quality of information available on these proteins at the time the HMMER data was generated may not have reflected their full domain architecture. As stated in **Section 4.3**, at least two of the outlier proteins are assigned a different set of domains according to information available from NCBI. It may be that the HMMER results were not fully accurate for them. Alternatively, it could indicate that domain architecture alone does not fully specify a protein's function, but is also defined by other factors affecting its three-dimensional structure. As it is, this approach only uses the domain architecture to infer similar structure, and therefore function, between related proteins. This may indicate that this assumption is not universally accurate.

All performance comparisons were conducted using the same system (a desktop running the 64-bit version of Windows 7, with 10 GB of RAM and a 6-core 3.5 GHz processor). The running time of the bit vector method is a substantial improvement (<1 minute vs. over 24 hours) over the running time of using maximum likelihood, which is the standard approach for quality phylogenetic reconstruction. Although the trees generated by these two methods differ in some ways according to how the RGS proteins are grouped in the branches, they also share several key similarities. Many of the clusters, and even whole trees generated from the clustering method are reproduced mostly intact in the maximum likelihood results as well. However, also bear in mind that these approaches each attempt to analyze different things. The maximum likelihood test used only the extracted RGS or RGS-like subsequences for each protein, and the resulting tree is based purely on the alignment of the sequences. This means that leaf placement is based on how similar the sequences are to one another. Highly similar sequences can be expected to be found within the same branch, and very divergent sequences will likely be on entirely separate branches. In contrast, the bit vector approach compares overall domain architecture, without regard to sequence similarity in and of itself. The E-value threshold has the effect of indirectly dealing with similarity, because a strong match for a domain in one protein versus a weaker match for the same domain in another protein can be assumed to mean the actual domain sequences are less similar than if their E-values were closer in value. Such a mismatch in E-values will be reflected in how the proteins cluster, and how clusters split or merge over several iterations. Ultimately, however, proteins are clustered, and similar clusters are grouped in branches, based on the similarity of their domain architecture. Because architecture can be used to infer

function, the bit vector approach reflects divergence in proteins' function, while maximum likelihood reflects divergence in the proteins' sequences themselves.

It should also be discussed that both maximum likelihood tests not only used the implementation found in MEGA, which is not as efficient as other implementations available, but also used 1000 bootstrap replications. While the use of bootstrapping is needed for quality phylogenetic trees, for the purposes of comparing basic running times of the two methods, it is not a fair assessment to include the bootstrapping in the running times, as there is no equivalent of this feature present in the bit vector program. When bootstrapping is removed, the MEGA implementation of maximum likelihood finished in approximately 9 minutes using the MAFFT alignment as input.

RAxML [33], a more efficient implementation of maximum likelihood, is available through Trex-online [34]. When the MAFFT alignment was provided as input, using no bootstrapping, the PROTCAT substitution model, the JTT matrix, and otherwise default settings, the program finished in approximately 6 minutes (based on the start and finish times provided in the resulting email from the server). However, this is also the running time on the Trex server, rather than the local system that previous comparisons were conducted on. Even so, this is still less efficient than the bit vector program's 23 seconds on the RGS data.

Furthermore, a comparison with maximum likelihood may itself not be fair, due to fundamental differences in how the two approaches operate. Neighbor-joining [6] is more similar to the pairwise comparisons of the bit vector approach, and so should be a fairer comparison. Using the Jones-Taylor-Thornton model, an assumption of uniform rates among sites, and 50% partial deletion, with 1000 bootstraps, MEGA's

implementation of neighbor-joining finished in 36 minutes using the MAFFT alignment. However, when bootstrapping was eliminated, the procedure finished in approximately 4 seconds, showing an improvement on the bit vector approach's performance of 23 seconds on the same data. Some possible reasons for this relative inefficiency may be the bit vector program's use of some pre- and post-processing steps on the input and output data, and also some computational techniques that could be optimized in the future.

Neighbor-joining uses the basic approach of calculating pairwise distances for each of the taxa in the unresolved tree. The pair of taxa with the lowest distance measure are then placed together as sister groups within the same branch. Distance measurements are then taken of the taxa in this branch with each of the remaining taxa, and so on until the tree is resolved. This procedure is simplified from the minimum evolution method. The primary reason for the speed increase from minimum evolution is that with neighbor-joining, the distance measurements of only certain topologies of the tree are calculated, rather than performing the calculation for every possible topology of the tree [35].

The basic approach for maximum likelihood, on the other hand, is to construct an initial tree, and then optimize it by creating variations on the tree topology. Many different topologies are constructed, and the likelihood of each topology is calculated until a topology with the best fit to the data is found. Searching so many topologies is very time consuming, which accounts for the greater running time observed with this method.

As was stated in **Chapter 3**, for *m* proteins, *n* domains, and *e* iterations on the E-value threshold, the bit vector approach has a time complexity of $\Theta(em^2n)$ to cluster the proteins over each E-value iteration, and an additional $\Theta(em^2)$ to generate the final

Newick strings upon which the tree is based. This accounts for the longer running time as compared to the more straightforward pairwise distance measurements used in neighbor-joining.

The Jaccard indices show that the bit vector trees generated were most similar to the clusters found within the MUSCLE-aligned maximum likelihood tree, with an average of 0.8. The comparison to the MAFFT-aligned maximum likelihood tree gave an average of 0.69. This difference seems to be partly due to the placement of NP_598838.3 within the branch containing the Tree 1 proteins, rather than placing it in its own branch, as the MUSCLE-aligned tree did.

4.5 *B. subtilis* **Data**

The run using the *B. subtilis* HMMER data was intended mostly to assess the performance of the bit vector program on large datasets. It generated a total of 2,009 individual trees, of which 517 contain more than one cluster, and the remaining 1,492 are all singleton trees. **Table 4.5.1** provides the details of the dataset information and running time (based on the time stamp of the last file created). The tier chart for this dataset is unreadable due to the high number of edges and edge crossings. Due to the size of the dataset, no comparison using maximum likelihood or any other alignment-based method were possible. However, based on the comparison of the running time on the RGS data (see **Table 4.3.14**), a maximum likelihood run using a dataset of equal size to the *B. subtilis* data would be considerably greater than the running time of the bit vector approach, to the point that doing so would be totally impractical.

Table 4.5.1: Data set information and running time of the bit vector method.

Dataset Information	
No. Proteins	3,973
No. Domains	4,737
No. E-value Iterations	208
Running Time	
73 hrs. 55 mins.	

During the testing phase, small subsets of the *B. subtilis* data were extracted from the HMMER file in order to run the program in much faster time using smaller data input. The trees associated with this data are displayed in **Figures 4.5.1**-**4.5.4**. Tree 656 (**Figures 4.5.3** and **4.5.4**) is actually substantially larger than what is shown, to the point that TreeView cannot display the cluster labels legibly. The original test set for this tree was also only a subset of the clusters participating in this tree, and so only the pertinent branch has been displayed.

Tables 4.5.2 and **4.5.4** present the first iteration membership of the clusters seen in Tree 187 and the Tree 656 subset. The individual proteins have been color-coded to display their function (as indicated by their name given in the NCBI entry for each protein), and each cluster also has its domain architecture listed as it appears in the initial iteration. Protein indices correspond to that used internally by the bit vector algorithm, and are based on their order of occurrence in the HMMER data file. A key is provided to give the specific functions. The clusters have been arranged according to their placement in the various tree figures, and are listed by branch order from top to bottom. The cluster labels within each of the figures have also been outlined with the color of the dominant function of the proteins contained within.

In addition to the versions of the trees without node labels or branch lengths, there are also versions of each of those trees that do use branch lengths and node labels

(**Figures 4.5.2** and **4.5.4**). The labels use the internal indices for each of the domains.

The associated tables (**Tables 4.5.2** and **4.5.4**) provide keys to those domains. The

branch lengths are calculated based on the number of iterations a branch survives before

being merged with another branch. Each iteration adds 0.1 to the branch length.

Key:
Synthases
Ligases / synthetases
ATP-binding proteins / permeases
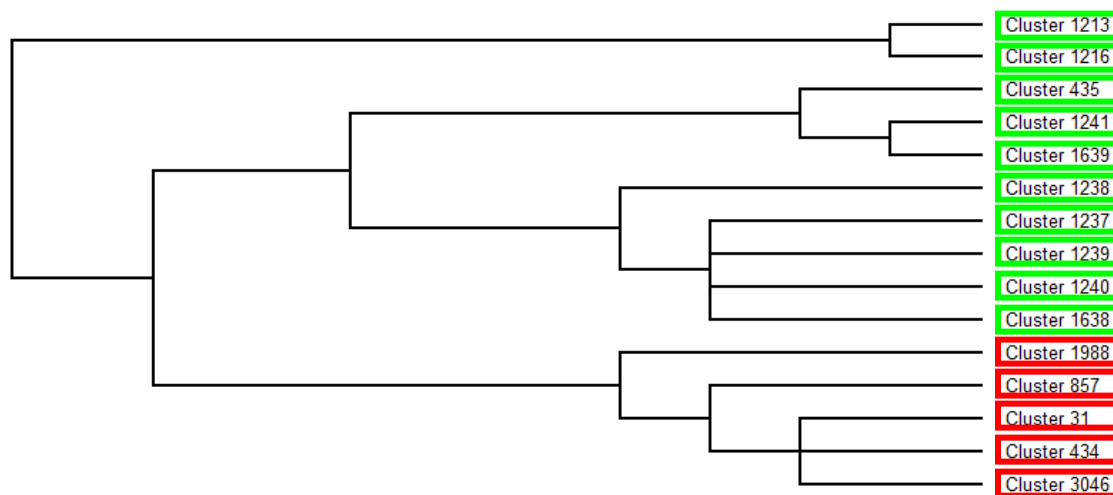
Figure 4.5.1: The version of Tree 187 from the *B. subtilis* data without node labels or branch lengths.

Table 4.5.2: The clusters present in Tree 187. Domain architecture is given on the right of each cluster header.

Cluster 1213	PP-binding, AMP-binding, AMP-binding_C, adh_short, KR, Polysacc_synt_2, Condensation, HxxPF_rpt, ketoacyl-synt, Ketoacyl-synt_C, Thiolase_N, AMP-binding_C_2, PS-DH, AATase
1303: P40806 - Polyketide synthase PksJ	
Cluster 1216	PP-binding, AMP-binding, AMP-binding_C, adh_short, KR, Epimerase, Condensation, HxxPF_rpt, ketoacyl-synt, Ketoacyl-synt_C, Thiolase_N, PS-DH, Poty_PP, DUF1307
1306: O31782 - Polyketide synthase PksN	
Cluster 435	PP-binding, AMP-binding, AMP-binding_C, Abhydrolase_6, Abhydrolase_5, Condensation, Thioesterase, HxxPF_rpt, HicB
454: P45745 - Dimodular nonribosomal peptide synthase	
Cluster 1241	PP-binding, AMP-binding, AMP-binding_C, Abhydrolase_6, Abhydrolase_3, Condensation, Thioesterase, DUF2974, AATase
1333: O31827 - Plipastatin synthase subunit E	
Cluster 1639	Transketolase_C, PP-binding, AMP-binding, AMP-binding_C, Abhydrolase_6, Abhydrolase_5, Condensation, Thioesterase, HxxPF_rpt
1773: Q08787 - Surfactin synthase subunit 3	
Cluster 1238	PP-binding, AMP-binding, AMP-binding_C, Condensation, HxxPF_rpt, Transferase, BacteriocIIc_cy
1330: P39846 - Plipastatin synthase subunit B	
Cluster 1237	PP-binding, AMP-binding, AMP-binding_C, Condensation, HxxPF_rpt, SPOB_ab
1329: P39845 - Plipastatin synthase subunit A	
Cluster 1239	PP-binding, AMP-binding, AMP-binding_C, Condensation, HxxPF_rpt
1331: P39847 - Plipastatin synthase subunit C	
1772: Q04747 - Surfactin synthase subunit 2	
Cluster 1240	PP-binding, AMP-binding, AMP-binding_C, UPF0122, Condensation, HxxPF_rpt
1332: P94459 - Plipastatin synthase subunit D	
Cluster 1638	PP-binding, AMP-binding, AMP-binding_C, Condensation, HxxPF_rpt, AATase
1771: P27206 - Surfactin synthase subunit 1	
Cluster 1988	AMP-binding, AMP-binding_C, AMP-binding_C_2, Trigger_C, Lipoprotein_3
2164: P96575 - Putative acyl--CoA ligase YdaB	
Cluster 857	AMP-binding, AMP-binding_C, AMP-binding_C_2
915: O07610 - Long-chain-fatty-acid--CoA ligase	
2539: O07619 - Uncharacterized acyl--CoA ligase YhfT	
Cluster 31	AMP-binding, AMP-binding_C
31: P39062 - Acetyl-coenzyme A synthetase	
475: P39581 - D-alanine--poly(phosphoribitol) ligase subunit 1	
914: P94547 - Long-chain-fatty-acid--CoA ligase	
1014: P23971 - 2-succinylbenzoate--CoA ligase	
2866: O31826 - Putative acyl-CoA synthetase YngI	
Cluster 434	AMP-binding, AMP-binding_C, SAP
453: P40871 - 2,3-dihydroxybenzoate-AMP ligase	
Cluster 3046	AMP-binding, AMP-binding_C, DUF4414
3390: C0SPB0 - Uncharacterized acyl--CoA ligase YtcI	

Figure 4.5.2: Tree 187 with branch lengths, node labels, and domain architecture displayed.

Table 4.5.3: Key to the domain indices present in
Figure 4.5.2.

| Index Number	Domain
75	Transketolase_C
109	PP-binding
114	AMP-binding
115	AMP-binding_C
378	UPF0122
437	adh_short
439	KR
440	Epimerase
441	Polysacc_synt_2
497	Abhydrolase_6
498	Abhydrolase_5
596	Abhydrolase_3
1091	SAP
1092	Condensation
1093	Thioesterase
1094	HxxPF_rpt
1095	HicB
1334	ketoacyl-synt
1335	Ketoacyl-synt_C
1336	Thiolase_N
1878	AMP-binding_C_2
2334	DUF2974
2336	PS-DH
2337	AATase
2338	Poty_PP
2340	DUF1307
2372	SPOB_ab
2373	Transferase
2374	BacteriocIIc_cy
2432	Trigger_C
3369	Lipoprotein_3
4408	DUF4414
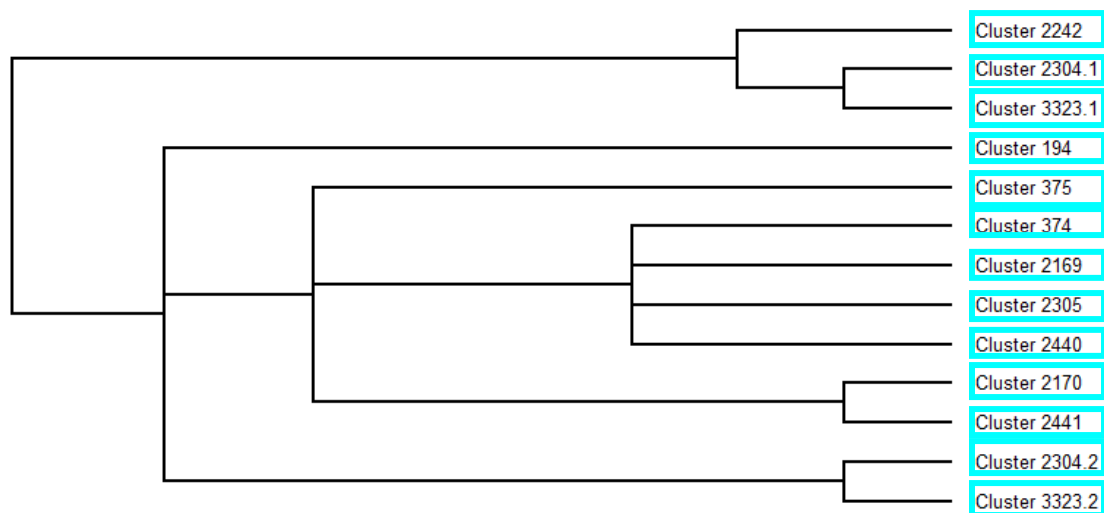
Figure 4.5.3: The version of a subset of Tree 656 from the *B. subtilis* data without node labels or branch lengths.

Table 4.5.4: The clusters present in the subset of Tree 656. Domain architecture is given on the right of each cluster header.

Cluster 2242	AAA_16, ABC_tran, SMC_N, AAA_29, DUF258, ABC_ATPase, NB-ARC, ABC_membrane, ABC_membrane_3, Chordopox_L2
2453: P71082 - Putative multidrug export ATP-binding/permease protein YgaD	
Cluster 2304	AAA_16, ABC_tran, AAA_21, SMC_N, SbcCD_C, AAA_29, DUF258, AAA_23, MMR_HSR1, AAA_10, FtsK_SpoIIIE, AAA_22, AAA_17, AAA_30, Dynamin_N, ABC_membrane
2523: O07549 - Probable multidrug resistance ABC transporter ATP-binding/permease protein YheH	
Cluster 3323	AAA, AAA_16, ABC_tran, AAA_21, SMC_N, SbcCD_C, AAA_29, DUF258, DEAD, AAA_10, AAA_25, AAA_22, AAA_18, AAA_17, AAA_33, MobB, AAA_30, ABC_membrane, ABC_membrane_3
3719: P45861 - Uncharacterized ABC transporter ATP-binding protein YwjA	
Cluster 194	AAA_16, ABC_tran, AAA_21, SMC_N, SbcCD_C, AAA_29, DUF258, AAA_23, AAA_25, AAA_22, AAA_18, AAA_17, ABC_ATPase, AAA_28, ABC_membrane, ABC_membrane_2, ABC_membrane_3
202: O06967 - Multidrug resistance ABC transporter ATP-binding/permease protein BmrA	
Cluster 375	ABC_tran, AAA_21, SMC_N, AAA_29, DUF258, FtsK_SpoIIIE, ABC_ATPase, ABC_membrane, IncA
390: P94367 - ATP-binding/permease protein CydD	
Cluster 374	ABC_tran, AAA_21, SMC_N, AAA_15, AAA_29, ArgK, DUF258, AAA_23, MMR_HSR1, AAA_17, ABC_membrane, MscS_TM
389: P94366 - ATP-binding/permease protein CydC	
Cluster 2169	AAA_16, T2SE, ABC_tran, AAA_21, SMC_N, AAA_29, AAA_23, FtsK_SpoIIIE, G-alpha, AAA_18, AAA_14, ABC_ATPase, DUF87, ABC_membrane
2371: P54718 - Uncharacterized ABC transporter ATP-binding protein YfiB	
Cluster 2305	Miro, AAA_16, T2SE, ABC_tran, SMC_N, AAA_29, DUF258, AAA_23, MMR_HSR1, cobW, AAA_25, ATP-synt_ab, FtsK_SpoIIIE, AAA_22, G-alpha, AAA_18, AAA_17, AAA_14, MobB, UPF0079, DUF87, NB-ARC, Zeta_toxin, ABC_membrane, TrwB_AAD_bind, Viral_helicase1
2524: O07550 - Probable multidrug resistance ABC transporter ATP-binding/permease protein YheI	
Cluster 2440	AAA_16, AAA_PrkA, ABC_tran, SMC_N, AAA_29, AAA_18, AAA_17, ABC_ATPase, ABC_membrane, Sterol-sensing
2680: O31707 - Uncharacterized ABC transporter ATP-binding protein YknU	
Cluster 2170	Mg_chelatase, AAA, AAA_16, T2SE, ABC_tran, AAA_21, SMC_N, AAA_29, DUF258, AAA_23, AAA_19, SRP54, AAA_10, AAA_25, AAA_22, AAA_18, AAA_17, AAA_33, MobB, AAA_30, AAA_28, Zeta_toxin, ABC_membrane, ABC_membrane_2, ABC_membrane_3, IstB_IS21, APS_kinase
2372: P54719 - Uncharacterized ABC transporter ATP-binding protein YfiC	
Cluster 2441	AAA_5, AAA, AAA_16, T2SE, ABC_tran, AAA_21, SMC_N, SbcCD_C, AAA_29, DUF258, AAA_23, AAA_19, AAA_10, AAA_25, AAA_22, AAA_18, AAA_17, AAA_30, Zeta_toxin, ABC_membrane, IstB_IS21, GPDPase_memb, FAST_2
2681: O31708 - Uncharacterized ABC transporter ATP-binding protein YknV	
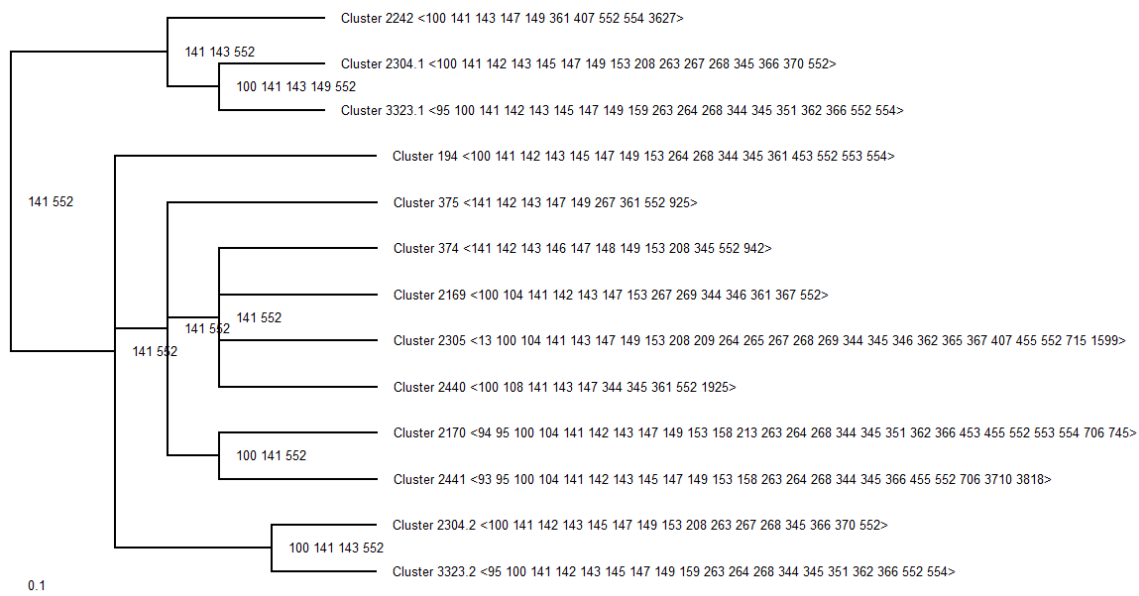
Figure 4.5.4: The Tree 656 subset with branch lengths, node labels, and domain architecture displayed.

Table 4.5.5: Key to the domain indices present
in **Figure 4.5.4**.

Index Number	Domain
13	Miro
93	AAA_5
94	Mg_chelatase
95	AAA
100	AAA_16
104	T2SE
108	AAA_PrkA
141	ABC_tran
142	AAA_21
143	SMC_N
145	SbcCD_C
146	AAA_15
147	AAA_29
148	ArgK
149	DUF258
153	AAA_23
158	AAA_19
159	DEAD
208	MMR_HSR1
209	cobW
213	SRP54
263	AAA_10
264	AAA_25
265	ATP-synt_ab
267	FtsK_SpoIIIE
268	AAA_22
269	G-alpha

344	AAA_18
345	AAA_17
346	AAA_14
351	AAA_33
361	ABC_ATPase
362	MobB
365	UPF0079
366	AAA_30
367	DUF87
370	Dynamin_N
407	NB-ARC
453	AAA_28
455	Zeta_toxin
552	ABC_membrane
553	ABC_membrane_2
554	ABC_membrane_3
706	IstB_IS21
715	TrwB_AAD_bind
745	APS_kinase
925	IncA
942	MscS_TM
1599	Viral_helicase1
1925	Sterol-sensing
3627	Chordopox_L2
3710	GPDPase_memb
3818	FAST_2

4.6 Discussion of *B. subtilis* Data

The *B. subtilis* dataset generated a large number of trees compared to the RGS

dataset. Although the tier chart is unreadable due to the high number of edge crossings,

the final tree visualizations still can be viewed clearly in most cases. The main purpose

of testing the program on this data was to understand its performance on proteome-scale

data, as this kind of analysis is not possible using alignment-based methods.

An important detail to point out is in the Tree 656 visualizations (**Figures 4.5.3**

and **4.5.4**). Some of the clusters (2304 and 3323) are duplicated in separate branches.

This is due to how the program adds clusters to the Newick string, and was allowed to

remain as a feature, rather than being seen as an error. **Figure 4.6.1** shows a tier chart

generated by extracting only the lines of data from the HMMER file that pertain to the

branch of Tree 656 seen in **Figures 4.5.3** and **4.5.4**. The cluster numbers are therefore

different. **Table 4.6.1** provides a key to show equivalent cluster indices between the two

sets of figures.

Table 4.6.1: Key to the cluster indices of **Figure 4.6.1**
and the original Tree 656 visualizations.

Original Indices	Figure 4.6.1 Indices
194	1
374	2
375	3
2169	4
2170	5
2242	6
2304	7
2305	8
2440	9
2441	10
3323	11

Figure 4.6.1 (next page): Illustration of the tier chart for the branch of Tree 656 seen in **Figures 4.5.3** and
4.5.4.
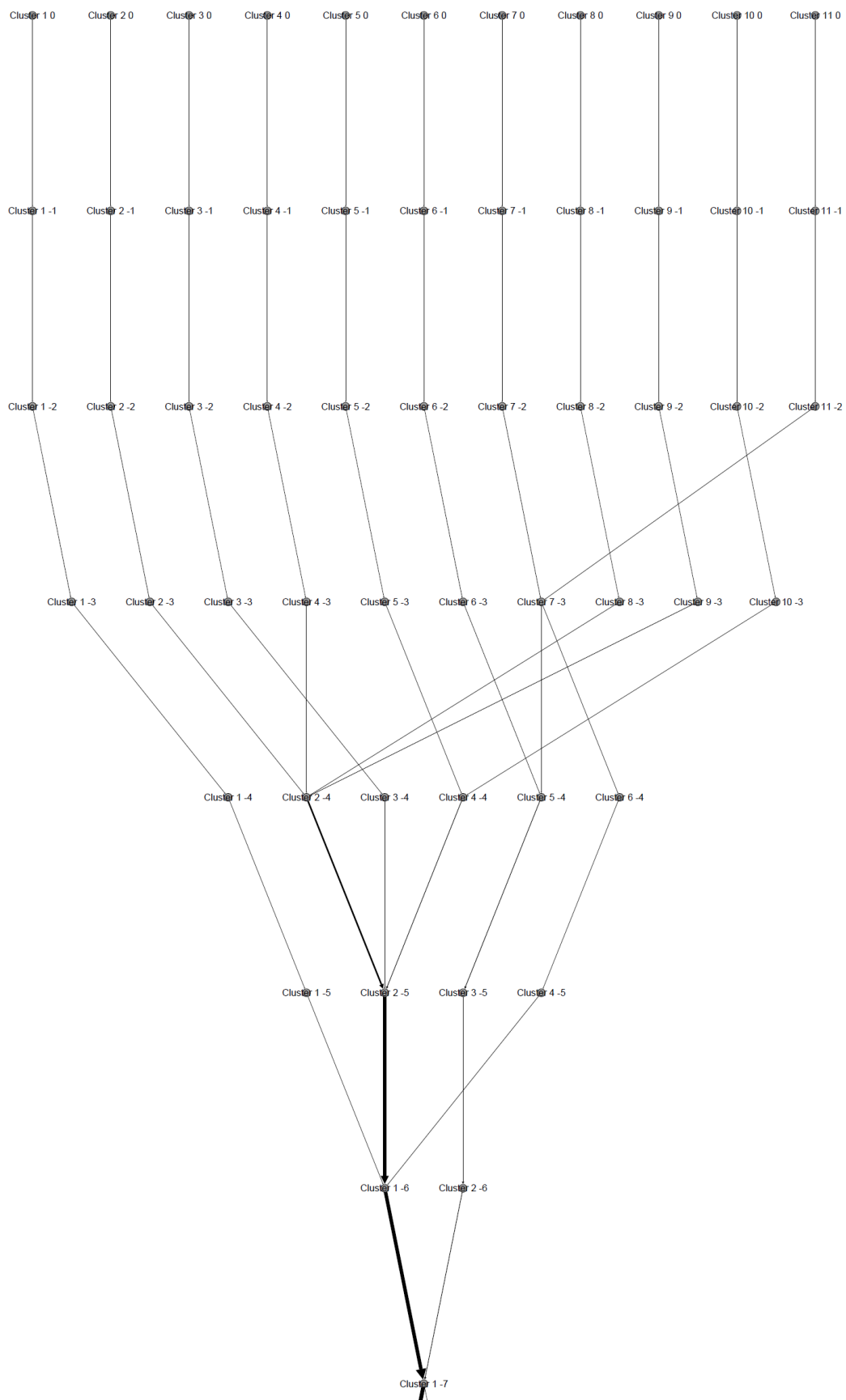
Figure 4.6.1 illustrates the reason for the cluster duplication seen in Tree 656. In this figure, the duplicated clusters are clusters 7 and 11. During the -3 iteration, 7 and 11 both merge together, but then immediately split apart in the subsequent iteration. One set of proteins merges with Cluster 6 in the -4 iteration, and the other set eventually merges with the branch containing clusters 1, 2, 3, 4, 5, 8, 9, and 10 in the -6 iteration. The branch containing Cluster 6 also subsequently merges with this same branch in the -7 iteration. Because the half that split off alone carries different information than the half that merged with Cluster 6, two copies of clusters 7 and 11 are made, one that is nested in the large polytomy that was just mentioned, and the other (with Cluster 6), that forms a sister group to this entire branch. This was seen as an asset, because this shows two different sets of relationships between the protein clusters, which would be advantageous to know about if and when it ever occurs.

Trees 187 and 656 both exhibit strong uniformity of the functional groups of the proteins contained within. The branch of Tree 656 that was investigated is composed entirely of ATP-binding proteins, while Tree 187 is composed of two groups of proteins: ligases and synthases, which are very closely related in their function. In this tree, the branch containing ligases forms a sister group with a branch containing synthases, which is an expected relationship, as opposed to the protein functions being scattered on different branches of the tree.

CHAPTER 5
CONCLUSION AND FUTURE WORK

As the available protein sequences and proteome data continues to accumulate in the coming years, the demand for work to be able to characterize unknown proteins and understand the relationships between protein groups will increase as well. Although methods like BLAST and maximum likelihood analysis are helpful in many cases, they cannot be used to analyze the relationships of all the members of a proteome. Instead, other methods must be used.

The bit vector program that has been described in this thesis has been shown to accurately cluster proteins together and to form phylogeny-like trees illustrating the relationships between proteins with similar but not identical domain architecture. This domain-based approach offers an alternative to more typical alignment-based techniques. Furthermore, it can be used on proteome data without concern for the need of a universal conserved subsequence to base the alignment on. Even if such a subsequence were present in proteome data, the amount of time it would take to perform a maximum likelihood phylogenetic reconstruction would be prohibitive. Tests have demonstrated that maximum likelihood can take several hours to complete on merely a few dozen proteins. For instance, the RGS dataset, consisting of sequences extracted from 66 proteins, took over 27 hours to complete a maximum likelihood analysis, compared to the bit vector approach taking only about 23 seconds to finish using the same data. Bacterial proteomes typically consist of thousands of diverse proteins. Not only is no alignment possible on such data, but even if it were, the size of the input data makes phylogenetic analysis unfeasible for maximum likelihood.

Although the neighbor-joining test conducted on the RGS data did finish in less time than the bit vector method (4 seconds versus 23 seconds), this also was only after taking shortcuts. Neighbor-joining is a faster algorithm than maximum likelihood, but its results are generally not deemed to be as high-quality as results from maximum likelihood. Furthermore, bootstrapping is a common requirement for generating quality trees using any phylogenetic algorithm, and so, although neighbor-joining can run faster than the bit vector approach by eliminating bootstrap replicates, the quality of the resulting tree will not be as reliable as what can be obtained from a bootstrapped maximum likelihood tree.

This thesis has made the contribution of the bit vector program, which extends the work from Shah's thesis to iterate over multiple E-value thresholds with increasing stringency. Not only does this show how the protein clusters change as weaker domains are removed from consideration, but this iterative process, coupled with how later clusters "inherit" their proteins from earlier ones, can be used to generate Newick trees conveniently displaying these relationships.

The bit vector program also offers an alternative analysis method for any type of protein dataset based on domain architecture rather than sequence similarity. Because three-dimensional structure of proteins is partly dictated by what domains are present, this domain-based alternative offers a different perspective to understand protein function, which sometimes may not be obvious using alignment-based methods. For example, tests have shown that some functionally-related protein groups may not necessarily be grouped together in a maximum likelihood tree. This is based purely on the sequence similarity and alignments of these proteins, rather than inference of three-

dimensional structure. This means that the bit vector program provides a higher-level overview of protein relationships.

Finally, as mentioned above, the bit vector program also offers a tool that makes analysis of large datasets of mostly unrelated proteins, as occurs frequently in proteome data, possible without the need for first dividing the dataset into smaller subsets of proteins that share conserved subsequences capable of being aligned. Furthermore, the initial use of this method would break such a dataset down into subsets that could be used in more focused maximum likelihood analysis, as the resulting trees would not only group the related proteins together, but would also highlight the primary conserved subsequence that could be used for alignment.

5.1 Future Work

Code Optimization

The current version of the bit vector algorithm operates very quickly on relatively small datasets, particularly those with fewer than 100 proteins and total unique domains. A clear example of when the program operates inefficiently is in the case of the *B. subtilis* data, with 3,973 proteins and 4,737 unique domains, in which case the bit vector algorithm took over 70 hours to complete. However, this is still considerably more efficient than a maximum likelihood approach, which (with bootstrapping) required over a day to complete just on the RGS data. Scaling the dataset size up by two orders of magnitude for the proteome-sized test would be expected to make the running time of a maximum likelihood approach completely impractical. Even without bootstrapping in the maximum likelihood approach, the best observed running time was 6 minutes on the Trex server using the RGS data as input.

No major attempts have been made to decrease the running time of the bit vector program, and so doing this would be a significant goal for future work. One possible improvement might be to replace the core clustering technique with an efficient alternative from the related work summarized in **Section 2.2**. Attempts at using program profiling methods would also be expected to aid in discovering any inefficient blocks of code that could be improved.

Domain Order

Currently, this technique ignores the order of the domains in a protein's architecture, instead focusing merely on a given domain's presence. For example, a protein may possess domains *A*, *B*, and *C*, but the order of the occurrence of these three domains along the sequence could potentially differ in other proteins otherwise with the same architecture. It may be beneficial to investigate if this affects the relationships between related proteins, or their possible function.

Improved Visualizations

The GEXF visualizations of the bipartite graphs showing the protein-domain matches for each iteration, as well as the tier chart are useful for smaller datasets (such as the RGS data), and work well with fewer than 100 proteins and unique domains. However, scaling this up to the proteome level, such as the *B. subtilis* data, generates largely unreadable visualizations, as the number of edge crossings becomes extreme. As the goal of this thesis was not to focus on visualization techniques, but was instead focused on outputting the trees generated from the data, improvement of the visualizations was not looked into to any great extent, but this would be an important

improvement for future incarnations of this method. Future improvements might attempt to use better visualization techniques to allow even large datasets to be viewed clearly.

First-Iteration Singleton Trees

Due to the way the tier-based charts are generated, it is possible for the final listing of Newick trees to leave out the reporting of singleton trees if they are removed before the start of the second iteration due to E-value threshold. This has only been observed on artificial datasets specifically designed to test the algorithm's performance, and would generally be expected to be quite rare occurrences in any real-world data. As the primary interest in the use of the method is the generation of non-singleton trees to understand the relationships between protein clusters with similar domain architecture, this was deemed a very minor issue. However, the possibility still exists of this occurring, and so may be worthwhile to fix the approach in the future.

Leaf Duplication in Some Cases

Generally speaking, it has frequently been observed in the tier-based charts that clusters do not split or merge uniformly from one iteration to the next. Instead, it is possible for individual proteins to have different E-values for the matchings of given domains. As a result, some proteins may leave their original cluster and join another during different iterations, rather than during one uniform step. This has the effect of having two or more cluster lineages merging several times over the course of more than one iteration. Currently, the way the algorithm is designed, this will not lead to duplication of clusters in the final Newick trees, because such duplication is something that the algorithm specifically checks for before it merges two portions of a Newick string. As a result, only the first occurrence of two or more lineages merging will be

considered in constructing the tree, as any merging that occurs in subsequent iterations is simply repetition of the same relationship. However, this is only the case when two or more branches are merging together, forming a two-way split or polytomy on the resulting phylogenetic tree.

It has been observed in some cases that a given cluster may actually split, and different proteins may merge into separate branches of the phylogenetic tree. This was seen in Tree 656 of the *B. subtilis* data, as discussed in **Section 4.6**. This has the effect of duplicating the affected leaves onto two or more branches of the tree. Rather than being seen as a problem needing to be repaired, this was seen as a feature of the algorithm, as it accurately displays that the duplicated leaves are actually related to proteins in more than one branch. Due to the way the domain deletion with increased E-value threshold stringency works, it is not always true that the same domains may be deleted at the same time for all members of a cluster. In fact, it is possible for some proteins to lose a domain at an early iteration, while the other proteins may retain the same domain until the final iteration of that lineage. This can potentially change the domain architecture of the affected proteins, and so would lead to a justified split in the cluster's lineage.

As an illustration of this point, a hypothetical cluster may have the domain architecture *A*, *B*, *C*. However, one subset of proteins may have a very weak match for domain *A*, and very strong matches for *B* and *C*. Meanwhile, another subset may have very strong matches to *A*, but very weak matches to *B* and *C*. As a result, the cluster would see two groups split off: one with architecture *B*, *C*, and the other with architecture *A*. Furthermore, it may be possible for a third subset of proteins to have strong matches to all three domains, but such that they lose domain *A* at a much later iteration, while still

retaining domains *B*, and *C*. In this case, they would re-merge with the other *B*, *C* proteins that had already split off into a different branch. This could potentially lead to the leaf duplication in separate branches that has been described.

A problem arises, however, in cases where, for example, two clusters merge into one during one iteration, but then subsequently split along the original cluster groupings. That is, if Cluster 1 and Cluster 2 were to merge, but the new cluster later splits such that one cluster retains only Cluster 1 proteins, and the other cluster retains only Cluster 2 proteins, this will lead to a problem when the split lineages merge with any outlying clusters. The reason for this is, even though only certain proteins are merging with a new cluster, they still retain the ancestry of both Cluster 1 and Cluster 2, and so both clusters would be reported in the leaf duplication, even though only one cluster's proteins were participating in a given merging. This is a problem that should be investigated and fixed in future work.

Consideration of Individual Proteins in a Lineage

Related to the above issue, the current version of the technique simply looks at the ancestry (in terms of previously merged clusters) of each cluster in a given iteration. It pays no attention to what actual proteins still remain within that cluster (the others potentially having been deleted due to the E-value threshold). In the tier charts, in order for an edge to exist between nodes in adjacent iterations, only a single protein needs to merge from the first cluster into the second cluster, although this edge will have a greater thickness if more proteins make the same journey. This means that two or more clusters that merge together in one iteration may have lost proteins in a subsequent iteration, such that they no longer are participating in the lineage. The implications of this approach

have not been fully investigated, but in future work it may be important to give greater

consideration to where in each lineage individual proteins actually remain, and where

they have already been deleted by the threshold. From a visualization perspective, one

example might be to visualize the chart such that if an edge is clicked on or hovered over,

the information on what proteins are actually traversing that edge could be displayed.

Singleton Cluster Labels

In the case of clusters containing only one protein, rather than labelling the cluster

as "Cluster *X*" on the tree (with *X* being the index number of that particular cluster), it

may be better to simply replace the cluster label with the name of the sole protein

contained within.

Use of Fewer E-value Thresholds

Currently, the bit vector program compiles a list of all E-values seen in the

HMMER data file and uses this list as a basis for iterating over the threshold. A sorted

descending-order list is created of the E-values, and after a given iteration, whatever the

next order of magnitude is in the list will be deleted for the following iteration. One

possible increase in efficiency could come from skipping some of these E-value entries.

For instance, rather than iterating from 10^{-5} to 10^{-10} one order of magnitude at a time

(provided that each order of magnitude is represented in the data, which may not always

be so, depending on the input), one possible approach might be to simply skip the in-

between values and simply iterate over 10^{-5} one iteration, and go directly to 10^{-10} in the

next iteration. This will reduce the total number of iterations during which the binary

matrix is updated and iterated over to construct the clusters.

The effects that this iterative technique would have on the resulting data should be investigated further to see if there are negative impacts on the quality of the final trees. As this would reduce the number of levels in the tier chart, one effect that could be expected from this approach is the reduction of various nested two-way splits into a single polytomy. For example, if after iterating over each distinct E-value, a tree has the structure $((((A,B),C),D),E)$, if the number of iterations is reduced, this structure could become the polytomy (A,B,C,D,E), depending on the exact relationships of the E-values present.

Combining Final Trees

As it is, the bit vector approach potentially generates separate trees, with no overlap, depending on how diverse the input data is. For example, if the roots of two trees do not share the same domain architecture, no merging is possible, otherwise they would have been generated as branches within the same tree. This would most likely require a different metric to measure similarity between the trees. One possible approach might be to assess the similarity of the domain architecture present between each tree, and place the trees with the closest similarity together as sister groups. This process could be repeated until each tree has been absorbed into a single, final tree containing all of the trees and clusters found by the method.

References

1 – Altschul, SF; Gish, W.; Miller, W.; Myers, EW; Lipman, DJ. (1990) "Basic local alignment search tool", *J. Mol. Biol.* 1990 Oct 5; 215 (3): 403-10.

2 – National Center for Biotechnology Information, BLAST: Basic Local Alignment Search Tool, <http://blast.ncbi.nlm.nih.gov/Blast.cgi>. Accessed July 2016.

3 – Thompson, JD; Higgins, DG; Gibson, TJ (1994) "CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice", *Nucleic Acids Res.* 1994 Nov 11; 22(22): 4673-80.

4 – Edgar, RC (2004) "MUSCLE: multiple sequence alignment with high accuracy and high throughput", *Nucleic Acids Res.* 2004; 32(5): 1792–1797.

5 – Katoh, K.; Misawa, K.; Kuma, K.; Miyata, T. (2002) "MAFFT: a novel method for rapid multiple sequence alignment based on fast Fourier transform" *Nucleic Acids Res.* 2002 Jul 15; 30 (14): 3059-66.

6 – Saitou, N.; Nei, M. (1987) "The neighbor-joining method: a new method for reconstructing phylogenetic trees" *Molecular Biology and Evolution*, vol. 4, issue 4, pp. 406-425, July 1987.

7 – Farris, JS (1970) "Methods for computing Wagner trees" *Systematic Zoology* 19, 83-92.

8 – Fitch, WM (1971) "Toward defining the course of evolution: minimum change for a specified tree topology" *Systematic Zoology* 20 (4), 406-416.

9 – Yang, Z.; Kumar, S.; Nei, M. (1995) "A new method of inference of ancestral nucleotide and amino acid sequences" *Genetics* 1995 Dec; 141 (4): 1641-50.

10 – Koshi, JM; Goldstein, RA. (1996) "Probabilistic reconstruction of ancestral protein sequences" *J. Mol. Evol.* 1996 Feb; 42 (2): 313-20.

11 – Pagel, M. (1999) "The Maximum Likelihood Approach to Reconstructing Ancestral Character States of Discrete Characters on Phylogenies" *Systematic Biology* 48 (3): 612–622.

12 – Enright, AJ; Van Dongen, S.; Ouzounis, CA (2002) "An efficient algorithm for large-scale detection of protein families" *Nucleic Acids Res.* 2002 Apr 1; 30 (7): 1575-84.

13 – Shah, Neethu (2013) "Clustering and Classification of Multi-domain Proteins", Computer Science and Engineering: Theses, Dissertations, and Student Research, Department of Computer Science and Engineering, University of Nebraska-Lincoln.

14 – Prelić, A.; Bleuler, S.; Zimmermann, P.; Wille, A.; Bühlmann, P.; Gruissem, W.; Hennig, L.; Thiele, L.; Zitzler, E. (2006) "A systematic comparison and evaluation of biclustering methods for gene expression data", *Bioinformatics*, (2006) 22 (9): 1122-1129.

15 – Bimax: Supplementary Materials Web Page for Bioinformatics, <http://people.ee.ethz.ch/~sop/bimax/>. Accessed June 2014.

16 – Oghabian, A.; Kilpinen, S.; Hautaniemi, S.; Czeizler, E. (2014) "Biclustering Methods: Biological Relevance and Application in Gene Expression Analysis", *PLoS ONE* 9(3): e90801.

17 – Shamir, R.; Maron-Katz, A.; Tanay, A.; Linhart, C.; Steinfeld, I.; Sharan, R.; Shiloh, Y.; Elkon, R. (2005) "*EXPANDER* – an interactive program suite for microarray data analysis", *BMC Bioinformatics*, 2005, 6: 232.

18 – Tanay, A.; Sharan, R.; Shamir, R. (2002) "Discovering statistically significant biclusters in gene expression data", *Bioinformatics*, vol. 18 suppl. 1 2002, pgs. S136-S144.

19 – EXPANDER: A Gene Expression Analysis and Visualization Software, <http://acgt.cs.tau.ac.il/expander/overview.html>. Accessed June 2014.

20 – Király, A.; Gyenesei, A.; Abonyi, J. (2014) "Bit-table based biclustering and frequent closed itemset mining in high-dimensional binary data", *Scientific World Journal*, 2014 Jan 30; 2014: 870406.

21 – Santamaría, R.; Therón, R.; Quintales, L. (2014) "BicOverlapper 2.0: visual analysis for gene expression", *Bioinformatics*, 2014 Jun 15; 30(12): 1785-6.

22 – Sun, P.; Speicher, NK; Röttger, R.; Guo, J.; Baumbach, J. (2014) "Bi-Force: large-scale bicluster editing and its application to gene expression data biclustering", *Nucleic Acids Res.* 2014 May; 42(9): e78.

23 – Henriques, R.; Madeira, SC (2014) "BicSPAM: flexible biclustering using sequential patterns," *BMC Bioinformatics*, 2014 May 6; 15: 130.

24 – Streit, M.; Gratzl, S.; Gillhofer, M.; Mayr, A.; Mitterecker, A.; Hochreiter, S. (2014)

"Furby: fuzzy force-directed bicluster visualization", *BMC Bioinformatics*, 2014; 15 Suppl 6: S4.

25 – Moore, AD; Held, A.; Terrapon, N.; Weiner 3rd, J.; Bornberg-Bauer, E. (2014) "DoMosaics: software for domain arrangement visualization and domain-centric analysis of proteins", *Bioinformatics*, (2014) 30 (2): 282-283.

26 – Eddy, SR. (2007) *HMMER - biosequence analysis using profile hidden Markov models*. Available: <http://hmmer.janelia.org>.

27 – Eddy, SR (2008) "A Probabilistic Model of Local Sequence Alignment That Simplifies Statistical Significance Estimation" *PLoS Comput. Biol.* 2008 May; 4 (5): e1000069.

28 – Eddy, SR; Wheeler, TJ (2015) *HMMER User's Guide*, Howard Hughes Medical Institute, <http://eddylab.org/software/hmmer3/3.1b2/Userguide.pdf>. Accessed July 2016.

29 – Bastian, M.; Heymann, S.; Jacomy, M. (2009) *Gephi: An Open Source Software for Exploring and Manipulating Networks*, International AAAI Conference on Weblogs and Social Media.

30 – Gephi – The Open Graph Viz Platform, <http://gephi.org>. Accessed July 2016.

31 – GEXF File Format, Gephi Website, <http://gephi.org/gexf/format>. Accessed July 2016.

32 – Page, RDM (1996) "Tree View: An application to display phylogenetic trees on personal computers". *Computer Applications in the Biosciences*, 12 (4): 357–358.

33 – Stamatakis, A. (2006) "RAxML-VI-HPC: maximum likelihood-based phylogenetic analyses with thousands of taxa and mixed models" *Bioinformatics* 22 (21): 2688–2690.

34 – Boc, A.; Diallo, AB; Makarenkov, V. (2012), "T-REX: a web server for inferring, validating and visualizing phylogenetic trees and networks" *Nucleic Acids Res.* 40 (W1), W573-W579.

35 – Nei, M.; Kumar, S. (2000) *Molecular Evolution and Phylogenetics*. Oxford University Press, New York.

36 – Rosetta Code, Sorting Algorithms: Merge Sort, <http://rosettacode.org/wiki/Sorting_algorithms/Merge_sort#MATLAB>. Accessed Oct. 2015.

APPENDIX A
SUPPLEMENTARY MATERIALS

A.1 Program Documentation

The current version of the bit vector clustering program does a number of things. It outputs a plain text file for each order of magnitude of E-value in the dataset, containing a listing of all the clusters, what domains the clusters contain, and what proteins are included in each cluster. In addition, it also outputs GEXF-format files, again, one for each order of magnitude of E-value, which allows the proteins and domains for each iteration to be visualized as a bipartite graph in Gephi. The bipartite graphs are currently designed so that proteins that have been eliminated due to E-value threshold requirements no longer have any nodes, but any "abandoned" domain nodes are still present (but without edges). The proteins are arranged by cluster. All of the proteins from cluster X will be present in an adjacent arrangement. Following this, there will be a gap before the next cluster is displayed.

In addition, one final, tier-based GEXF file is produced. This file visualizes each cluster as a single node. All of the clusters from a particular iteration of the program (one iteration for each order of magnitude of E-value) is arranged in a tier. If a given cluster contains at least one protein that is "passed on" to a cluster in the next tier, an edge will be present between those clusters, indicating that the subsequent cluster has "inherited" at least one protein from the "ancestral" cluster.

Program Overview:
```
prime_funct()
|       hmmer_script (script)
|       matrix_extract (script)
|       |_____ mergeSort()
|       file_out()
|       domains()
|       node_data()
|       save_mat()
|       get_weights()
|       |_____ log_calc()
|       gexf_out()
|       get_node_IDs()
|       get_clust_anc()
|       |       ind_extract()
|       |       de_newick()
|       |       list_substr()
|       |_____ inc_str()
|       newick_out()
|       newick_out_lbl()
|       |_____ un_label()
|       re_number()
|       |       un_label()
|       |_____ de_newick()
|       print_newick()
|       get_edges()
|       mergeSort2()
|_____ tier_gexf()
```

prime_funct()
This function ties all of the other scripts and functions together. Its input argument list is as follows:

fileName – mandatory input argument; the file name of the HMMER data; example: 'RGS_hmmer.out' [note that quotation marks must be included otherwise interprets the input argument as a variable name or function].

varargin – optional input arguments; intended to allow the user to specify an order of magnitude to terminate at; example: -10 [in which case, the algorithm will only decrease the E-value threshold to a magnitude of 10^{-10}, at which point it will stop]; additionally, an order of magnitude to start at can also be specified, if both start and stop parameters are given.

Full Details on Input:
Example Format (in the Matlab console):
prime_funct('RGS_hmmer.out',-50,-10);

This invocation of the program accesses file "RGS_hmmer.out" for the HMMER data, begins at E-values of 10^{-10} (any higher values are deleted and ignored), and stops once it clusters everything based on values of 10^{-50}, but no smaller. In such a case, the "mat 0" files will consist of clusters based on the 10^{-10} threshold. Also note that the third parameter can be, at most, -1.

The second and third parameters are optional, but using the third parameter requires the second parameter to also be specified. An acceptable invocation would also be:

prime_funct('RGS_hmmer.out');

In this case, the same file is accessed for the HMMER data, but there will no longer be any restrictions on the start or stop E-value thresholds. It will begin at whatever the initial E-value happens to be (although internally, this has an ultimate cutoff at E-values of 1.0 – anything greater will not even be considered), and continue until all E-value orders of magnitude are exhausted. Note that setting the second parameter to "-Inf" is identical to the default behavior just described.

It is important to bear in mind that the numerical component in the "mat X" titles corresponds to the order of magnitude of E-value that was just deleted – for example, "mat -10" contains E-values at 10^{-11} at the highest, not 10^{-10}, because all 10^{-10} values were deleted at the start of the iteration.

prime_funct() calls the scripts **hmmer_script** and **matrix_extract**, which read the HMMER input file, reformat it so as to be easier to import into Matlab, and then extract the data into a cell array. The function then iterates over the data, outputting the data files (for example, "mat 0.txt" and "mat 0.gexf") for each iteration, as the E-value threshold is lowered (made more stringent). This data is generated and interpreted by calls to the functions **file_out()**, **domains()**, **save_mat()**, **get_weights()**, and **gexf_out()**.

hmmer_script

This script opens the file specified by the fileName input argument from **prime_funct()**. Its main purpose is to scan through the HMMER file line by line and convert any white space separating data fields with a single tab (the HMMER files can have fields separated by a varying number of spaces, which makes importing of the data into Matlab problematic). The updated lines of information are then written to a new output file. This will be a tab-delimited version of the HMMER input file when the script finishes. It should also be noted that the original HMMER file has had any headers and footers manually deleted. Its only contents will therefore be the raw data itself, with no extra lines above or below it.

matrix_extract

This script uses the new tab-delimited file generated by **hmmer_script** and imports it into Matlab using Matlab's **textscan()** function. The data is stored in a cell array. This array is then used to generate several other data structures. The primary one is a simple binary $m \times n$ matrix, where m is the number of unique proteins in the HMMER data, and n is the number of unique domains present in the entire dataset. A cell in this matrix will be 0 if HMMER determined a given protein did not possess a given domain (or if its E-

value was given above the acceptable threshold). Alternatively, it will be 1 if HMMER did report finding the domain in the given protein. Another data structure is a matrix of identical dimensions to the binary matrix, but with the actual E-values replacing the cells equal to 1. The other cells will be 0 if the binary matrix also has a 0 at the corresponding cell. In the event an E-value is given as 0 by the HMMER data, this can be distinguished because the corresponding cell in the binary matrix will equal 1, and not 0. Finally, the E-values are also organized into a one-dimensional array, which is then sorted (using Merge Sort) in descending order (the weakest E-values at the start). This array will then be used in the iterative process (in **prime_funct()**) of removing weak E-values.

mergeSort()
This function is a modified implementation of Merge Sort taken from Rosetta Code [36]. It is modified to accept three input arrays instead of just one. The purpose is that two of the arrays will contain row and column indices for the data contained in the corresponding cell of the array being sorted, so the value contained there may be looked up directly afterwards in its original matrix. The function sorts the main array as normal, but also moves the associated row and column values so they remain synchronized with the values in the sorted array.
list – the array to be sorted; values are taken from a two-dimensional matrix.
row_ar – an array containing the row coordinates for the data contained in 'list', taken
     from the original matrix.
col_ar – an array containing the column coordinates for the data contained in 'list', taken
     from the original matrix.

The output of this function is the same three arrays, now sorted in ascending order based on the values in 'list'.

file_out()
This function outputs two files, "rows.txt" and "cols.txt". The former is a base-1 indexed list of the proteins found in the HMMER data, while the latter is a similar list of the domains. The input argument list is as so:
prot – a cell array generated by **matrix_extract** containing the accession numbers of the
     proteins in the HMMER data.
dom – a cell array generated by **matrix_extract** containing the names of the domains in
     the HMMER data.
x – the number of proteins in the list.
y – the number of domains in the list.

domains()
This function generates a list of clusters based on comparisons of the proteins' domain architecture. The input arguments are:
bitvec_mat – the binary matrix containing information on the domain architecture of each
     protein.
prot – a list of protein accession numbers.
dom – a list of domain names.
x – the number of proteins in the matrix.

y – the number of domains in the matrix.

name – a string based on the theme "mat *X*.txt", where *X* is the order of magnitude of the
current iteration.

members – a cell array that is initialized in **prime_funct()**; it has new data added onto it
for each subsequent iteration, and displays which cluster a given protein
participates in during a given iteration.

'list' is the principle output argument, and contains a three-column cell array where the
first column contains the binary strings representing a given domain architecture, the
second column contains the lists of protein indices in a given cluster, and the third
column contains the lists of domain indices in a given cluster. The second and third
columns are in the format "*a*_*b*_*c*…", where *a*, *b*, and *c* each represent either a protein or
domain index number.

'label_list' is very similar to 'list', but instead presents the proteins and domains using
their actual labels, not indices. This will be useful in outputting cluster lists containing
the actual names of the protein and domain nodes being clustered (as opposed to just
index values, which are difficult for humans to understand without using a look-up table
of indices).

'count' is a simple numerical value representing the number of clusters found.

'members', as described above, shows the cluster a protein participates in for each
iteration. **domains()** adds new data to this variable during each subsequent iteration.

The function iterates over the matrix and generates a binary string for each protein. It
then makes systematic comparisons between each protein's string. Protein 1 gets
compared to all other proteins, but higher-numbered proteins never get compared back to
proteins that have already been iterated over previously (so the sequence of comparisons
would be 1 to 2, 1 to 3, …, 1 to *n*th protein, then 2 to 3, 2 to 4, …, 2 to *n*th protein, etc.).
Once a protein has been matched into a cluster, it is removed from the list, so it is not
compared again.

node_data()

This function generates a cell array of the node data, which will be printed to a GEXF file
later. The input arguments are:

nodes_ar – a cell array containing the labels of the protein and domain nodes.

list – the list of clusters generated in **domains()**.

num_prot – the number of proteins.

list_row – the number of rows in 'list'.

The output arguments are:

nodes_out – the cell array of node labels.

count – the count of proteins.

The function generates a string in XML format, which is then inserted into the
'nodes_out' array. When finished, each node will have its own entry in the output array.

save_mat()
This function saves the set of clusters for a given iteration to a text file. The input arguments are:
list – the list of clusters generated in **domains()**.
label_list – the same list of clusters, but with actual node labels, not indices.
name – the file name for the current iteration; example: "mat 0.txt".
row – the number of rows in the variable 'list'.

The function scans through 'list' and 'label_list' and writes each cluster to a text file. The output format for each cluster is as follows (where X represents the index number of the current cluster):

```
Cluster X
Proteins:
Protein_1, Protein_2, Protein_3, …

Domains:
Domain_1, Domain_2, Domain_3, …
```

get_weights()
A function that constructs an array of edges, indicating which nodes are connected. The input arguments are:
e_val – the matrix of E-values, synchronized with 'bitvec_mat'.
bitvec_mat – the binary matrix indicating which domains are present in a given protein.
e_val_row – array containing the row coordinates of each E-value in 'e_val'.
e_val_col – array containing the column coordinates of each E-value in 'e_val'.
let – an array of index values corresponding to the domain nodes.

'edges_ar' is the only output argument, and consists of a list of sources and targets for each edge, as well as the edge type (undirected, in this case), and the weight of the edge.

The function gets the number of edges in the graph by summing all the 1's in 'bitvec_mat'. It then generates the array of edges, consisting of their sources and targets, as well as edge weights. The edge weights are calculated by a call to the function **log_calc()**, which is then multiplied by 10 (Gephi will not display easily-distinguished edge thicknesses otherwise).

log_calc()
This function calculates the weight of each edge. The sole input argument is:
num_in – the E-value of the current protein-domain match.

The function has two output arguments:
num_out – the calculated edge weight, based on the E-value input
bool – a Boolean that indicates whether or not an E-value is above an acceptable
threshold; should not be possible for this to be an issue.

The function finds the negative base-10 logarithm of the E-value and divides it by 100. 100 has been arbitrarily chosen as the best logarithm value, and has the effect of working

like a percentage. In the event an E-value's logarithm is greater than 100, it gets mapped back to 100.

gexf_out()

This function outputs the node and edge data in GEXF format. These files are then readily opened in Gephi to display the graph. The input arguments are:

edges_ar – the array of edges, giving their sources and targets, as well as weights.

edge_name – the file name for the current iteration; example: "mat 0.gexf".

nodes_out – the cell array containing the node information, in XML format.

num_dom – the number of domains.

The function accesses the data in 'edges_ar' and 'nodes_out' and writes the information to the output file in XML format. It is also responsible for spacing the nodes and cluster groups in the graph.

get_node_IDs()

A function to create a matrix that is synchronized to 'members', containing the node ID's to be used in the GEXF files. 'members' is the only input argument, and is a cell array consisting of one row for each protein in the HMMER data. Each column represents one of the E-value threshold iterations (one for each order of magnitude). The contents of each cell indicates which cluster that a given protein participates in during a given iteration. The output arguments are:

node_IDs – a matrix synchronized with 'members', and consisting of the node ID's for each cluster that will be used in the GEXF files.

node_labels – another cell array, synchronized with 'members', this time consisting of the node labels to be displayed; this will be used in the GEXF files.

node_ancestors – another cell array, synchronized with 'members', this time consisting of the labels of the nodes in the initial iteration that a given cluster in a subsequent iteration is derived from, due to the protein in that row being present in the initial cluster on the first iteration.

tier_clusters – a cell array with one column for each iteration; the contents of each column consist of a listing of each cluster node label.

The function works by iterating over the data shown in 'members', and accessing the information present in order to build the output variables that have been described above. This is for constructing the tier-based graph, which displays which clusters inherit proteins from which other clusters in the previous iteration.

get_clust_anc()

A function that creates a cell array showing which initial ancestral cluster a given cluster in the tier-based visualization ultimately is derived from. The input arguments are:

node_labels – a cell array created by **get_node_IDs()**; consists of the node labels to be displayed.

node_ancestors – a cell array created by **get_node_IDs()**; consists of the labels of the

nodes in the initial iteration from which a given cluster in a subsequent iteration inherited its proteins.

big_list – a cell array into which the variable 'list' from **domains()** is copied for each iteration in **prime_funct()**.

The output arguments are:

tier_clust_anc – this is a cell array in which each cell contains a list of all clusters in the initial iteration from which a given cluster inherited its proteins.

newick_tree – a two-dimensional cell array with *n* rows and *m* columns, where *n* is the number of proteins in the dataset, and *m* is the number of iterations of changes to the E-value threshold; each cell contains a Newick string describing the phylogeny-like relationships between related clusters.

newick_tree_lbl – a two-dimensional cell array identical to 'newick_tree', except that the Newick strings include node labels and branch lengths as well.

The function iterates over the contents of 'node_labels' and 'node_ancestors' to extract a list of the clusters in the initial iteration from which a given cluster is ultimately derived. It builds a list of the ancestors of each node, and places this information in the 'tier_clust_anc' output variable. Furthermore, the function also progressively builds Newick-format strings describing the relationships between related clusters, based on the merges that can be seen in the "tier" chart data from one iteration to the next.

ind_extract()

A function that accepts a string in the format "Cluster *X Y*", where *X* is the cluster index number, and *Y* is either 0 or a negative integer, extracts *X*, and converts it to a numerical value for output.

'clust_str' is the only input argument, representing the string from which the index number is to be extracted.

'clust_ind' is the only output variable, and is the cluster's index value, converted to numeric format.

de_newick()

This function removes the spaces and brackets from a Newick-format input string, places the cluster labels each in a separate cell of a cell array, and sorts the cell array before output.

'newick_str' is the Newick-format input string, and is the only input argument.

'out_newick' is the only output variable, and is a cell array consisting of the cluster labels in natural sort order / ASCII dictionary order (that is, "Cluster 10" would follow "Cluster 1" and precede "Cluster 2").

list_substr()
A function that accepts a pair of cell arrays, compares them element by element, and then removes entries that have been matched. It outputs both arrays, minus the deleted entries. If both arrays contain substrings of the other, they will be empty. The input and output arguments are the same (cell arrays 'A' and 'B').

inc_str()
A function that accepts a string in the format "*XXXX*:*Y*", where *XXXX* can be any string, followed by a final colon, followed by a numerical value *Y*. The function extracts *Y*, converts it to a numerical value, increments this value, converts it back into a string, replaces *Y* with the updated value, and presents the modified string as the output. 'input_str' is the only input argument, and is as described above.

'output_str' is the only output variable, and is the updated string, as described above.

newick_out()
This function accepts the un-labelled version of the set of Newick strings from **get_clust_anc()** and processes the set to extract the unique Newick strings. The function then places these strings in a cell array for output.
'newick_tree' is the only input, and is a two-dimensional cell array of *n* rows and *m* columns, where *n* is the number of proteins in the HMMER data, and *m* is the number of iterations of changing the E-value threshold.

'newick_list' is the only output, and is a one-dimensional cell array consisting of each unique Newick string taken from 'newick_tree'.

newick_out_lbl()
This function is an alternative version of **newick_out()**, meant specifically to handle Newick strings containing node labels and branch lengths. As such, it is necessary to call the **un_label()** function in order to get accurate results. Otherwise, input and output variables are identical to that of **newick_out()**.

un_label()
A function that accepts a Newick string with node labels and branch lengths (for example, "[*A*:*x*, *B*:*y*]*C*:*z*", where *A* and *B* are both leaf labels, *C* is a node label, and *x*, *y*, and *z* are branch lengths), and removes the node labels and branch lengths, leaving only the un-labelled Newick string (for example, "[*A*, *B*]").
'newick_str' is the only input, and once modified, is also the only output variable.

re_number()
This function looks at the list of clusters in a Newick string, finds if there are any duplicates, and then renumbers the duplicates so the first occurrence is called "Cluster *X*.1", the second is "Cluster *X*.2", and so on. The function also replaces some elements in the string to conform to true Newick format. There are two input arguments:

newick_list – a cell array of Newick strings to be processed.
bool – a Boolean variable that indicates of the input strings carry branch lengths and node labels or not; if true, **un_label**() needs to be called before **de_newick**().

The modified 'newick_list' array is the only output variable.

print_newick()
A function that prints the non-singleton Newick strings in the labelled and un-labelled versions of the tree lists to plain text files. Each tree is printed to its own file, following the naming convention of "tree*X*.txt" for the un-labelled tree, and "tree*X*_lbl.txt" for the labelled version. *X* is the index number within the list that the string came from. The input variables are:
newick_list – a cell array containing the un-labelled versions of the Newick output trees.
newick_list_lbl – a cell array containing the same Newick output trees, but with added node labels and branch lengths.

The function prints the (non-singleton) contents of each cell to its own file. It is formatted so that the entire Newick string is printed on a single line.

get_edges()
A function to create a list of the edges to be used in the tier-based graph. The input arguments are:
node_IDs – a matrix consisting of the node ID's for each cluster that will be used in the GEXF files.
members – a cell array that displays which cluster a given protein participates in during a given iteration.

'tier_edges' is the only output variable, and is a two-row array indicating the edges of the tier-based graph. The top row is the source node ID, and bottom row is the target node ID.

The function iterates over 'node_IDs'. Each time there is a protein participating in clusters in two adjacent columns of the matrix, it indicates there should be an edge between those clusters in the tier-based graph. Each edge is added only once.

mergeSort2()
This function is another modified implementation of Merge Sort taken from Rosetta Code [36]. This implementation has been modified to accept the array to be sorted ('list'), which is expected to be two-dimensional and consists of two rows. The top row of 'list' is meant to be sorted, but the associated second row values are to remain synchronized with the sorted top row values. This is a similar approach to the first version of Merge Sort this program uses, but has a different expected input format.

The output of this function is the same two-row array, now sorted in ascending order based on the values in the top row.

tier_gexf()
This function outputs the node and edge data for the tier-based graph in GEXF format. These files are then readily opened in Gephi to display the graph. The input arguments are:
node_IDs – a matrix consisting of the node ID's for each cluster that will be used in the GEXF files.
tier_edges – a two-row array indicating the edges of the tier-based graph; top row is the source node ID, and bottom row is the target node ID.
tier_clusters – a listing of each cluster node label for each iteration.

The function accesses the data in 'node_IDs', 'tier_edges', and 'tier_clusters' and writes the information to the output file in XML format. It is also responsible for spacing the nodes and tiers in the graph.

A.2 A Note on File Format

A.2.1 Input Files

HMMER Data
The file extension may vary (.out or .txt files are normally used), as this will be specified in the program input parameters. The file contents must be provided in typical HMMER output format. However, in this case, the files have been edited slightly to remove the field headers and program and setting details from the top and bottom of the file. The only contents that remain are the lines that contain the actual HMMER data that is to be processed. The fields need to be in the following order: <target name>, <accession>, <tlen>, <query name>, <qlen>, <E-value>, <score>, <bias>, <number>, <of>, <c-Evalue>, <i-Evalue>, <score>, <bias>, <from>, <to>, <from>, <to>, <from>, <to>, <acc>, and <description of target>. It should be mentioned that in the HMMER files, there is also an <accession> field in between <query name> and <qlen>, however, in each of the files made available from Shah's original data, this field is left blank (indicated by a "-" character). For the purposes of field indexing and reformatting of the HMMER file, this field was ignored and deleted. Taking this indexing into consideration, the fields of interest are therefore the 1st, 4th, and 12th (<target name>, <query name>, and <i-Evalue>, respectively). These fields carry the information of 1) the domain names, 4) the protein names / accession numbers, and 12) the E-values of the match between the domain and protein sequences.

A.2.2 Output Files

Reformatted HMMER Data
The typical HMMER data file will contain white space between fields. However, this white space is not uniform, and may consist of multiple individual spaces or tabs, with the intent of lining each field up into a given column on the page. However, this is problematic for importing the data into memory, and so it is reformatted to replace all white space between fields with a single tab. The blank <accession> field mentioned above is also removed during this stage.

Row and Column Data

There are two plain text files ("rows.txt" and "cols.txt") which match the row and column indices to the protein accession numbers and domain names they represent, respectively. The following is an example of this content:

```
rows.txt:                    cols.txt:
1:   NP_598838.3             1:   RRM_5
2:   NP_058038.2             2:   RRM_6
3:   NP_001123624.1          3:   RRM_1
4:   NP_796052.2             4:   Nup35_RRM_2
5:   NP_056627.1             5:   PWI
     ...                          ...
```

Cluster Membership Lists

These files are in plain text format, and are named based on the theme "mat *X*", where *X* represents the order of magnitude of E-value that was just deleted before that file was generated. The file of the initial iteration (before the E-value threshold is used) is always "mat 0.txt", including cases where the start E-value parameter is specified to be lower than the highest E-value in the HMMER data. These files provide the protein membership of each cluster, as well as a listing of the domains present in that cluster. An example of the file contents follows:

```
Cluster 1
Proteins:
NP_598838.3

Domains:
RRM_5, RRM_6, RRM_1, Nup35_RRM_2, PWI, zf-CCCH, RGS-like, DUF2785


Cluster 2
Proteins:
NP_058038.2

Domains:
RBD, RGS, GoLoco, PSD4
...
```

GEXF Files

These files are meant to be used in Gephi, and encode the node and edge data for the bipartite graphs in a way very similar to XML encoding. They are also named according to the same theme as that used for the cluster membership lists ("mat 0.gexf", etc.).

Tier-Based Visualization

This GEXF file ("tiers.gexf") visualizes the clusters by reducing each cluster to an individual node, and displaying them in tiers, with each tier corresponding to an order of magnitude of E-value, just as the cluster membership and associated GEXF bipartite visualizations do. If an edge exists between clusters in adjacent tiers, it indicates that the

subsequent clusters "inherit" at least one of their protein members from the previous clusters.

Newick Trees

These are a series of plain text files that each carry one of the Newick trees found in the procedure. There are labelled and un-labelled versions of the trees. These files can be easily imported into a Newick tree visualization program (TreeView was used for these tests) for viewing.